Reorganisation von Datenbanken

- Auslöser, Verfahren, Nutzenermittlung -

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik der
Friedrich-Schiller-Universität Jena

von Diplom-Informatiker Stefan Dorendorf
geboren am 23. März 1966 in Glauchau

Gutachter:

1. Professor Dr.-Ing. Klaus Küspert
2. Professor Dr. habil. Stefan Braß
3. Professor Dr.-Ing. habil. Thomas Ruf

Tag der letzten Prüfung des Rigorosums: 11. Oktober 2006

Tag der öffentlichen Verteidigung: 20. Oktober 2006

Kurzfassung

Beim Betrieb großer datenbankgestützter Anwendungssysteme kommt es im Laufe der Zeit oft zu einer nachhaltigen Verschlechterung der Systemleistung durch in den physischen Speicherungsstrukturen entstehende Degenerierungen (z.B. eingestreuter Freiplatz, Überlaufbereiche, migrierte Tupel, nicht mehr vorliegende Daten-Clusterung oder -sortierung). Die Beseitigung von Degenerierungen kann mit *Datenbankreorganisationen* erfolgen. Solche Wartungsarbeiten sind allerdings, selbst bei Online-Durchführung, meist mit Einschränkungen im normalen Datenbankbetrieb verbunden, die von einer temporären Verschlechterung der Systemleistung bis hin zu eingeschränkter Verfügbarkeit von Daten während der Reorganisationsdurchführung reichen. Diese Einschränkungen kollidieren immer mehr mit wachsenden Verfügbarkeitsanforderungen von Datenbank-Management-Systemen (DBMS). Eine Möglichkeit zur Verringerung der negativen Auswirkungen stellt die *sorgfältige Auswahl und Priorisierung von Reorganisationsmaßnahmen* dar. Damit können besonders nutzbringende Maßnahmen identifiziert und bevorzugt ausgeführt, andere zurückgestellt oder unterlassen werden.

In dieser Arbeit wird das Themengebiet der Reorganisation von Datenbanken zunächst allgemein im Kontext relationaler sowie in Ansätzen auch objektrelationaler Datenbanksysteme betrachtet. Weiterhin werden gängige physische Speicherungsstrukturen und das Verhalten von darauf angewendeten Operationen beschrieben. Darauf aufbauend lassen sich auch Entstehung und Auswirkungen von Degenerierungen darstellen. Weiterhin werden Methoden und Vorgehensweisen zur Durchführung von Datenbankreorganisationen betrachtet.

Der Schwerpunkt der Arbeit liegt auf der *Quantifizierung des* von Datenbankreorganisationen in einer konkreten Systemumgebung *zu erwartenden Nutzens* bezüglich der Systemleistung. Dies geschieht im Vorfeld der eigentlichen Durchführung (als Nutzenvorhersage). Die Quantifizierung erfolgt rechnerisch auf der Basis von Informationen über die Datenbank-Workload und den Zustand der physischen Speicherungsstrukturen. Die Methoden können ohne größere Probleme auf vorhandene Systeme aufgesetzt werden. Zur Vermeidung von Systemabhängigkeiten wird ein neutrales Speicher- und Verhaltensmodell von DBMS entwickelt.

Methoden zur Abschätzung des mit der Reorganisationsdurchführung verbundenen Aufwands werden ebenfalls betrachtet. Damit kann dem zu erwartenden Nutzen auch der durch die Reorganisation verursachte Aufwand gegenübergestellt werden. Mit Hilfe eines Optimierungsverfahrens wird zudem die Auswahl und Priorisierung von Reorganisationskandidaten (etwa einzelner Tabellen) unterstützt.

Weiterhin wird ein Ansatz zur Bewertung der Effizienz unterschiedlicher physischer Repräsentationen von Datenbankobjekten in einer konkreten Systemumgebung vorgestellt.

Die Ergebnisse von anhand der Produkte DB2, Informix und Oracle durchgeführten Untersuchungen sowie von Aufwandsmessungen, die mittels prototypischer Implementierungen für Oracle realisiert wurden, werden präsentiert. Ein wesentliches Anwendungsgebiet der erarbeiteten Konzepte stellt die Erweiterung von Werkzeugen zur (weitgehend automatisierten) Administration von Datenbanksystemen dar.

Danksagung

An dieser Stelle möchte ich all jenen danken, die zum erfolgreichen Abschluss dieser Arbeit beigetragen und mich auf vielfältige Weise unterstützt haben.

Besonderer Dank gilt zunächst meinem Betreuer, Herrn Professor Dr. Klaus Küspert, der mir zu jeder Zeit mit Rat und Tat zur Seite stand, in den vergangenen Jahren stets Verständnis für meine berufliche Situation bewies und mich stets ermunterte, erreichte Ergebnisse auch zu publizieren. Besonders wertvoll waren für mich die vielen fachlichen Anregungen, die sich aus den oft auch kritischen Diskussionen in Jena ergaben.

Ich möchte den Gutachtern Herrn Professor Dr. Stefan Braß und Herrn Professor Dr. Thomas Ruf für ihre Mühe bei der Bewertung der doch etwas umfangreicheren Arbeit sowie für ihre Einschätzungen danken.

Weiterhin gilt mein Dank dem Vorsitzenden der Prüfungskommission, Herrn Professor Dr. Walter Alt sowie den Mitgliedern der Prüfungskommission Herrn Professor Dr. Werner Erhard, Frau Professor Dr. Birgitta König-Ries, Herrn Professor Dr. Hans-Dietrich Hecker sowie Herrn Dr. Hermann Döhler.

Bedanken möchte ich mich auch bei Frau Franziska Wieczorek, Herrn Oliver Klinger, Frau Anja Heisrath, Frau Anna Singer, Herrn Kai Ganskow und Herrn Pawel Williams, auf deren Arbeiten Teile meiner Dissertation aufbauen.

Dank gilt auch meinen Kollegen von der Berufsakademie Gera, hier besonders den ehemaligen Direktoren Herrn Professor Dietrich Tesmer, Herrn Professor Dr. Gerd Lämmermann und Herrn Professor Dr. Benno Kaufhold für die geleistete Unterstützung.

Meiner Familie, die mich während der vergangenen Jahre immer wieder zur Weiterarbeit ermunterte, gebührt Dank, besonders meiner Schwester Frau Dr. Heike Kühn dafür, dass sie mir bei der „Entwirrung“ einiger furchtbar verschachtelter Sätze in der fast fertigen Version der Arbeit geholfen hat. Ich danke auch meinen Kindern Annegret und Johannes, die öfter auf gemeinsame Stunden verzichten mussten.

Zum Schluss möchte ich dem wichtigsten Menschen danken, meiner Frau Konstanze. Sie hat von allen Genannten den wichtigsten Beitrag zum erfolgreichen Abschluss dieser Arbeit geleistet. Ihre kritischen fachlichen Anmerkungen und ihre Hilfe in redaktionellen Dingen sind für mich von unschätzbarem Wert. Mit schier unendlicher Geduld hat sie alle Höhen und Tiefen bei der Erstellung dieser Arbeit mit mir durchlebt und mir jederzeit den notwendigen Freiraum verschafft. Ohne sie wäre diese Arbeit nicht möglich gewesen.

Vielen Dank

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Herausforderungen, die sich beim Betrieb von größeren datenbankgestützten Anwendungssystemen ergeben, sind in den vergangenen Jahren trotz immer höherer Leistungsfähigkeit der den Systemen zu Grunde liegenden Hardware nicht geringer geworden – im Gegenteil. Die Betreiber sehen sich mit ständig steigenden Datenmengen, mit teilweise stark angestiegenen Benutzerzahlen und damit mit einem stark gestiegenen Transaktionsvolumen sowie hohen Verfügbarkeitsanforderungen konfrontiert. Durch Bemühungen der Hersteller von Datenbank-Management-Systemen (DBMS) und von Drittanbietern wurden Mechanismen zur Durchführung von Wartungsarbeiten an Datenbanken (z.B. Backup-Erstellung, Archivierung von Daten, Konfiguration von DBMS, *Datenbankreorganisation*) zur Verfügung gestellt, die online und parallel zum laufenden Betrieb ausgeführt werden können. Trotzdem sind solche Wartungen mit Einschränkungen des normalen Datenbankbetriebs verbunden, denn der Wartungsaufwand bleibt erhalten und kann auch nicht „versteckt“ werden. Hinzu kommt der Aufwand zur Aufrechterhaltung der nahezu vollständigen Verfügbarkeit der Daten. Durch Verbesserungen im Speicher-Management wird die Entwicklung von Degenerierungen in Speicherungsstrukturen verlangsamt, was bspw. zu einer Verlängerung von Reorganisationsintervallen führen soll. Durch die Steigerung der Nutzungsraten (rapide zunehmende Workload) wird der Erfolg hier aber oft gemindert.

In den letzten Jahren ist durch die Weiterentwicklung datenbankgestützter Softwarelösungen und durch neue Anwendungen näher am Endnutzer die *Zahl der Transaktionen stark gestiegen,* die gegen die zu Grunde liegenden Datenbanken ausgeführt werden. Im Bankbereich z.B. steigt die Zahl anfallender Buchungen durch die erhebliche Zunahme des bargeldlosen Zahlungsverkehrs [Joc05]. Die Zugangsmöglichkeit über das Internet und Automaten hat die Zahl der Banktransaktionen (inkl. Kontoabfragen etc.) ebenfalls stark ansteigen lassen. Die Transaktionszahlen der Volks- und Raiffeisenbanken in Deutschland stiegen allein im Jahr 2000 im Electronic-Cash-System um 25% [BVR01] und im Geldkarten-System im Jahr 2002 um 28% [BVR03]. Auf dem Gebiet der Telekommunikation ist im Mobilfunkbereich die Zahl der Nutzer in den letzten Jahren enorm angestiegen. Die Vermittlung der Gespräche und das Sammeln der Verbindungsdaten erfolgen hier über datenbankgestützte Softwarelösungen. Große Logistikunternehmen betreiben u.a. Systeme zur Sendungsverfolgung, die jetzt Informationen in zentralen Datenbanken speichern, die an Sortier- und Weiterleitungsstellen anfallen und die früher nur dort lokal verwendet und nach deren „Gebrauch“ gelöscht wurden.

Die Deutsche Post World Net betreibt im Unternehmensbereich BRIEF ein System zur Sammlung und Analyse produktionsbezogener Daten mit ca. 1000 Tabellen, einer derzeitigen Datenbankgröße von ca. 5 Terabyte und täglichen Datenbewegungen von ca. 4,5 Gigabyte [DPW05]. In [Aue04] wird die Datenbank der US Land Registry-Behörde mit einer Größe von damals 18,3 Terabyte als zu dieser Zeit größte operative Datenbank angegeben. Kernstück des Systems für Kundenservice, Abwicklung von

Störungsmeldungen und Gebührenerfassung der Britisch Telecom in Großbritannien ist eine Datenbank mit einer Größe von 11,7 Terabyte im Jahr 2003. Aber auch im produzierenden Bereich werden die an den Produktionsanlagen anfallenden Daten verstärkt für die Sicherung einer gleichbleibenden Qualität, zur frühzeitigen Erkennung von Problemen und zur Qualitätsverbesserung inklusive Revisionssicherheit genutzt. Im Bereich großer integrierter betrieblicher Informationssysteme (z.B. SAP-Systeme) sind Datenbanken in der Größenordnung von mehren hundert Gigabyte und mehr sowie mit vielen tausend Nutzern keine Seltenheit.

Mit *steigender Nutzerzahl* und damit wachsendem Transaktionsvolumen steigt (von eher retrievalorientierten Data-Warehouse-Systemen abgesehen) bei OLTP-Systemen i.d.R. auch der Wartungsbedarf der Datenbanken. Einfüge-, Lösch- und Änderungsoperationen führen u.U. zu Degenerierungen in den zur Speicherung der Daten verwendeten internen physischen Strukturen. Bei der oft üblichen Speicherung von Daten in als Heap organisierten Bereichen werden bei Einfügeoperationen evtl. wünschenswerte interne Sortierreihenfolgen i.d.R. nicht beachtet, Speicherbereiche für logisch zusammengehörige Daten müssen über Datenträger verstreut reserviert werden etc. Bei Löschoperationen entstehen oft weitere Lücken im Datenbestand, weil der frei werdende Speicher nicht ohne größeren Aufwand sofort wiederverwendet werden kann. Solche Degenerierungen führen also zu einem erhöhten Speicher- und Verarbeitungsaufwand und damit zu höheren Kosten (allg. Performance-Einbußen). In Datenbestände eingestreute Freiplatzfragmente müssen beispielsweise bei Suchoperationen oftmals mit verarbeitet werden. Damit kommt es im Laufe der Zeit zu einer schleichenden Leistungsverschlechterung. Eine Beseitigung vorhandener Degenerierungen kann und muss durch *Datenbankreorganisationen* erfolgen.

Wachsende Datenmengen führen zu Performance-Problemen, die u.U. durch Reorganisationsmaßnahmen (wie bspw. die Verteilung von Daten auf verschiedene Datenträger) gemildert werden können. Allerdings wächst mit der Datenmenge i.d.R. auch der Aufwand für solche Wartungsarbeiten mit einem entsprechenden Zeitbedarf. Die Archivierung von Daten ist eine Methode, das Wachstumstempo großer Datenbanken zumindest abzumildern [Her97, Sch99, Ste02]. Dabei werden Daten in zwei Gruppen unterteilt. Daten, die im operativen Betrieb oftmals verwendet werden und eine hohe Aktualität aufweisen, bilden die Gruppe der operativen Daten. Daten, auf die nur (noch) selten zugegriffen werden muss, die aber langfristig zur Verfügung stehen sollen (z.B. im Rahmen der gesetzlich vorgeschriebenen Produkthaftung), bilden die Gruppe der nichtoperativen Daten. Diese Daten können somit zwar nicht endgültig gelöscht, aber zur Entlastung der operativen Datenbanken in Archive ausgelagert (verschoben) werden. Dieses Verschieben schließt ein Löschen aus der operativen Datenbank ein. Dadurch entstehen dort innerhalb der Datenbestände Freispeicherlücken und andere Degenerierungen, durch die wiederum ein *Reorganisationsbedarf* entsteht [Ste02].

Hohe Verfügbarkeitsanforderungen lassen die für Wartungsarbeiten zur Verfügung stehenden Zeitfenster, innerhalb derer zumindest Teile der Datenbanken nicht verfügbar sind, schrumpfen oder nahezu vollständig verschwinden. Im Werk des

Halbleiterherstellers AMD in Dresden steht lt. [Ric03] z.B. jährlich lediglich eine geplante Unterbrechungszeit von einer Stunde für eventuelle offline durchzuführende IT-Wartungsarbeiten (Hardware-Aufrüstung, Release-Wechsel von Betriebssystem oder DBMS usw.) zur Verfügung. Auf dem Gebiet der Telekommunikation werden datenbankgestützte Softwaresysteme längst nicht mehr nur für Kundenverwaltungs- und Abrechnungssysteme eingesetzt. Die Verwaltung moderner Telekommunikationsdienste (sog. Mehrwertdienste wie Service 700, Universal Number, Freephone, Televotum, Virtual Private Network usw.), die über „virtuelle" Telefonnummern mit Vorwahlnummern wie bspw. 0700, 0180, 0800 erreichbar sind, erfolgt über datenbankgestützte Softwaresysteme. Auch für die Vermittlung zwischen den Endpunkten durch die zu den Diensten gehörenden Verkehrsführungsprogramme sind Zugriffe auf die entsprechenden Datenbestände notwendig. Die Herstellung von Verbindungen im Bereich der mobilen Kommunikation ist ohne Zugriffe auf Datenbanken (im GSM-System sind dies u.a. das Visitor Location Register und das Home Location Register) nicht möglich. Von solchen Systemen wird somit eine Verfügbarkeit von nahezu 100% gefordert. Auf Grund der hohen Verfügbarkeitsanforderungen müssen Wartungsarbeiten jeglicher Art häufig parallel zum normalen Betrieb durchgeführt werden. Damit verbundene Einschränkungen des Datenbankbetriebs (bspw. durch zusätzliche Systemlast verursachte Performance-Einbußen) müssen dabei auf ein Minimum begrenzt werden. Hier ist eine wohlüberlegte Planung zwingend erforderlich. Wartungsarbeiten, die einen entsprechenden Nutzen erwarten lassen oder aus unterschiedlichen Gründen als „dringend" eingestuft werden, sind durchzuführen, unnötige oder noch nicht nötige Arbeiten müssen vermieden bzw. zurückgestellt werden. Hier ergibt sich die Notwendigkeit, den zu erwartenden Nutzen der Wartungsarbeiten vorab möglichst genau einzuschätzen. Dazu ist es wichtig, das Ausmaß vorhandener Degenerierungen einschätzen und die daraus resultierenden Auswirkungen quantifizieren zu können.

Das Thema Datenbankreorganisation an sich ist natürlich nicht brandneu. In der Praxis sind Datenbankadministratoren seit Jahren damit konfrontiert und gezwungen, Lösungen zu finden, bei denen die Beeinflussung des normalen Datenbankbetriebs während einer Reorganisation möglichst gering ist. Auf Grund

- immer größer werdender Datenbanken auf der einen Seite und hoher Verfügbarkeitsanforderungen auf der anderen Seite,
- größerer Freiheitsgrade beim physischen Datenbankentwurf durch neue Speicherungskonzepte bzw. durch neue Kombinationsmöglichkeiten und
- der Tatsache, dass Wartungsarbeiten wegen des damit verbundenen teilweise hohen Aufwands nicht einfach „auf Verdacht" (online oder offline) ausgeführt werden können [Sch04],

ergibt sich eine Verschärfung der Problematik. Vorhandene Problemlösungen in Praxis *und* Wissenschaft sind nach wie vor nicht immer überzeugend. Es erscheint daher opportun, dieses Thema wissenschaftlich weiter aufzugreifen. Viele der seit längerer Zeit diskutierten Datenbankthemen sind vom Grunde her noch immer aktuell, auch wenn sich vielleicht die Erscheinungsformen geändert haben und die Methoden zur (temporären) Problemlösung weiterentwickelt wurden. Dies zeigt sich

auch an den wachsenden Anstrengungen der Anbieter von DBMS-Produkten zur Verbesserung von Werkzeugen zur Datenbankadministration [AF06] und dem Trend der letzten Jahre zur verstärkten Entwicklung und Verbesserung von *Self-Managing Database Systems* [Rab06, Ruf05].

1.2 Ziele der vorliegenden Arbeit

Ein Ziel der Arbeit ist es, einen ausführlichen Überblick über das Themengebiet der Datenbankreorganisation zu geben. Auch wenn bei einer weiten Fassung des Begriffs Datenbankreorganisation Aspekte der logischen Datenorganisation ebenfalls von Bedeutung sind, liegt der Schwerpunkt der Arbeit auf der physischen Organisation der Daten auf Sekundärspeichermedien. Für die Betreiber von Datenbanksystemen ist ein Überblick über die Auslöser von Datenbankreorganisationen wichtig. Dazu werden die gängigen internen Speicherungsstrukturen und die darauf angewendeten Operationen betrachtet. Zur Verzögerung der Entstehung von Degenerierungen ist eine geeignete Abbildung der logischen Datenstrukturen auf die zur Verfügung stehenden internen Speicherungsstrukturen nötig, bei der auch die auf die Datenbankobjekte angewendeten Operationen zu berücksichtigen sind. Hierzu zählt auch eine überlegte Festlegung der die interne Speicherung beeinflussenden Parameter. Bezüglich der internen Datenorganisation existieren trotz teilweise unterschiedlicher Begriffsverwendung und unterschiedlichen Implementierungsumfangs bei DBMS-Produkten signifikante Gemeinsamkeiten. Eine Vereinheitlichung der Begriffe und der Beschreibung der internen Speicherungsstrukturen, wie sie für die logische Datenorganisation durch Datenmodelle und Normierung (bspw. die SQL-Norm) vorgenommen wurde, erscheint deshalb für die Zwecke der vorliegenden Arbeit sinnvoll. Aber auch bei gut abgestimmten internen Speicherungskonzepten lassen sich Datenbankreorganisationen i.d.R. nicht vollständig vermeiden. Die Wahl einer geeigneten Methode kann den Aufwand für die Reorganisation und die damit verbundenen Einschränkungen im normalen Datenbankbetrieb verringern.

Das Kernstück der Arbeit bildet die Bewertung des Nutzens von Reorganisationen interner Speicherungsstrukturen. Möglichkeiten der physischen Speicherung von Daten und Reorganisationsverfahren werden in der Literatur ausführlich diskutiert. Werkzeuge von DBMS-Herstellern und Drittanbietern unterstützen Datenbankadministratoren (DBA) bei der Lokalisierung von Degenerierungen und bei der Reorganisationsdurchführung. Dass die Durchführung einer Datenbankreorganisation einen vom Degenerierungsgrad abhängigen Nutzen erwarten lässt, ist ebenfalls unstrittig. Der Degenerierungsgrad kann aus statistischen Daten, die den Zustand physischer Speicherungsstrukturen beschreiben, ermittelt werden. Was fehlt, sind Informationen darüber, wie hoch der zu erwartende Reorganisationsnutzen insbesondere bezüglich der Systemleistung ist.

Ziele von Datenbankreorganisationen sind im Wesentlichen die Verbesserung der Systemleistung und eine verbesserte Speicherplatzausnutzung. Die zu erwartende Verbesserung der Speicherauslastung (Einsparung an erforderlichem Speicherplatz) kann über den Degenerierungsgrad relativ einfach bestimmt werden. Der Nutzen

bezüglich der Systemleistung ist aber nicht nur vom Degenerierungsgrad abhängig. Hier hat der auf die Strukturen angewendete Operationsmix (Workload) wesentlichen Einfluss. Die Auswirkungen von Degenerierungen unterscheiden sich bei verschiedenen Operationen. Sollen also die Auswirkungen einer Reorganisation bezüglich der Systemleistung beziffert werden, so müssen zusätzlich Informationen über die auf die Speicherungsstrukturen angewendete Workload in die Betrachtungen einbezogen werden.

Mit der in dieser Arbeit vorgeschlagenen Methode kann, durch die Einbeziehung eines Workload-Protokolls und von Informationen über den Zustand der physischen Speicherungsstrukturen, die mit der Reorganisation von Datenbankobjekten (Tabellen, Indexe etc.) verbundene Veränderung des zur Workload-Abarbeitung notwendigen I/O-Aufwands quantifiziert werden (I/O-Kosteneinsparungen). Dies stellt eine Erweiterung der derzeit gebräuchlichen Methoden dar, die für Empfehlungen zur Reorganisationsdurchführung lediglich auf der Auswertung von statistischen Informationen über den Zustand der betrachteten Speicherungsstrukturen beruhen. Die Anwendung kann in einer konkreten Systemumgebung erfolgen, unter Berücksichtigung der dort vorhandenen Gegebenheiten. Die vorgeschlagene Methode greift dabei größtenteils auf vorhandene Daten und Komponenten zurück, wie auf Statistikdaten im Datenbankkatalog und auf den Anfrageoptimierer. Möglichkeiten zur Protokollierung der gegen eine Datenbank gerichteten Anweisungen existieren für viele DBMS-Produkte ebenfalls. Im Rahmen der Anfrageoptimierung werden auch Kostenbetrachtungen zu auszuführenden Datenbankoperationen angestellt.

Dem von einer Reorganisation zu erwartenden Nutzen soll auch der damit verbundene Aufwand gegenübergestellt werden. Dazu werden drei verschiedene Vorgehensweisen zur Reorganisationsdurchführung von der (I/O-)Kostenseite her betrachtet. Eine Erweiterung vorhandener Systeme um die vorgeschlagenen Konzepte zur Nutzen- und Kostenermittlung von Datenbankreorganisationen ist insbesondere für die Hersteller von DBMS-Produkten mit relativ geringem Aufwand möglich. Notwendige Erweiterungen bestehender Konzepte werden beschrieben.

Ziel ist es, den DBA in die Lage zu versetzen, die konkret zu erwartende Verbesserung der Systemleistung zu quantifizieren und in Reorganisationsentscheidungen einfließen zu lassen. Werden die aus einzelnen Nutzenanalysen gewonnenen Informationen über einen Zeitraum hinweg gespeichert, können die Entwicklung vorhandener Degenerierungen und des von einer Datenbankreorganisation zu erwartenden Nutzens verfolgt [Wil04] und Trends abgeleitet werden. Abhängig von diesen Trends können Analyseintervalle weitgehend automatisch bestimmt und dem DBA Vorschläge für durchzuführende Reorganisationen unterbreitet werden. Wir sehen dies auch als Erweiterungsmöglichkeit der derzeit bei DBMS-Herstellern und Drittanbietern verstärkt in Entwicklung befindlichen Werkzeuge zur automatisierten Ausführung von Datenbankadministrationsaufgaben an.

Die Erweiterung von relationalen DBMS um Konzepte der Objektorientierung zu objektrelationalen Datenbank-Management-Systemen (ORDBMS) lässt künftig eine größere Vielfalt an Speicherungsstrukturen und Kombinationsmöglichkeiten

(größere Zahl an „Stellschrauben“) erwarten oder zumindest erhoffen – evtl. könnte man hier aber auch von „befürchten“ sprechen. Das erschwert gleichzeitig für den DBA die Abbildung der logischen Datenorganisation auf interne Strukturen [Ska06]. Der Aufwand für die Abarbeitung der gegen die Datenbankobjekte gerichteten Workload ist von der internen Datenorganisation abhängig. Die physisch beieinanderliegende Speicherung (Clusterung) von Daten, die auf der logischen Ebene zueinander in Beziehung stehen, kann den Aufwand für deren gemeinsame Verarbeitung gegenüber der bei rein relationalen Systemen verbreiteten separaten Speicherung von Tabellen u.U. deutlich vermindern. Eine Unterstützung für den DBA beim physischen Datenbankentwurf ist besonders wünschenswert. Ein Ansatz dazu, bei dem zu einem großen Teil die Konzepte des Verfahrens zur Quantifizierung des Nutzens von Datenbankreorganisationen verwendet werden, wird in dieser Arbeit ebenfalls kurz aufgezeigt.

1.3 Aufbau der Arbeit

In *Kapitel 2* werden die Grundlagen der Datenbankreorganisation behandelt und wichtige Begriffe eingeführt. Neben einer Klassifizierung der Gründe von Datenbankreorganisationen werden Ziele sowie Anforderungen betrachtet, die aus praktischer Sicht an den Reorganisationsprozess gestellt werden. Dabei wird auch auf Probleme eingegangen, die sich im laufenden Datenbankbetrieb ergeben. Die Betrachtung unterschiedlicher Ebenen als Ansatzpunkte von Datenbankreorganisationen und unterschiedlicher Reorganisationsformen erfolgt zunächst aus dem Blickwinkel der Funktionalität und (noch) nicht bezüglich möglicher Implementierungstechniken. Diese werden später noch im Detail erläutert.

Das Thema Datenbankreorganisation ist eng mit der internen Speicherung von Daten verbunden. *Kapitel 3* bietet eine genauere Behandlung verbreiteter Speicherungsstrukturen. Degenerierungen in diesen Speicherungsstrukturen, die meist Datenbankreorganisationen erfordern, entstehen durch Änderungsoperationen (Einfügen, Löschen, Ändern von Daten). Daher wird auch das Verhalten von Änderungsoperationen betrachtet, um die Entstehung der unterschiedlichen Degenerierungsformen darzustellen. Hierbei werden ebenfalls Mechanismen beleuchtet, die die Entwicklung von Degenerierungen verzögern können.

Eine Darstellung des aktuellen Stands der Forschung und der verfügbaren DBMS-Produkte erfolgt in *Kapitel 4*. Schwerpunkt sind dabei gängige Verfahren und Methoden zur Reorganisation von Datenbankobjekten. Neben Beiträgen aus der Literatur wird die Leistungsfähigkeit von am Markt verfügbaren Überwachungs- und Administrationswerkzeugen von DBMS-Herstellern und Drittanbietern in die Betrachtungen einbezogen. Dabei werden Reorganisationsbedarfsanalysen und Reorganisationsdurchführung getrennt behandelt. Diese Trennung der Aufgaben spiegelt sich auch im Aufbau der Werkzeuge wider.

Die interne Speicherorganisation ist bei den verbreiteten relationalen DBMS-Produkten in großen Teilen ähnlich. Dies gilt auch für die Implementierung der auf diesen Strukturen arbeitenden Operationen. Unterschiede finden sich bezüglich der

verwendeten Begriffswelt, realisierter Konzepte (z.B. zur Partitionierung [Now01a] oder Clusterung von Daten) sowie implementierungstechnischer Details. Auch der Aufbau der Datenbankkataloge und der darin enthaltenen Informationen, die den Zustand interner Speicherungsstrukturen beschreiben, ist unterschiedlich. Dies ist auch darauf zurückzuführen, dass physische Aspekte der Datenspeicherung von der Normierung nicht erfasst werden. Um die weiteren Teile dieser Arbeit möglichst unabhängig von konkreten Produkten halten zu können, wird in *Kapitel 5* ein verallgemeinertes Modell der Speicherungsstrukturen und der darüber implementierten Operationen von Datenbank-Management-Systemen entwickelt. Teil dieses Modells ist auch ein Vorschlag zur einheitlichen Darstellung der die internen Speicherungsstrukturen beschreibenden Informationen (das sog. eInformationsschema).

In *Kapitel 6* wird die Ermittlung des Nutzens von Datenbankreorganisationen detailliert behandelt. Die Betrachtungen erfolgen bezüglich der Speicherkosten und der Verarbeitungskosten. Die auf Speicherkosten bezogenen Nutzenbetrachtungen können durch die Auswertung von statistischen Informationen über den Zustand der Speicherungsstrukturen erfolgen. Zur Quantifizierung des Nutzens in Bezug auf die Systemleistung werden zunächst Verfahren zur Gewinnung der benötigten Informationen über die gegen die Datenbankobjekte gerichtete Workload und das Zusammenspiel der beteiligten Komponenten beschrieben. Bei der vorgeschlagenen Methode wird der zu erwartende Nutzen ausgehend von statistischen Informationen über den Zustand der internen Speicherungsstrukturen von Datenbankobjekten und den Informationen über die Workload unter Nutzung von geeigneten Kostenfunktionen errechnet. Ansatzpunkte bieten hier Mechanismen zur kostenbasierten Anfrageoptimierung. Erweiterungen und Änderungen, die an den Kostenfunktionen für unsere Zwecke vorgenommen werden müssen, werden beschrieben. Die Abfolge der einzelnen Arbeitsschritte wird dargestellt und es werden Alternativen mit ihren Vor- und Nachteilen aufgezeigt.

Bei der Entscheidung über die Durchführung von Datenbankreorganisationen sollten dem zu erwartenden Nutzen auch die durch die Reorganisation verursachten Kosten gegenübergestellt werden. In *Kapitel 7* werden drei verschiedene Methoden zur Reorganisation von Datenbankobjekten näher betrachtet und Berechnungsfunktionen zur Abschätzung der Kosten entwickelt, die bei deren Anwendung zur Reorganisation von Datenbankobjekten zu erwarten sind.

Besonders bei großen Datenbanken kann eine rein manuelle Auswahl der Reorganisationskandidaten (tausende, zehntausende Tabellen, Indexe, ...) durch den DBA, bedingt durch die große Menge an Datenbankobjekten, u.U. nur nach Gutdünken erfolgen. Zur Unterstützung wird in *Kapitel 8* eine Methode vorgeschlagen, ausgehend von den Nutzen- und Kostenwerten der potenziellen Reorganisationskandidaten diejenigen auszuwählen, durch deren Reorganisation bei einem vorgegebenen Kostenlimit der maximale Nutzen erzielt wird. Zur Problemlösung wird ein Branch-and-Bound-Verfahren eingesetzt.

Neben der Beseitigung von Degenerierungen kann auch die Änderung der internen Repräsentation von Datenbankobjekten als eine Form der Reorganisation von

Datenbanken angesehen werden. Bei objektrelationalen Datenbanksystemen ergeben sich hier noch deutlich größere Herausforderungen als bei rein relationalen Systemen. Die Vielzahl von Kombinationsmöglichkeiten unterschiedlicher Speicherungsstrukturen erschwert das Finden einer weitgehend „optimalen“ internen Darstellung der Datenbankobjekte erheblich. Auch hier spielen Informationen über die gegen die Datenbankobjekte gerichtete Workload eine wichtige Rolle. In *Kapitel 9* wird ein Ansatz zur Bewertung alternativer interner Repräsentationen von komplexen Datenbankobjekten bezüglich ihres Einflusses auf den zur Workload-Abarbeitung notwendigen I/O-Aufwand diskutiert. Der Ansatz basiert dabei auf dem in Kapitel 6 vorgestellten Verfahren zur Gewinnung von Workload-Informationen.

In *Kapitel 10* wird eine Zusammenfassung der Ergebnisse und ein Ausblick auf weitere mögliche Arbeiten gegeben. Dabei werden u.a. auch Einbettungsmöglichkeiten in Konzepte zur Automatisierung von Datenbankadministrationsaufgaben aufgezeigt.

In den *Anhängen* sind weitere Informationen aufgeführt, auf die im Rahmen der Ausführungen der Arbeit Bezug genommen wird. Detaillierte Beschreibungen der Elemente des eInformationsschemas oder auch von Berechnungsschemata für Speicherplatzabschätzungen sowie Kosten- und Mehraufwandsschätzungen für spezielle Strukturen (bspw. indexorganisierte Tabellen oder Cluster) hätten zu einer schlechten Übersichtlichkeit des Hauptteils geführt. Weiterhin finden sich in den Anhängen Auszüge aus den Programmtexten prototypischer Implementierungen von Reorganisationsverfahren für das DBMS-Produkt Oracle und zur Umsetzung des Branch-and-Bound-Verfahrens sowie Erläuterungen zu an Beispielen durchgeführten praktischen Untersuchungen und Leistungsmessungen.

2 Datenbankreorganisation – Grundlagen und Abgrenzung

In diesem Kapitel werden grundlegende Begriffe und Konzepte des Themengebiets Datenbankreorganisation vorgestellt. Dabei wird an mehreren Stellen auch auf den Beitrag [SG79], in dem ein guter Überblick zum Thema Datenbankreorganisation gegeben wird und auf den auch andere Quellen (bspw. [MDL87]) verweisen, Bezug genommen.

Nach einem kurzen Überblick über die Architektur von Datenbanksystemen in *Abschnitt 2.1* werden in *Abschnitt 2.2* Gründe für Datenbankreorganisationen erarbeitet und klassifiziert. *Abschnitt 2.3* beinhaltet eine Beschreibung von verschiedenen Reorganisationsmaßnahmen und von Ebenen, auf denen diese Maßnahmen durchgeführt werden können. Vor der Betrachtung unterschiedlicher Reorganisationsgranulate in *Abschnitt 2.5* erfolgt eine Abgrenzung (*Abschnitt 2.4*), wie der Begriff *Datenbankreorganisation* in dieser Arbeit verwendet wird. Auch verschiedene Reorganisationsmethoden werden kurz vorgestellt (*Abschnitt 2.6*). Eine abschließende Formulierung von Anforderungen, die im Zusammenhang mit Datenbankreorganisationen gestellt werden und die Nennung wichtiger Faktoren, die Einfluss auf die Häufigkeit der Durchführung von Datenbankreorganisationen haben, erfolgt in *Abschnitt 2.7*

2.1 Überblick über die Architektur eines Datenbanksystems

Zur Orientierung und zur kurzen Einführung der in dieser Arbeit häufig verwendeten Begriffe soll zunächst auf die Architektur von Datenbanksystemen eingegangen werden. Ein *Datenbanksystem (DBS)* besteht aus einem *Datenbank-Management-System (DBMS)* und einer *Datenbank*. Vereinfachend wird im Folgenden an einigen Stellen auch nur der Begriff *System* verwendet. Innerhalb eines Datenbanksystems existieren verschiedene logische und physische Strukturen auf unterschiedlichen Architekturebenen. Ein gebräuchliches Architekturmodell [Här05], das die Abbildungen zwischen den einzelnen Architekturschichten von Datenbank-Management-Systemen beinhaltet, ist in *Abbildung 2.1* dargestellt. Die Ausführungen in dieser Arbeit beziehen sich größtenteils auf die Ebenen **L2** bis **L4**. Die Datei-Management-Ebene (**L1**) liegt i.d.R. außerhalb von Datenbank-Management-Systemen und wird daher hier nicht eingehend betrachtet. Erläuterungen zur Ebene **L5** finden sich in Abschnitt 2.4.

In den Strukturen dieser Ebenen, von denen gebräuchliche Vertreter in *Kapitel 3* vorgestellt werden, treten Degenerierungen auf, die zu Leistungsverschlechterungen führen und damit Datenbankreorganisationen erforderlich machen.

Zwei weitere Begriffe, die im Laufe dieser Arbeit häufig verwendet werden, sind *Datenbankobjekt* und *Datenobjekt*. Als Datenbankobjekte sollen logische Daten- bzw. Zugriffspfadstrukturen (bspw. Tabellen, Indexe oder Cluster) bezeichnet werden. Als *Zugriffspfad* wird eine Möglichkeit zum Zugriff auf Datenobjekte bezeichnet, die durch die interne Datenorganisation gegeben ist. Datenobjekte sind die Elemente, die die in der Datenbank gespeicherten Informationen beinhalten. Solche Datenobjekte

sind z.B. Tupel oder Einträge, die Informationen über komplexe Anwendungsobjekte darstellen. Physisch werden Datenobjekte als (Daten-)*Sätze* innerhalb von *Segmenten* gespeichert, die zur Verringerung von Leistungsverlusten bei der Abbildung auf die DB-Pufferschnittstelle in *Seiten*[1] zerlegt werden. Abweichend von den in [HR01] skizzierten flexiblen Möglichkeiten werden bei DBMS-Produkten i.d.R. jeweils eigene, getrennte Segmente für die einzelnen Datenbankobjekte (Tabellen, Indexe, Cluster usw.) angelegt.

Level of abstraction		Objects	Auxiliary mapping data
L5	Nonprocedural or algebraic access	Tables, views, tuples	Logical schema description
L4	Record-oriented, navigational access	Records, sets, hierarchies, networks	Logical and physical schema description
L3	Record and access path management	Physical Records, access paths	Free space tables, DB-key translation tables
L2	Propagation control	Segments, pages	DB buffer, pages tables
L1	File management	Files, blocks	Directories, VTOCs, etc.

Abbildung 2.1: Abbildungshierarchie innerhalb von Datenbank-Management-Systemen (nach [Här05])

Im Rahmen des Datenbankbetriebs kommt es zu I/O-Operationen, wenn Daten von den Datenträgern gelesen oder auf diese geschrieben werden. I/O-Operationen erfolgen nicht byteweise, sondern in Blöcken. Zur Beschleunigung von Datenzugriffen werden Blöcke von Datenbank-Management-Systemen im Arbeitsspeicher gepuffert. Datenzugriffe erfolgen über die im Puffer zwischengespeicherten Blöcke. Der bei solchen Zugriffen anfallende I/O-Aufwand wird als *logischer I/O-Aufwand* bezeichnet. Befindet sich ein benötigter Block nicht im Puffer oder soll ein im Puffer geänderter Datenblock geschrieben werden, so ist jeweils ein Zugriff auf den Datenträger nötig. Der dabei anfallende Aufwand wird als *physischer I/O-Aufwand* bezeichnet. Wegen der erheblichen Zeitunterschiede zwischen im Arbeitsspeicher auszuführenden Zugriffen und Zugriffen auf Datenträger, hat besonders die physische (die „eigentliche“) I/O großen Einfluss auf die Antwortzeit. Nach gängigen Annahmen wird bei Aufwandsschätzungen der logische I/O-Aufwand oft vernachlässigt und nur der um mehrere Zehnerpotenzen stärker zu Buche schlagende physische I/O-Aufwand berücksichtigt. Ziel bei der Beschleunigung von Datenzugriffen ist es, die Zahl der physischen I/O-Operationen im Vergleich zu den bei der Operationsausführung anfallenden logischen I/O-Operationen möglichst weit zu reduzieren. Ein Maß für dieses Verhältnis zwischen logischen und physischen I/O-Operationen ist der *Pufferungsgrad.* In dieser Arbeit wird die Wahrscheinlichkeit, dass sich ein benötigter Block im Arbeitsspeicher befindet, als Pufferungsgrad (Buffer Hit Ratio) bezeichnet. In realen Systemumgebungen hängt der Pufferungsgrad stark von der

[1] In der Begriffswelt von DBMS-Produkten, an die sich auch diese Arbeit anlehnt, wird oftmals der Begriff *Block* synonym zu *Seite* verwendet.

Anwendungscharakteristik ab. In einigen Szenarien (SAP-OLTP-Betrieb etwa) sind Pufferungsgrade von 95% und mehr zu finden.

2.2 Gründe für Datenbankreorganisationen

Eine Datenbank bildet einen bestimmten Ausschnitt aus der realen Umwelt (*Miniwelt*) ab. Diese Miniwelt unterliegt durch äußere und innere Einflüsse mehr oder weniger starken Veränderungen, z.B. durch die Erweiterung der zur Aufgabenerledigung notwendigen Informationen, durch Veränderungen im Datenaufkommen oder veränderte Nutzung der Daten (Workload). Diese Veränderungen müssen auch in der Datenbank entsprechend nachvollzogen werden, damit diese die Miniwelt noch korrekt abbildet. Hierzu sind teils *Veränderungen an der logischen Struktur* der Datenbank (Datenbankschema) nötig. Änderungen am Datenaufkommen, wie eine starke Erhöhung der zu speichernden Datenmenge oder eine veränderte Nutzung, bspw. wenn produktionsbezogene Daten auch für Auskunftssysteme verwendet werden sollen, erfordern eine *Anpassung der internen Strukturen* zur Datenspeicherung bzw. eine *Neuabstimmung von die Datenspeicherung beeinflussenden Parametern*. Aber nicht nur Veränderungen des jeweils abgebildeten Umweltausschnitts erfordern eine derartige Wartung von Datenbanken, auch die bloße Benutzung der Datenbestände führt durch Änderungsoperationen dazu, dass interne Speicherungsstrukturen nicht mehr optimal organisiert sind. Die notwendige Pflege der Strukturen kann aus Performance-Gründen i.d.R. nicht oder nicht vollständig bzw. nicht sofort ausgeführt werden. Eine zyklische *Bereinigung der Strukturen* wird nötig. All diese Wartungsarbeiten werden mit dem Begriff *Datenbankreorganisation* in Verbindung gebracht.

Von Sockut und Goldberg werden in [SG79] Szenarien aufgeführt, die Datenbankreorganisationen nach sich ziehen können. Genannt werden:

- Änderungen in der Definition der zu speichernden Daten, bspw. weil zukünftig zu einer Person beliebig viele Adressen (statt bisher nur eine) gespeichert werden sollen
- Erweiterung der zu speichernden Informationen, weil neue Datenfelder hinzugefügt werden
- Erweiterung der Informationsmenge durch das Zusammenführen bisher in getrennten Datenbanken abgebildeter Bereiche
- Entfernen von Informationen, die z.B. aus datenschutzrechtlichen Gründen nicht mehr gespeichert werden dürfen
- Erweiterungen der Nutzungsform und die damit verbundene notwendige Schaffung neuer Zugriffspfade
- Starker Anstieg der Datenmenge
- Änderung der Häufigkeit von Zugriffen auf bestimmte Daten im Laufe der Zeit, weil bspw. auf aktuelle Daten häufig, auf Daten aus der Vergangenheit seltener zugegriffen werden muss

- Verschlechterungen der Systemleistung im normalen Datenbankbetrieb durch Degenerierungen in internen Speicherungsstrukturen, die durch Änderungsoperationen entstehen

Die ersten vier Szenarien führen zu Änderungen am Datenbankschema, wohingegen die letzten vier Szenarien Maßnahmen zur Verbesserung der Systemleistung bzw. eine Bereinigung oder den Neuaufbau interner Speicherungsstrukturen erforderlich machen. Allerdings ziehen auch Änderungen am logischen Datenbankschema bei ihrer Umsetzung i.d.R. eine Umstellung der internen Speicherungsstrukturen sowie die Schaffung neuer Zugriffspfade nach sich. Aus diesem Grund lässt sich die Problematik der Schemaevolution nicht vollständig von der Reorganisationsthematik im Sinne dieser Arbeit lösen.

Die sich aus den genannten Szenarien ergebenden Gründe zur Durchführung einer Datenbankreorganisation, die in *Abbildung 2.2* (vgl. auch [DK00]) dargestellt sind, können grob in *zwingende Gründe* und solche, die Anpassungen *empfehlenswert* erscheinen lassen [MDL87], eingeteilt werden.

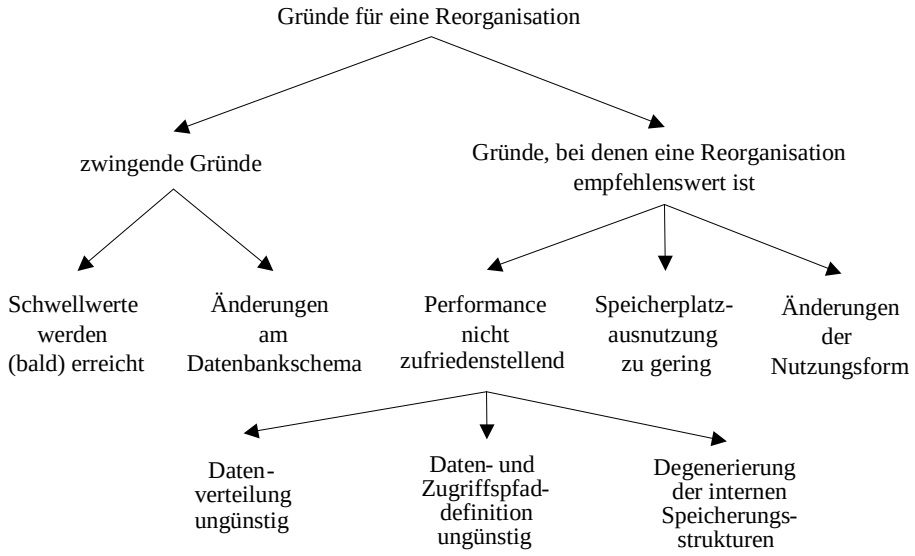


Abbildung 2.2: Gründe für eine Datenbankreorganisation

Bei **zwingenden Gründen** steht die *Aufrechterhaltung des Betriebs des Datenbanksystems* bzw. der Anwendung im Vordergrund. Beispiele sind hier das (baldige) Erreichen von Grenzen (Schwellwerte), die vom Datenbank-Management-System oder vom Betriebssystem vorgegeben werden, wie maximale Dateigrößen oder die maximale Anzahl verwaltbarer Speichereinheiten (z.B. Extents je Tabelle). Aber nicht nur Gründe, die in direktem Zusammenhang mit dem DBMS oder der Systemplattform stehen, sondern auch notwendige Veränderungen am Datenbankschema können eine Reorganisation der internen Strukturen erzwingen.

Empfehlenswert sind Datenbankreorganisationen oft aus *Performance-Gründen*. Im Rahmen des physischen Datenbankentwurfs wird ein Konzept (eine Strategie) zur Abbildung logischer Datenbankobjekte auf physische Speicherungsstrukturen entwickelt. Dabei muss auch berücksichtigt werden, dass das interne (physische)

Datenbankschema die Abarbeitung der gegen die Datenbank gerichteten Workload gut unterstützt. Zur Lastverteilung und zur Ausnutzung von Möglichkeiten zur Parallelisierung der Verarbeitung von Datenbankanfragen werden große Tabellen oft horizontal partitioniert und damit gezielt auf mehrere Datenträger verteilt. Basis für die Partitionierung ist ein Verteilungsschema [Now01b]. Wenn aber (was in der Praxis durchaus häufiger vorkommen kann) die Verteilung der Schlüsselwerte, auf denen das Partitionierungsschema beruht, nicht genau vorhersehbar ist oder sich im Laufe der Zeit verändert, kann es durch „Schieflagen" zu Performance-Einbußen kommen, die durch eine Umverteilung der Daten (durch die Anpassung des Partitionierungsschemas) beseitigt werden können.

Im Unterschied zu relationalen Datenbank-Management-Systemen, bei denen teils nur wenig Einfluss auf die interne Speicherung von Tabellen und Indexen genommen werden kann, existier(t)en bei vorrelationalen Systemen hier sehr viele Freiheitsgrade. Dies bietet vielfältige Möglichkeiten zur Schaffung von Zugriffspfaden, die Einfluss auf die Systemleistung haben. Auch im Zusammenhang mit der Weiterentwicklung von objektrelationalen Datenbank-Management-Systemen können den Administratoren mehr Möglichkeiten zur Beeinflussung der internen Darstellung eingeräumt werden [Ska06], damit die Abarbeitung der gegen die Datenbank gerichteten Workload möglichst gut unterstützt wird. Durch Indexierung und die damit verbundene Schaffung weiterer Zugriffspfade, bspw. über Sekundärschlüssel, können Datenzugriffe u.U. erheblich beschleunigt werden. Allerdings verursachen Indexe oder eine intern geordnete Datenspeicherung (Vorsortierung) Pflegeaufwand bei Änderungsoperationen. Kommt es zu Änderungen in der Workload-Zusammensetzung, so ist eine Anpassung der *Zugriffspfaddefinitionen* notwendig.

Degenerierungen der zur Datenspeicherung verwendeten internen Strukturen verursachen ebenfalls Performance-Einbußen. Der Aufwand für Datenzugriffe über Indexe steigt bspw. an, wenn eine hohe Anzahl Datensätze vorhanden ist, die nach verlängernden Änderungsoperationen in andere Datenblöcke verschoben werden mussten und deshalb bei Zugriffen über Indexe nur über einen zusätzlichen (Auslagerungs-)Zeiger auf den neuen Speicherort zu erreichen sind. Der Aufwand für Sortieroperationen kann deutlich verringert werden, wenn Daten physisch möglichst gut nach dem entsprechenden Sortierkriterium geordnet abgespeichert werden. Aus Performance-Gründen wird die Einhaltung solcher interner Sortierungen bei Einfüge- und Änderungsoperationen aber i.d.R. nicht erzwungen. Damit geht die interne Sortierreihenfolge sukzessive verloren und der Aufwand für Sortierläufe steigt an.

Fragmentierungen [Dor99a] und durch Löschoperationen entstehender eingestreuter Freiplatz[2] sowie dünn besetzte Knoten von Indexbäumen führen oft neben einer Leistungsverschlechterung noch zu einer *verschlechterten Auslastung des* zur Datenspeicherung belegten *Sekundärspeichers*.

[2] Eine große Zahl an Freiplatzfragmenten entsteht z.B. im Zusammenhang mit der Archivierung von Daten [Ste02], weil die archivierten Daten, anders als bei Backup-Operationen, typischerweise aus den operativen Daten gelöscht werden.

Änderungen bzw. Erweiterungen der Nutzungsform des Datenbestands, z.B. durch die Einführung von Zugriffsmöglichkeiten für eine große Zahl von Nutzern übers Internet, können ebenfalls eine Neuorganisation erfordern. Aber auch in Data-Warehouse-Systemen, zur Unterstützung von Entscheidungsfindungsprozessen mit überwiegend langen und aufwendigen sequenziellen Leseprozessen, sind andere Organisationsformen des Datenbestands sinnvoll, als in einer operativen Datenverarbeitung mit meist kurzen Transaktionen, bei denen häufig auch Änderungen am Datenbestand vorgenommen werden [MHR96, MRB99]. Je nach Art und Umfang der Änderungen der Nutzungsform kann der Reorganisationsbedarf schon nahezu zwingend werden.

Mit einer Reorganisation können im Wesentlichen folgende Ziele angestrebt werden:

- Aufrechterhaltung des Datenbankbetriebs
- Verbesserung der Systemleistung
- Wiedergewinnung von Speicher

2.3 Reorganisationsmaßnahmen und Reorganisationsebenen

Reorganisationsmaßnahmen unterschiedlicher Art können auf den verschiedenen Architekturebenen von Datenbanksystemen durchgeführt werden. In [SG79] werden anhand des Data Independent Accessing Model (DIAM II nach [SA76]) für vorrelationale und relationale Systeme Beispiele für auf den einzelnen Architekturebenen anfallende Reorganisationsmaßnahmen aufgeführt. *Abbildung 2.3* zeigt einen Überblick über die DIAM II-Architektur und die Zuordnung der Ebenen zur ANSI-SPARC-Datenbankarchitektur [TK78]. Rechtecke kennzeichnen die verschiedenen Strukturelemente der einzelnen Ebenen. Die Abbildungsmechanismen werden in Ovale eingeschlossen.

Die oberste Ebene bildet das *End-User Level*. Dies entspricht der Ebene der externen Schemata im ANSI-SPARC-Architekturvorschlag. Hier werden die Daten so aufbereitet, wie es der Anwender oder ein auf der Datenbank arbeitendes Anwendungsprogramm aus der eigenen Sicht erwartet. Oftmals wird nur ein Ausschnitt aus dem darunter liegenden logischen Gesamtschema der Datenbank präsentiert. Änderungen können bei der Nutzung von View-Konzepten teilweise durchgeführt werden, ohne dass die darunter liegenden Ebenen berührt werden. Benötigt aber bspw. eine Anwendung zukünftig Daten, die bisher noch nicht in der Datenbank gespeichert wurden, so muss das logische Gesamtschema erweitert werden.

Das sog. *Infological Level* entspricht dem konzeptuellen Schema in der ANSI-SPARC-Datenbankarchitektur (logisches Gesamtschema der Datenbank). Reorganisationsmaßnahmen auf dieser Ebene verändern das Datenbankmodell der Informationsstrukturen des abgebildeten Umweltausschnitts. Zu den Reorganisationsmaßen zählen hier etwa das Hinzufügen oder Entfernen von Attributen, Änderungen der Wertebereiche von Attributen, das vertikale Aufspalten oder auch das Zusammenführen von Tabellen bzw. das Aufspalten oder

Zusammenführen komplexer Objekte im Sinne von objektrelationalen Datenbanksystemen, das Hinzufügen neuer Tabellen/Objekte bzw. das Entfernen von Tabellen/Objekten. Umstrukturierungen komplexer Objekte, das Zusammenführen oder Aufteilen von Tabellen sowie Änderungen der verwendeten Attribute sind oftmals nur dann möglich, wenn das zur physischen Datenspeicherung verwendete Satzformat ebenfalls geändert wird. Reorganisationen auf der Schemaebene ziehen oft zwangsläufig Reorganisationsmaßnahmen auf den tiefer liegenden internen Ebenen nach sich. Reorganisationen auf der Schemaebene werden notwendig, wenn sich die abzubildenden Informationsstrukturen des betrachteten Umweltausschnitts durch Veränderungen der Anforderungen, Aufgaben usw. ändern. Aber auch Änderungen der Nutzungsform können Anpassungen auf der Schemaebene, wie z.B. gezielte Denormalisierungen relationaler Schemata zur Performance-Verbesserung, empfehlenswert erscheinen lassen.

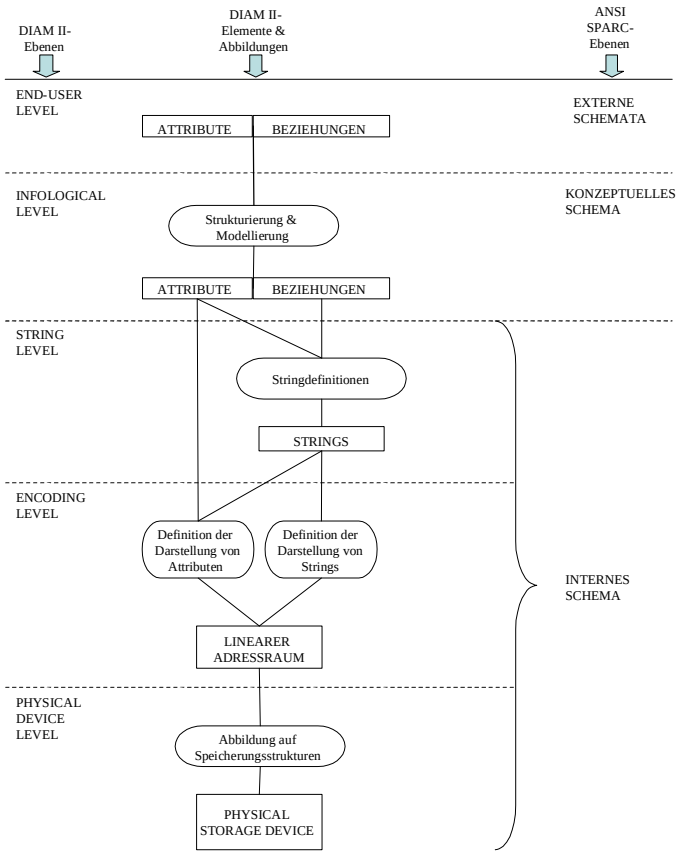


Abbildung 2.3: Überblick über DIAM II (entnommen aus [SG79])

Auf dem *String Level* werden Zugriffspfade definiert, die meist Beziehungen zwischen Attributen ausdrücken. Bei vorrelationalen Systemen wird dies bspw. über Referenzen auf der internen Ebene realisiert. Bei rein relationalen Systemen werden elementare Beziehungen zwischen Attributen durch die Bildung von Tabellen (Relationen) ausgedrückt. Eine bestimmte Wertekombination einer Tabelle beschreibt ein Datenobjekt (Entity). Beziehungen zwischen Entities werden durch korrespondierende Attributwerte in verschiedenen Tabellen (Schlüssel-

Fremdschlüssel) – also noch auf der logischen Ebene – abgebildet. Änderungen der Beziehungen einzelner Entities untereinander werden durch Werteänderungen in den entsprechenden Attributen realisiert. Ein direkter Reorganisationsbedarf lässt sich daraus zunächst nicht ableiten.

Änderungen der Beziehungen zwischen verschiedenen Entity-Mengen, die meist unterschiedlichen Tabellen zugeordnet sind, wie das Hinzufügen oder Entfernen von Informationen über Beziehungen oder Änderungen des Beziehungstyps (1:1, 1:N, M:N), lassen sich mit Änderungen des Datenbankschemas (Restrukturierungen) abbilden. Eine Beschleunigung von Datenzugriffen kann durch das Hinzufügen von Indexen (Schaffung zusätzlicher Zugriffspfade) erreicht werden. Das Entfernen von Indexen verringert den Aufwand bei Änderungsoperationen. Bei vorrelationalen Systemen werden Beziehungen durch die Bildung interner Satzstrukturen und Verweise (Links) auf interner Ebene ausgedrückt. Mit den Erweiterungen von relationalen Datenbanksystemen um Konzepte der Objektorientierung zu objektrelationalen Datenbanksystemen wurden ebenfalls Möglichkeiten eingeführt, Beziehungen über interne Verweise (Referenzen - ganz oder teilweise auf physischer Ebene) auszudrücken. Ändert sich jetzt der Speicherort eines Datenobjekts (z.B. durch Reorganisationsmaßnahmen), auf das von einem anderen Objekt aus verwiesen wird, so muss, je nach Realisierungsform, auch der Wert der entsprechenden Referenz aktualisiert werden. Änderungen des Beziehungstyps bedingen Änderungen des internen Satzformats.

Weitere Freiheitsgrade, die die interne Speicherung komplexer Datenobjekte beeinflussen, sollen (nicht nur) in objektrelationalen Datenbanksystemen geschaffen werden. So kann es workload-abhängig sinnvoll sein, alle Daten eines komplexen Objekts jeweils dicht zu speichern (objektbezogen zu clustern) oder aber auch Teile verschiedener komplexer Objekte objektübergreifend nach dem Typ der Subobjekte zu clustern. Das erfordert die Möglichkeit, Daten von Subobjekten in das interne Satzformat der übergeordneten Objekte einbetten oder aber auch auslagern zu können. Änderungen auf dem String Level ziehen oft auch Reorganisationen auf dem tiefer liegenden Physical Device Level nach sich.

Auf dem *Encoding Level* wird die Darstellung von Daten und Verweisen bestimmt. Beispiele für Änderungen auf dieser Ebene sind:

- Längenänderungen von Attributen (bspw. Zeichenketten),
- Änderungen an der internen Darstellung von Zahlen oder Zeitwerten (bspw. durch Änderungen der Genauigkeit),
- Änderungen der Darstellung von Datumsangaben (bspw. von Monatsnamen hin zu deren Nummer und umgekehrt) oder
- Änderungen von logischen in physische Verweise und umgekehrt.

Änderungen auf dem Encoding Level erfordern Änderungen am internen physischen Satzformat und ziehen Reorganisationen auf dem Physical Device Level nach sich.

Auf dem *Physical Device Level* wird nach DIAM II die physische Darstellung von Daten und Indexen auf den Sekundärspeichermedien festgelegt. Neben der Verteilung

von Daten und Indexen auf die zur Verfügung stehenden Datenträger können hier auch die Speicherungsform von Indexen, Speicherparameter (z.B. Füllungsgrade von Datenblöcken), die Speicherungsform von Datensatzmengen (bspw. als Heap oder Liste) oder interne Satzformate geändert werden. Neben diesen Änderungen, die Reorganisationsmaßnahmen erfordern, sind auf dieser Ebene Wartungsmaßnahmen zur Beseitigung von Degenerierungen in den physischen Speicherungsstrukturen angesiedelt, die ebenfalls zu den Reorganisationsmaßnahmen zählen.

Die Betrachtungen in dieser Arbeit beziehen sich hauptsächlich auf die derzeit verbreiteten relationalen und objektrelationalen Datenbanksysteme. Diese zeichnen sich gegenüber den vorrelationalen Systemen durch eine wesentlich größere Entkopplung der logischen Konzepte von den physischen Strukturen aus. Daher erscheint die in *Abbildung 2.4* dargestellte vereinfachte Struktur mit drei Ebenen, die sich auch an den unterschiedlichen Arten von Reorganisationsmaßnahmen anlehnt, ausreichend. Reorganisationsmaßnahmen wirken sich dabei auf die jeweils tiefer liegenden Ebenen aus, was auch durch die Pfeile angedeutet wird.

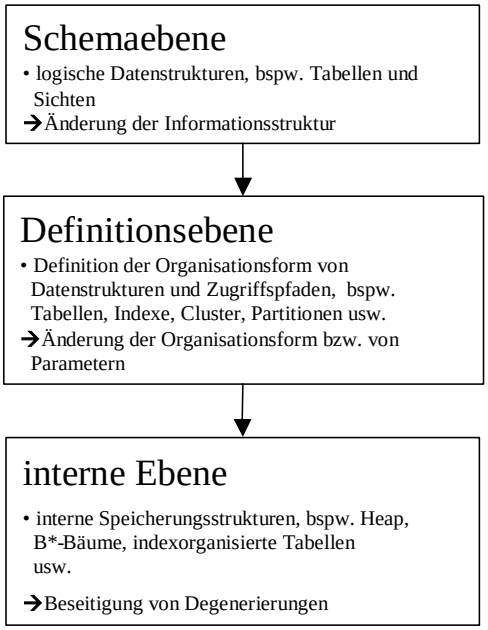


Abbildung 2.4: Reorganisationsebenen

Die oberste Ebene bildet dabei das logische Datenbankschema. Diese Ebene beinhaltet Infological Level und End-User Level aus DIAM II. Umstellungen auf der *Schemaebene* sind i.d.R. dann nötig, wenn sich die in der Datenbank abgebildete Miniwelt ändert.

Die *Definitionsebene* enthält die Beschreibungen zur Abbildung der logischen Informationsstrukturen auf die internen Speicherungsstrukturen. Diese Ebene umfasst String Level, Encoding Level und Teile des Physical Device Level. Bei Reorganisationsmaßnahmen auf dieser Ebene werden statische Eigenschaften von Datenbankobjekten dauerhaft geändert. Veränderungen auf der Definitionsebene werden oftmals vorgenommen, um durch

- die gezielte Umverteilung von Daten oder
- die Verwendung und Kombination neuer Speicherungsstrukturen und
- die damit verbundene Bereitstellung von zusätzlichen oder für die Anwendung besser geeigneten Zugriffspfaden

eine Verbesserung des Leistungsverhaltens, die Senkung der Speicherkosten oder eine Vergrößerung der Intervalle zwischen notwendigen Wartungsmaßnahmen an den Speicherungsstrukturen zu erreichen.

Auf der *internen Ebene* steht die Beseitigung von Degenerierungen der physischen Speicherungsstrukturen von Daten und Indexen im Vordergrund. Sie bildet die unterste Ebene und umfasst große Teile des Physical Device Level. Wegen der unvollständigen Pflege der physischen Speicherungsstrukturen durch die Datenbank-Management-Systeme im laufenden Betrieb, müssen die auf dieser Ebene anfallenden Reorganisationsmaßnahmen zyklisch zur Wartung der Strukturen ausgeführt werden. Eine detailliertere Betrachtung verbreiteter Speicherungsstrukturen, häufig auftretender Degenerierungen sowie von Maßnahmen zur Verzögerung der Entwicklung von Degenerierungen und zu deren Beseitigung erfolgt in *Kapitel 3*.

2.4 Abgrenzung

Nach [TL91] umfassen Datenbankreorganisationen „alle Maßnahmen zur Wiederherstellung der Bedingungen für eine effiziente Datenmanipulation – vergleichbar mit dem Leistungsverhalten zu Nutzungsbeginn – ". Diese Definition ist recht allgemein und schließt

- Veränderungen des logischen Schemas einer Datenbank,
- Änderungen der internen Struktur einer Datenbank,
- die Bereinigung bzw. den Neuaufbau interner Speicherungsstrukturen und
- eine Neuabstimmung von Systemparametern, die die Performance eines DBMS beeinflussen (z.B. Größe des Puffer-Pools, Parameter zur Steuerung vorausschauenden Lesens usw.),

mit ein. Die im letzten Punkt aufgeführten Maßnahmen zur Beeinflussung des Leistungsverhaltens von DBMS haben i.d.R. keine Auswirkungen auf die logische und physische Datenorganisation und werden oft auch unter dem Begriff *Datenbank-Tunnig* zusammengefasst [EN02].

In verschiedenen Beiträgen [NF76, SG79, MDL87, SBC97] wird der Begriff Datenbankreorganisation etwas enger gefasst und beinhaltet die Änderung der logischen (*Restrukturierung*) bzw. internen (*Reformatierung*) Organisation einer Datenbank.

Teilweise (z.B. [YDT76, Tue78, Ora04]) wird der Begriff Datenbankreorganisation lediglich für Maßnahmen auf der internen Ebene verwendet.

In [SG79] werden noch weitere Beispiele für Begriffsverwendungen für die unterschiedlichen Arten von Reorganisationsmaßnahmen genannt. So wird in

[BCS75] der Begriff *Restructuring* bei Schemaänderungen, bei Änderungen der Datenbeschreibung mit der Data Storage Description Language (DSDL) der Begriff *Strategy Reorganization* und bei Maßnahmen unterhalb der DSDL-Ebene der Begriff *Physical Placement Reorganization* verwendet. Es kann zwischen einmaligen Operationen, bei denen dauerhafte Änderungen an Strukturdefinitionen vorgenommen werden und zu denen Restructuring- und Strategy-Reorganization-Maßnahmen zählen und Wartungsoperationen, die zyklisch ausschließlich auf der physischen Ebene zur Beseitigung von Degenerierungen in den Speicherungsstrukturen vorgenommen werden, unterschieden werden. Im Zusammenhang mit konkreten DBMS-Produkten wird zwischen Redefinitions- und Reorganisationsmaßnahmen unterschieden [Ora04]. Bei *Redefinitionsmaßnahmen* werden Daten- und Zugriffspfaddefinitionen geändert. Als *Reorganisationsmaßnahmen* werden Wartungsmaßnahmen zur Beseitigung von Degenerierungen bezeichnet (bei denen also die genannten Definitionen unverändert bleiben).

Anpassungen des logischen Datenbankschemas werden i.d.R. durch veränderte Anforderungen der Anwendung (bspw. wenn zukünftig zusätzliche Informationen zu berücksichtigen und zu speichern sind) unabhängig vom zu Grunde liegenden Datenbanksystem vorgenommen. Hier geht es vorrangig darum, den Informationsbedarf der Anwendung zu befriedigen. Performance-Aspekte spielen eine eher untergeordnete Rolle. Wichtig ist die Wahrung der *logischen* Konsistenz der Datenbank. Weitere Probleme ergeben sich hier bspw. auch auf dem Gebiet der Datenarchivierung, weil sich zu archivierende Daten auf das jeweilige Datenbankschema beziehen und einmal archivierte Daten aber nicht mehr verändert und an neue Datenbankschemata angepasst werden dürfen [Sch99]. Hier sind Versionierungskonzepte notwendig. Probleme der Wahrung der logischen Konsistenz, evtl. auch über lange Zeiträume hinweg, treten vom konkreten Datenbanksystem, auf dem eine Anwendung aufbaut, unabhängig auf. Sie können unabhängig vom DBS gelöst werden. Für Restrukturierungen der logischen Datenorganisation wird oft auch der Begriff *Schemaevolution* verwendet. Das Thema wird mittlerweile auch in der Literatur meist getrennt vom Themengebiet Datenbankreorganisation behandelt. Die Ursachen für Schemaumstellungen liegen i.d.R. außerhalb des Verantwortungsbereichs des Administrators und des verwendeten DBMS. Daher werden Reorganisationen auf der Schemaebene selbst in dieser Arbeit nicht eingehender betrachtet.

Unter den Begriff *Datenbankreorganisation im Sinne dieser Arbeit* fallen Maßnahmen, die

- in engem Bezug zum Datenbanksystem stehen,
- den logischen Informationsgehalt der Datenbank nicht verändern und
- üblicherweise zur Verringerung der Speicher- und/oder Verarbeitungskosten

beim Betrieb vorhandener Anwendungen dienen. Damit sind, wie auch in Beiträgen, die das Thema aus Sicht der Betreiber von Datenbanksystemen betrachten (z.B. [Nob95, Sha95, Sch00]), Maßnahmen auf der Definitionsebene und auf der internen

Ebene eingeschlossen. Der Schwerpunkt liegt aber deutlich auf den Maßnahmen auf der internen Ebene.

2.5 Reorganisationsgranulate

Die Granulate, die für Datenbankreorganisationen denkbar sind, lassen sich grob in zwei Gruppen einteilen [DK98]. Einerseits existieren logische Granulate, wie Datenbanken, Tabellen oder Mengen von Tabellen. Andererseits lassen sich physische Granulate bilden, die zum Teil vom verwendeten DBMS abhängig sind. Zu den physischen Granulaten zählen Partitionen, Indexe und die vom Datenbank-Management-System verwendeten Speichereinheiten.

Das gröbste Granulat stellt eine *gesamte Datenbank* dar. Durch deren Reorganisation werden alle darin enthaltenen Strukturen wieder bereinigt. Bei sehr großen Datenbanken mit einem Speichervolumen von mehreren hundert Gigabyte bis einigen Terabyte und einer vier- bis fünfstelligen Anzahl von Tabellen und Indexen ist dieses Granulat allerdings schon aus Aufwandsgründen[3] nahezu ausgeschlossen. Die Reorganisation einer gesamten Datenbank ist meist sehr arbeits- und zeitintensiv, und ein globales Optimum im Ergebnis wird nur selten erreicht (und gilt dann vielleicht nur für kurze Zeit). Weiterhin bleibt bei der Verwendung dieses Granulats das unterschiedliche Degenerierungsverhalten einzelner Datenbankteile unberücksichtigt. Bereiche, die Stammdaten enthalten, die normalerweise einmal erfasst, selten geändert und (wenn überhaupt) erst nach längeren Zeiträumen gelöscht werden, degenerieren mit Sicherheit langsamer als Bereiche, die Bewegungsdaten enthalten. Es werden Teile der Datenbank reorganisiert, für die dies nicht oder zumindest noch nicht nötig gewesen wäre.

Als weiteres Granulat für eine Datenbankreorganisation sind *einzelne Tabellen* denkbar. Die Verwendung dieses Granulats erlaubt die gezielte Reorganisation degenerierter Bereiche der Datenbank. Da bei Reorganisationen i.d.R. Daten bewegt werden, müssen meist auch Indexe, die auf eine zu reorganisierende Tabelle verweisen, reorganisiert (typischerweise neu aufgebaut) werden. Einige DBMS (z.B. Oracle, Informix) erlauben aus Performance-Gründen die Ermittlung und „externe“ Verwendung physischer Referenzen auf die Speicherorte von Tupeln. Damit können solche Verweise auch als Fremdschlüssel verwendet werden. Wird jetzt eine Tabelle reorganisiert, so ändern sich typischerweise die physischen Speicherorte der einzelnen Tupel und damit auch die Werte der physischen Verweise. Eine Aktualisierung der Inhalte von Fremdschlüsselspalten anderer Tabellen, die physische Verweise auf die Tupel der reorganisierten Tabelle enthalten, wird damit notwendig. In vielen Fällen ist auch noch die Aktualisierung evtl. über den Fremdschlüsseln existierender Indexe erforderlich. Durch derartige Abhängigkeiten zwischen Datenbankobjekten wird das Reorganisationsgranulat u.U. zwangsläufig erweitert.

---

[3] Neben dem Systemaufwand, der zur eigentlichen Reorganisation anfällt, ist der manuelle Aufwand zur Veranlassung der einzelnen Teilreorganisationen nicht zu unterschätzen, wenn dies nicht von geeigneten Werkzeugen übernommen wird.

Typische Reorganisationsgranulate stellen *Mengen von Tabellen* dar. Nach welchen Kriterien diese Menge zusammengestellt wird, ist unterschiedlich. Hier können, besonders auch bei objektrelationalen Systemen, logische Abhängigkeiten zwischen Tabellen eine Rolle spielen. Ein anderes Kriterium kann das Speicherkonzept des verwendeten DBMS sein. So können bspw. alle innerhalb einer auf DBMS-Ebene ansprechbaren Verwaltungseinheit (z.B. einem Table Space) gespeicherten Tabellen und Indexe zusammengefasst werden, um eine Reorganisation der gesamten Verwaltungseinheit durchzuführen und dort ein lokales Optimum zu erreichen.

Auch *Indexe* können unabhängig von den ihnen zu Grunde liegenden Tabellen reorganisiert werden. Dies ist nach längerer Zeit u.U. notwendig, um bei B-Baum-Indexen wieder eine akzeptable und gleichmäßige Auslastung der einzelnen Indexknoten zu erreichen oder um zwar als gelöscht gekennzeichnete, aber nicht physisch gelöschte Einträge zu entfernen und dadurch „unnötigen Ballast abzuwerfen“. Die DBMS-Produkte weichen hier meist in der Implementierung von der „reinen Lehre“ der B-Baum-Realisierung ab. Üblicherweise wird eine manuell veranlasste Indexreorganisation durchgeführt, indem der Index gelöscht und anschließend neu aufgebaut wird.

Aus Performance-Gründen, aber auch aus Gründen der Verbesserung der Administrierbarkeit von sehr großen Tabellen und Indexen, wurden in den letzten Jahren Möglichkeiten zur horizontalen Partitionierung von Tabellen und Indexen in mehrere DBMS-Produkte integriert. Damit ist es möglich, gezielt einzelne Teile von Tabellen oder Indexen, sog. *Partitionen*, zu reorganisieren. Eventuell muss eine solche Reorganisation bei den derzeit verfügbaren Datenbank-Management-Systemen über Umwege (Herauslösen der entsprechenden Partition aus der Relation, Reorganisation und abschließendes Wiedereingliedern des reorganisierten Teils) erfolgen.

2.6 Reorganisationsformen und Reorganisationsmethoden

Datenbankreorganisationen können, wie in *Abbildung 2.5* dargestellt, nach der Art der Veranlassung und nach den zur Reorganisationsdurchführung verwendeten Komponenten unterschieden werden. Zunächst kann grob zwischen explizit veranlasster oder impliziter Reorganisation unterschieden werden [Dor99a]. Manuelle (*explizit veranlasste*) Datenbankreorganisationen können noch in systembasierte und systemintegrierte Reorganisationsformen unterteilt werden. Bei den *impliziten* Reorganisationen kann zwischen automatischer Reorganisation und Reorganisationsfreiheit durch ständige, also zeitnahe Pflege (Reorganisieren) der Speicherungsstrukturen unterschieden werden.

Als *systembasierte Reorganisation* sollen Reorganisationsformen bezeichnet werden, die in der Regel unter Verwendung von Hilfsprogrammen (Utilities) arbeiten und die aufgesetzt auf einem DBMS realisiert sind (z.B. Dienstprogramme zum Export bzw. Import von Daten).

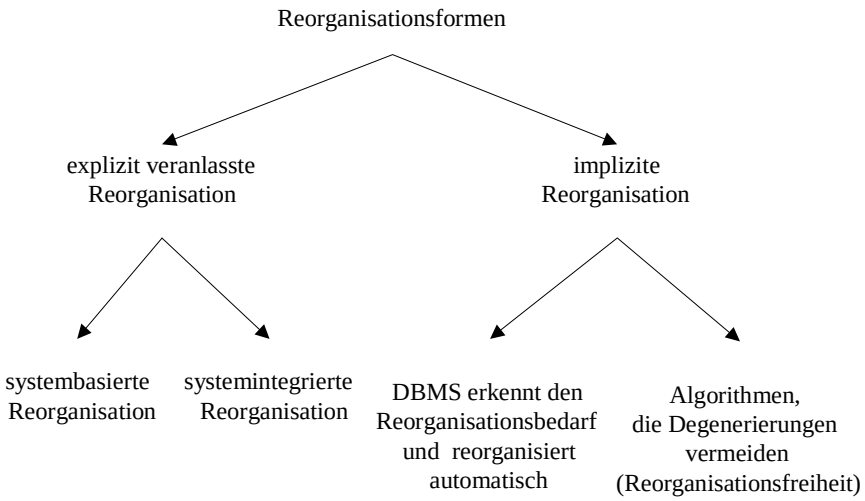


Abbildung 2.5: Einteilung der Reorganisationsformen für Datenbanken

Bei *systemintegrierten Reorganisationsformen* bietet das DBMS (z.B. auf erweiterter SQL-Ebene) Möglichkeiten an, einen Datenbestand oder die entsprechenden Zugriffspfade ohne Aus- und Einlagern der Daten, sozusagen aus Anwendersicht in „einem Schritt“, zu reorganisieren. Die Datenbankreorganisation muss aber vom Benutzer bzw. dem DBA veranlasst werden.

Explizit veranlasste Reorganisationen können für den Durchführenden u.U. recht aufwendig sein, wenn auch Abhängigkeiten zwischen Datenbankobjekten (z.B. Indexe, physische Verweise auf Tupel anderer Tabellen) berücksichtigt werden und abhängige Objekte mit reorganisiert werden müssen. Unterstützung durch die entsprechenden Werkzeuge ist hier dringend angeraten.

Als implizit soll eine Reorganisationsform bezeichnet werden, bei der das DBMS selbst oder ein spezieller Reorganisationsprozess *Reorganisationserfordernisse erkennt und automatisch eine Reorganisation ausführt*. Die dadurch verursachte Systemlast, welche u.U. die Performance auch zu Zeiten hoher Belastung durch die Benutzer negativ beeinflusst, ist in der Praxis allerdings oft ungern gesehen. Moderne Datenbank-Management-Werkzeuge erlauben es daher, Kriterien festzulegen, die zu einer Reorganisation führen sollen und die dann überwacht werden sowie „Reorganisationspläne“ zu definieren, die dann automatisch (z.B. in lastarmen Zeiten) abgearbeitet werden.

Als eine weitere Form einer impliziten Datenbankreorganisation kann die *Nutzung von Algorithmen* angesehen werden, die während Änderungsoperationen auf den Speicherungsstrukturen dafür sorgen, dass diese nicht degenerieren. In diesem Zusammenhang ist teils auch von *Reorganisationsfreiheit* die Rede [Nuß97, SAG97].

Bei der Reorganisation von Datenbankobjekten können, unabhängig von der Veranlassung, unterschiedliche Methoden (*Abbildung 2.6*) angewendet werden. *Zyklisch* (**I**) werden Reorganisationsmethoden nach einer bestimmten Nutzungsphase angewendet, um während dieser Nutzungsphase entstandene Degenerierungen zu

beseitigen. Solche Reorganisationen sind typischerweise nach einer kurzen Zeitspanne beendet.

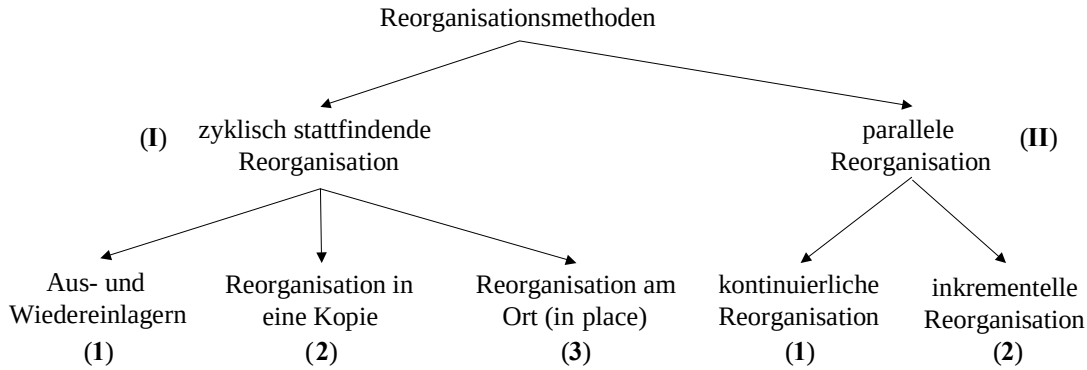


Abbildung 2.6: Reorganisationsmethoden

Parallel (**II**) arbeitende Reorganisationsmethoden nehmen Bereinigungen bzw. Veränderungen physischer Speicherungsstrukturen parallel zum laufenden Datenbankbetrieb vor. Dabei werden meist sehr kleine Granulate „nach und nach“ bearbeitet und die Reorganisationsoperationen u.U. mit anderen (Änderungs-) Operationen (z.B. des normalen Datenbankbetriebs) verknüpft.

Bei der Methode **I.1** wird zunächst der Inhalt der zu reorganisierenden Datenbankobjekte aus der Datenbank *ausgelagert* (entladen bzw. exportiert). Danach werden die Datenbankobjekte zerstört und wieder neu angelegt. Anschließend werden die zuvor ausgelagerten Daten wieder *eingelagert* (importiert bzw. geladen). Diese Möglichkeit ist bei allen DBMS gegeben und sorgt durch den meist kompletten Neuaufbau aller betroffenen Speicherungsstrukturen (Tabellen, Indexe etc.) für eine zuverlässige Beseitigung vorhandener Degenerierungen. Allerdings besitzt diese Methode auch einige Nachteile. Nach dem Auslagern der Daten stehen diese bis nach dem Abschluss des Wiedereinlagerns nicht für Anwendungsprozesse zur Verfügung. Weiterhin unterliegen die Daten in dieser Zeit – also außerhalb der Datenbank befindlich – auch nicht den Integritätssicherungsmechanismen des DBMS. Deshalb ist die Erstellung eines Datenbank-Backups vor der Nutzung dieser Reorganisationsmethode zu empfehlen. Zur Gewährleistung der Wiederherstellbarkeit nach Systemfehlern muss für die betroffenen Datenbankobjekte nach einer Reorganisation möglichst zeitnah erneut ein Backup erstellt werden. Weiterhin sollte, um den für die Reorganisation benötigten Zeitaufwand zu verringern, das Anlegen von Indexen erst erfolgen, nachdem die Daten eines Datenbankobjekts vollständig eingelagert wurden.

Bei Methode **I.2** werden die Daten intern an einen anderen Speicherort *umkopiert*. Wenn über der zu reorganisierenden Tabelle ein Index existiert, können die einzelnen Tupel auch entsprechend der Sortierung des Index am neuen Speicherort physisch abgelegt werden. Dadurch wird auch erreicht, dass die Daten nach der Reorganisation zunächst auch physisch nach dem Ordnungskriterium des Index sortiert gespeichert werden. Anschließend wird die Kopie nach dem Vorbild des Originals indexiert. Zum

Abschluss wird das Original samt zugehörigen Indexen gelöscht und die Kopie und die dazu gehörenden Indexe erhalten die Namen der ehemaligen Originale.

Das wesentliche Merkmal der Methode **I.3** ist, dass die die Tupel einer Tabelle darstellenden Sätze bzw. Indexeinträge einzeln innerhalb der bereits von der Tabelle belegten Speicherbereiche (quasi *am Ort bzw. in place*) so umkopiert werden, dass Freispeicherlücken wieder gefüllt oder angestrebte interne Sortierreihenfolgen wieder erreicht werden. Dabei müssen die auf die Tabelle verweisenden Indexe mit gepflegt werden.

Bei einer *parallelen* Reorganisation kann die Wartung der Speicherungsstrukturen durch einen *kontinuierlich* arbeitenden Reorganisationsprozess parallel zum laufenden Datenbankbetrieb durchgeführt werden (**II.1**). Bei der *inkrementellen* Methode (**II.2**) erfolgt die Reorganisation nach und nach. Die Reorganisation wird in kleinen Einheiten durchgeführt und der aktuell vom Reorganisationsprozess bearbeitete Teil des Datenbestands wird für Zugriffe von Benutzertransaktionen gesperrt, während der Rest voll verfügbar bleibt [Omi85]. Diese Methode eignet sich beispielsweise, wenn Attribute (Spalten von Tabellen) hinzugefügt oder gelöscht wurden. Dabei wird zunächst dafür gesorgt, dass die betroffenen Attribute auf der Ebene der Datenmanipulationssprache (DML) verfügbar oder eben nicht mehr verfügbar (sichtbar) werden. Die gespeicherten physischen Datensätze bleiben zunächst unverändert. Erst wenn auf einen Datensatz ohnehin zugegriffen wird, wird dieser an das neue Format angepasst und gespeichert. So wird die Strukturveränderung nacheinander (inkrementell) im laufenden Betrieb an den einzelnen Datensätzen nachvollzogen – eine große Änderung „an einem Stück“ wird also vermieden.

Neben dem durch eine Datenbankreorganisation verursachten Aufwand ergibt sich das Problem, dass die Daten der in Reorganisation befindlichen Datenbankobjekte während der Reorganisation oft nicht oder nur eingeschränkt nutzbar (d.h. gesperrt) sind. Eine Methode, die Daten über Export und Import reorganisiert, ist nur *offline*[4] durchführbar. Durch das Auslagern der Daten und das Zerstören der Schemaelemente stehen die betroffenen Datenbankobjekte nämlich vorübergehend nicht zur Nutzung zur Verfügung (was hier mit offline gemeint ist). Diese relativ starke Einschränkung steht im Widerspruch zu hohen Verfügbarkeitsanforderungen. Daher wurden Möglichkeiten zur *Online-Reorganisation* entwickelt, um zumindest die Einschränkungen in der Benutzbarkeit der Daten gering zu halten. Dabei reicht die Spanne von einer Verfügbarkeit der Daten nur für Lesezugriffe bis hin zum uneingeschränkten Zugriff, der teilweise lediglich zum Abschluss der Reorganisation kurzzeitig unterbrochen werden muss. Der Reorganisationsaufwand bleibt aber auch hier erhalten. Hinzu kommt noch der Aufwand für die Aufrechterhaltung der Verfügbarkeit. (Ähnliche Ansätze – online versus offline – sind aus dem Bereich der Backup- und Recovery-Durchführung bekannt [Stö01]).

[4] Offline bedeutet in diesem Zusammenhang, dass die in Reorganisation befindlichen Daten nicht für Zugriffe von Anwendungen zur Verfügung stehen.

2.7 Anforderungen an Datenbankreorganisationen und Einflussfaktoren

Eine Datenbankreorganisation ist, abhängig von der Menge der zu reorganisierenden Daten, wie oben schon deutlich wurde, oft ein aufwendiger Prozess. Je nach Art der Reorganisation kommt es während der Reorganisation zu mehr oder weniger starken Einschränkungen im normalen Datenbankbetrieb. Solche Einschränkungen reichen von einer vorübergehenden Verschlechterung des Antwortzeitverhaltens durch die zusätzliche Belastung des DBMS bis dahin, dass die Datenbank oder Teile davon zeitweilig nicht verfügbar sind. Da solche Einschränkungen in vielen Bereichen aber nicht oder nur in geringem Umfang tolerierbar sind, stellt die Forderung, dass die Reorganisation eines Datenbankteils den übrigen *Datenbankbetrieb möglichst wenig beeinflussen* sollte, eine Kernforderung dar.

Um dieser Kernforderung weitgehend entsprechen zu können ist es nötig, *gezielt* nur die Bereiche einer Datenbank zu *reorganisieren*, die solche Degenerierungen enthalten, deren Beseitigung einen entsprechenden Nutzen erwarten lässt. Deshalb sollte das DBMS bzw. das Reorganisationswerkzeug den Datenbankadministrator (DBA) bei der *Bestimmung des Reorganisationserfordernisses* unterstützen und möglichst gezielt Hinweise geben, wo und wann eine Datenbankreorganisation angesetzt werden sollte. Moderne Werkzeuge [BMC04, EMB03, IBM02, Que04, Ora05] sind in der Lage, Degenerierungen zu erkennen, einen statistischen Degenerierungsgrad zu bestimmen und bei Über- bzw. Unterschreitungen eine Reorganisation zu empfehlen. Eine quantifizierte Angabe über den von der Reorganisation zu erwartenden *Nutzen*, besonders bezüglich der Systemleistung, wird allerdings nicht getroffen. In diesem Zusammenhang sollte also auch eine Gegenüberstellung von Reorganisationsaufwand und zu erwartendem Nutzen (workload-abhängig) erfolgen.

Bezüglich der Vorbereitung und Durchführung einer Reorganisation von Datenbanken sollte für den Durchführenden (z.B. den DBA) möglichst *wenig organisatorischer Aufwand*, wie die Bereitstellung von Bändern oder Plattenspeicher, entstehen.

Die Durchführung der Reorganisation sollte möglichst *einfach und systemkontrolliert* erfolgen, um z.B. Bedienungsfehler zu vermeiden. Hier werden möglichst einfache Anweisungen benötigt, die alle zur Durchführung einer Reorganisation notwendigen Schritte ausführen und die Möglichkeit bieten, bei Wunsch bzw. Bedarf (z.B. wenn physische Referenzen zum Ausdrücken von Beziehungen verwendet werden) auch abhängige Datenbankobjekte mit zu reorganisieren.

Auch die Möglichkeit, Reorganisations-Jobs zu erstellen und diese zeitgesteuert zu lastarmen Zeiten ausführen zu lassen, trägt zur Reduzierung der Beeinflussung des normalen Datenbankbetriebs bei. Um hier Fehler während der Reorganisation zu vermeiden, sollte vorab geprüft werden, ob die zur Reorganisation notwendigen *Voraussetzungen erfüllt* sind, bspw. ob genügend Speicherplatz zur Verfügung steht oder ob die benötigten Zugriffsrechte vorhanden sind. Auch hier ist eine Unterstützung des DBA durch Reorganisationswerkzeuge notwendig. Die im Zusammenhang mit Reorganisationsbedarfsanalysen und der eigentlichen Durchführung von Datenbankreorganisationen verbundenen Abläufe lassen sich als

Workflows [HWW04] modellieren und somit weitgehend automatisieren. In *Abschnitt 4.4* werden derzeit verfügbare Reorganisationswerkzeuge von DBMS-Herstellern und Drittanbietern bezüglich ihrer Leistungsfähigkeit noch genauer betrachtet.

In welchem Umfang und in welchen Zeitintervallen Datenbankreorganisationen durchgeführt werden, ist von mehreren Faktoren abhängig. Hier sind zunächst sicherlich *Reorganisationsaufwand* und *–nutzen* zu nennen, die sich proportional zueinander verhalten sollten. Auch das *Datenbankschema* hat Einfluss auf Art und Häufigkeit von Datenbankreorganisationen. Beim physischen Datenbankentwurf kann durch eine geschickte Wahl von Längen und Datentypen von Attributen der Entwicklung von Degenerierungen entgegengewirkt werden, wenn die Anforderungen der auf der Datenbank arbeitenden Anwendungen dies erlauben. Die *Nutzungsform* der Datenbank stellt einen weiteren Einflussfaktor dar. Eine Datenbank, die überwiegend lesend genutzt wird, wird wesentlich langsamer (oder gar nicht) degenerieren als eine Datenbank, an der häufig Änderungen am Datenbestand vorgenommen werden. Auch die Art des Änderungsbetriebs spielt eine Rolle. Eine Mischung aus Einfüge-, Änderungs- und Löschoperationen führt sicherlich schneller zu starken Fragmentierungen als überwiegende Einfügeoperationen. Auch *Hardwarebasis* und *Betriebssystem* haben Einfluss darauf, ob bestimmte Reorganisationsmaßnahmen sinnvoll sind oder nicht. Vor allem die Parallelisierung von Datenbankoperationen setzt die Unterstützung seitens der Hardware und des Betriebssystems voraus. So wird bei der Partitionierung von Tabellen und Indexen die Unterstützung von parallelen Lese- und Schreiboperationen auf verschiedenen Datenträgern durch Hardware und Betriebssystem benötigt, um die gewünschten positiven Effekte zu erreichen.

3 Speicherorganisation, Schwellwerte, Degenerierungen

Das Thema Datenbankreorganisation ist sehr eng mit Konzepten und Strukturen zur Speicherung von Daten verbunden. Die Organisation des verwalteten Sekundärspeichers ist trotz unterschiedlicher Begriffsverwendung bei verschiedenen DBMS, auf die später noch eingegangen wird, ähnlich. Zunächst werden in *Abschnitt 3.1* die verschiedenen Organisationseinheiten vorgestellt. Anschließend werden in DBMS-Produkten häufig verwendete logische Speicher- und Zugriffskonzepte beschrieben (*Abschnitt 3.2*). Die Beschreibung gebräuchlicher interner Speicherungsstrukturen, über die die Abbildung der Speicher- und Zugriffspfadkonzepte realisiert wird, erfolgt in *Abschnitt 3.3*. Dabei wird auch auf mögliche Degenerierungen und Methoden, die das Entwicklungstempo von Degenerierungen beeinflussen können, eingegangen. In *Abschnitt 3.4* werden noch einige Speicherungskonzepte zur Unterstützung komplexer Objekte erläutert.

3.1 Sekundärspeicherorganisation bei DBMS

Die prinzipielle Speicherorganisation von gängigen DBMS-Produkten ähnelt einander weitgehend. Da von verschiedenen Produkten aber oft unterschiedliche Bezeichnungen verwendet werden, sollen hier zunächst die in dieser Arbeit verwendeten Bezeichnungen eingeführt werden. Zur Speicherorganisation gehören Verwaltungs- und Speichereinheiten (Elemente), die auf unterschiedlichen Ebenen der in Abbildung 2.1 dargestellten Schichtenarchitektur angesiedelt sind. Die Verwaltungseinheiten dienen als logische Container für die Speichereinheiten. Die Eigenschaften der Verwaltungseinheiten können auf einer gegenüber der SQL-Norm (die dieses ausklammert) erweiterten DDL-Ebene verändert werden. Solche Erweiterungen können im Rahmen von Datendefinitionsanweisungen benutzt werden, um beispielsweise die Platzierung von Datenbankobjekten zu beeinflussen. *Abbildung 3.1* zeigt den Zusammenhang der einzelnen Elemente.

Zu den *Speichereinheiten* zählen Dateien, Extents und Blöcke. Die eigentliche Speicherung von Daten erfolgt auf externen Speichermedien (typischerweise Plattenspeicher). Die Verbindung zu diesen externen Speichermedien wird bei modernen Rechnerarchitekturen und Betriebssystemen über eine Dateischnittstelle realisiert. Die Organisation dieser *Dateien* unterhalb der Schnittstelle zum Betriebssystem ist unterschiedlich. So kann eine Datei vom Dateisystem des Betriebssystems verwaltet werden. Das ist dann für die Reservierung oder Freigabe von Speicher verantwortlich und führt i.d.R. auch eine Zwischenpufferung im Arbeitsspeicher durch. Durch Reservierung und Freigabe von Speicher verschiedener Dateien kann es im Laufe der Zeit dazu kommen, dass der einer Datei zugeordnete Externspeicher in kleinen Stücken (Fragmenten) über den physischen Datenträger verstreut ist. Dies erfordert beim Lesen der Datei, gegenüber einer physisch fortlaufenden Speicherung, eine erhöhte Anzahl notwendiger Positionierungsoperationen der Lese-/Schreibköpfe, was zu Performance-Einbußen führt. Zur Verringerung der Gefahr solcher Fragmentierungen wird typischerweise bereits bei der Erzeugung von Datenbankdateien ein größerer Speicherbereich soweit

möglich zusammenhängend reserviert [Dor99b]. Eine andere Form der Dateischnittstelle stellen die sog. Raw Devices dar. Dahinter verbirgt sich ein zusammenhängender Speicherbereich, auf den über eine vom Betriebssystem zur Verfügung gestellte Dateischnittstelle direkt, auch unter Umgehung der Dateipufferung des Betriebssystems, zugegriffen werden kann. Durch hochentwickelte Hard- und Softwarekomponenten (RAID-Systeme, Volume Manager usw.) kann der für ein Dateisystem oder als Raw Device verfügbare Speicher auf unterschiedliche Art und Weise aus Teilen eines oder mehrerer Datenträger zusammengestellt werden. Die unterschiedlichen Dateiorganisationsformen bleiben dem DBMS allerdings größtenteils verborgen. Sie liegen „unterhalb“ des Datenbank-Management-Systems. Auch die Verantwortung für innerhalb der Dateien eventuell vorhandene (aus Sicht des DBMS externe) Fragmentierungen liegt außerhalb des DBMS. Deshalb wird innerhalb dieser Arbeit darauf nicht detaillierter eingegangen.

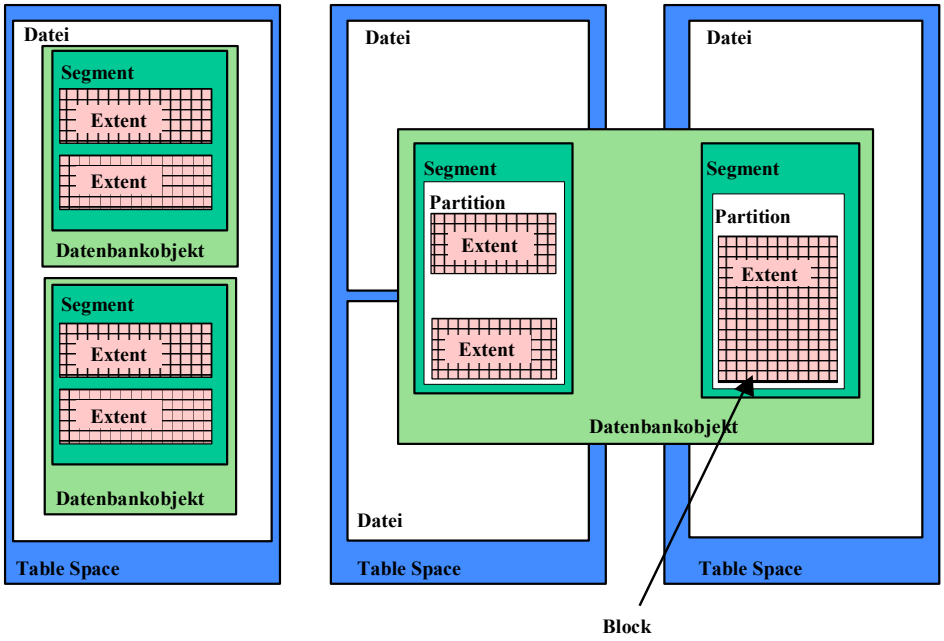


Abbildung 3.1: Sekundärspeicherorganisation im Überblick

Lese- und Schreiboperationen (I/O-Operationen) werden über die Dateischnittstelle aus Effizienzgründen nicht byteweise, sondern in *Blöcken* oder Sequenzen von Blöcken abgewickelt. Dabei entspricht die Größe eines Blocks typischerweise der Speichermenge, die mit einer I/O-Operation von einem Datenträger gelesen bzw. auf diesen geschrieben werden kann oder einem Vielfachen davon. Die wesentlichen Teile eines Blocks sind

- der Block Header, der allgemeine Verwaltungsinformationen enthält,
- eventuell eine Slot-Liste (bzw. Row Directory), die Verweise auf die einzelnen im Block gespeicherten Einträge enthält und
- der Bereich, der zur Speicherung der eigentlichen Daten bzw. Indexeinträge zur Verfügung steht.

Um Fragmentierungen zu verringern, wird der Speicherplatz von Datenbankobjekten bei Bedarf nicht blockweise, sondern in Form von *Extents* reserviert. Ein Extent ist

eine Menge physisch beieinander liegender Blöcke innerhalb einer Datei. Die Extent-Größe kann entweder vom DBMS vorgegeben sein oder sie gehört zu den (spezifizierbaren) Eigenschaften des entsprechenden Datenbankobjekts oder des das Objekt enthaltenden Table Space.

Die einem Datenbankobjekt zugeordneten Extents bilden ein *Segment*. Liegen in einem Table Space mehrere Segmente, die fortlaufend erweitert werden, so ist es möglich, dass die zu einem Segment gehörenden Extents nicht physisch fortlaufend reserviert werden können (*Abbildung 3.2*). Damit erhöht sich der Aufwand für Positionierungsoperationen der Lese-/Schreibköpfe jeweils beim Übergang zum nächsten Extent. Eine wohlüberlegte und datenbankobjektbezogene Festlegung von Extent-Größen kann solchen Fragmentierungen entgegenwirken, erschwert aber auch die Wiederverwendung freigegebener Extents für andere Datenbankobjekte. Wegen der typischerweise geringen Anzahl von Extents, die das Segment eines Datenbankobjekts umfasst, wird in verschiedenen Quellen [HL03, Nuß97, Sha95] der Einfluss der Segmentfragmentierung auf die Systemleistung (gegenüber dem Einfluss anderer Degenerierungen) als eher gering eingeschätzt. Ausgefeilte Techniken zur internen Verwaltung physischer Speicherungsstrukturen können auch eine effiziente Wiederverwendung von Freispeicherfragmenten unterschiedlicher Größe ermöglichen [Sch00].

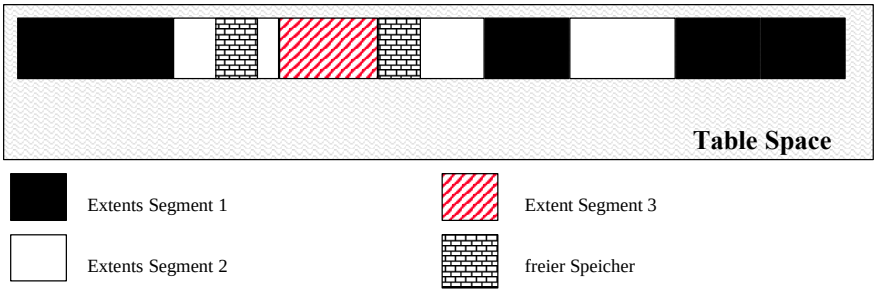


Abbildung 3.2: Segmentfragmentierung

Verwaltungseinheiten sind Table Spaces, Datenbankobjekte und Partitionen. Der gesamte Speicherplatz einer Datenbank wird in einem oder mehreren *Table Spaces* verwaltet. Ein Table Space umfasst eine oder mehrere Dateien. Table Spaces stellen um Dateien erweiterbare Container dar, die Datenbankobjekte enthalten können. Wird einem Table Space jeweils nur eine Datei zugeordnet, so kann darüber die Platzierung von Datenbankobjekten auf den Datenträgern gesteuert werden (etwa gezielt auf unterschiedliche Datenträger oder auch gemeinsam auf einem Datenträger).

Innerhalb von Table Spaces können *Datenbankobjekte* abgelegt werden. Zu den Datenbankobjekten, die für diese Arbeit von Bedeutung sind, zählen Tabellen, Indexe und Cluster. Während in Tabellen nur Daten einer Relation (im Sinne des relationalen Datenmodells) gespeichert werden, können Cluster die Daten mehrerer logisch zusammengehöriger Relationen enthalten. Auf das Cluster-Konzept wird in *Abschnitt 3.2* noch detaillierter eingegangen.

Tabellen und Indexe können bei Bedarf nach bestimmten Kriterien weiter unterteilt (partitioniert) werden. Diese Unterbereiche werden als *Partitionen* bezeichnet. Die Partitionen werden Table Spaces zugeordnet, in denen für jede Partition ein eigenes Segment angelegt wird.

Durch die von DBMS zur Datenspeicherung verwendete Speicherorganisation sind in der Regel gewisse Grenzen (Schwellwerte) bezüglich der maximalen Größe von Segmenten bzw. der maximalen Anzahl Extents, die ein Segment enthalten kann, gesetzt. Weiterhin ist es möglich, dass der in einem Table Space zur Verfügung stehende Speicherplatz erschöpft ist und der Table Space vor weiteren Operationen erst erweitert werden muss.

Auch auf Betriebssystemebene existieren solche Grenzen (bspw. maximale Dateigrößen). DBMS- und Betriebssystemhersteller haben in den vergangenen Jahren intensiv daran gearbeitet, solche Grenzen zu überwinden oder die Grenzen so weit zu verschieben, dass sie in der Praxis kaum erreicht werden können. Allerdings steht u.U. die Frage nach der Effizienz solcher Implementierungen. Oft ist der Verwaltungsaufwand für selbsterweiternde Strukturen mit potenziell „unendlichem“ Fassungsvermögen wesentlich höher als bei Strukturen, bei denen klare Grenzen gesetzt sind. Dort ist aber meist wieder der Administrationsaufwand höher.

Ist z.B. die Anzahl der Extents, die für eine Tabelle reserviert werden können, begrenzt, weil nur ein Block für Verwaltungsdaten von Extents zur Verfügung steht, so muss der DBA, wenn die Grenze erreicht wird, mit einer Datenbankreorganisation dafür sorgen, dass zukünftig größere Extents reserviert werden. Sonst wäre beim weiteren Anwachsen der Tabelle der Datenbankbetrieb gefährdet. Ist die Zahl der Extents nicht begrenzt, z.B. weil weitere Verwaltungsblöcke reserviert und in einer Kette verwaltet werden können, so tritt die akute Gefährdung des Datenbankbetriebs nicht ein. Dafür ist hier ein höherer Aufwand beim Verwalten und auch beim Auffinden der Extents durch das DBMS nötig. Dies wirkt sich negativ auf die Systemleistung aus und führt u.U. auch wieder zu einem Reorganisationsbedarf.

Welche Grenzen existieren und wo die jeweilige Grenze genau liegt, ist systemabhängig und kann allgemein gültig kaum erfasst werden. Hier muss der DBA die in der speziellen Systemkonfiguration auftretenden Grenzen ermitteln. Zusätzlich sind geeignete Methoden anzuwenden, um Entwicklungen, die zum Erreichen solcher Grenzen führen können, rechtzeitig zu erkennen und zu überwachen. Dabei wirken als wesentliche Einflussfaktoren die noch verbleibende Zeit, bis der normale Datenbankbetrieb auf Grund der erreichten Grenzwerte nicht mehr aufrecht erhalten werden kann und die Einschränkungen im normalen Datenbankbetrieb, die die Reorganisation verursacht. Es muss ein Kompromiss gefunden werden, der einerseits die rechtzeitige Reorganisation sicherstellt und andererseits die Verfügbarkeitsanforderungen der Anwender weitestgehend berücksichtigt.

3.2 Speicher- und Zugriffskonzepte

3.2.1 Tabellen und Indexe

Für rein („klassisch“) relationale Systeme ist die Beschreibung der logischen Datenstrukturen recht einfach. Daten werden in einfacher Tabellenform, ohne mengenwertige oder zusammengesetzte Attribute dargestellt. Weder die Reihenfolge der Attribute noch die der Tupel einer *Tabelle* sind vorgeschrieben. Eine Tabelle kann daher auf eine ungeordnete Folge von Datensätzen abgebildet werden. Um Zugriffe auf einzelne Sätze bzw. kleinere Satzmengen zu beschleunigen, können häufig für Suchoperationen verwendete Attribute bzw. Attributkombinationen (*Suchschlüssel*) indexiert werden. Über den *Index* wird eine Zuordnung zwischen Suchschlüsselwert und den Speicherorten der den Suchschlüsselwert enthaltenden Datensätze vorgenommen. Damit werden zusätzliche Zugriffspfade geschaffen. Im Zusammenhang mit der Indexierung bieten Datenbank-Management-Systeme die Möglichkeit, Sätze nach dem Suchschlüssel eines Index sortiert (geclustert) zu speichern. Ein solcher Index wird häufig auch als *Clustered Index* bezeichnet. Durch die *wertebasierte Clusterung* kann insbesondere für Bereichsanfragen und Anfragen, bei denen das Ergebnis nach dem Ordnungskriterium des Index sortiert benötigt wird, der notwendige Such- und Sortieraufwand verringert werden. Deshalb sollte jeweils der Schlüssel zur Clusterung ausgewählt werden, der bezüglich der Tabelle zur Ausführung der genannten Operationen die größte Systemlast – und damit den größten Nutzen durch die Clusterung – erwarten lässt. Abhängig von der internen Realisierung solcher Indexe (z.B. bei der Invertierung von als Heap organisierten Datenbereichen) kann die Clusterung im Laufe der Zeit durch Einfüge- und Änderungsoperationen verloren gehen.

Die bei den derzeit verbreiteten objektrelationalen DBMS erweiterten Möglichkeiten zur Datenmodellierung erlauben, zumindest auf konzeptueller Ebene, benutzerdefinierte Datentypen, die Unterstützung einfacher Kollektionstypen und die Schachtelung von Tabellen (Nested Tables). Einige dieser Konstrukte wurden mittlerweile auch in die SQL-Norm integriert [Luf05]. DBMS-Produkte bilden die Erweiterungen intern meist jedoch (noch) auf rein relationale Strukturen ab. Dadurch werden Redundanzen und Anomalien bei Update-Operationen vermieden. Allerdings müssen viele Verbundoperationen ausgeführt werden, wenn bspw. alle Daten eines komplexen Objekts benötigt werden. Eine andere Variante ist die Speicherung der Daten komplexer Objekte im Binärformat, dessen interne Struktur dem DBMS aber verborgen bleibt. Vorschläge zur Verbesserung der Unterstützung von Kollektionen und komplexen Objekten, die Ergebnisse aktueller Forschungsarbeiten (z.B. [Luf05, Ska06]) sind, werden in *Abschnitt 3.4* näher beschrieben.

3.2.2 Horizontale Partitionierung von Tabellen

Zur Beschleunigung von gegen sehr große Tabellen gerichteten Anfragen und zur Verbesserung der Administrierbarkeit bieten einige DBMS-Produkte (z.B. Oracle, DB2, Informix) Konzepte zur *horizontalen Partitionierung* von Tabellen und Indexen an. Bei dieser Partitionierungsform wird eine Tabelle horizontal und vollständig in

disjunkte Teiltabellen (*Partitionen*) unterteilt, die dann gezielt physischen Bereichen der Datenbank zugeordnet werden (*Abbildung 3.3*). Die einzelnen Partitionen können dann typischerweise einzeln administriert werden und Wartungsaufgaben können zielgerichteter und auf kleinere Einheiten beschränkt („lokaler“) durchgeführt werden.

Wie die Aufteilung erfolgt, wird durch ein *Partitionierungsschema* unter Verwendung einer bestimmten *Partitionierungsstrategie* definiert [Now01b]. Dazu können prinzipiell die folgenden Strategien angewendet werden, die allerdings nicht von allen DBMS-Produkten in jener Breite angeboten werden:

- Bei der *Round-Robin-Strategie* werden die Tupel einer Tabelle bei Einfügung der Reihe nach, also gleichmäßig, auf die einzelnen Partitionen verteilt. Dadurch kann beim späteren lesenden Zugriff eine gute Lastbalancierung erreicht werden. Weiterhin wird eine Beschleunigung von sequenziellen Suchoperationen möglich, indem die einzelnen Partitionen parallel sequenziell durchsucht werden und am Ende die Ergebnisse der (Teil-)Anfragen zusammengeführt werden. Punkt- oder Bereichsanfragen profitieren von dieser Strategie jedoch nicht.
- Bei der *Hash-Strategie* wird aus dem Wert eines oder mehrerer Attribute eines Tupels ein Hash-Wert berechnet, der die Zuordnung zu einer Partition bestimmt. Neben der Möglichkeit, sequenzielle Suchoperationen zu parallelisieren, profitieren bestimmte Punktanfragen von dieser Partitionierungsstrategie.
- Bei *ausdrucksbasierten Partitionierungsstrategien* bestehen die größten Freiheiten. So können Tupel z.B. bereichsweise nach dem Partitionierungsschlüssel in die verschiedenen Partitionen einer Tabelle eingeordnet (wertebereichsbezogen geclustert) werden. Sequenzielles Suchen und Bereichsanfragen profitieren von dieser Partitionierungsform neben der Clusterung auch dadurch, dass Partitionen, die definitiv keine Tupel der Treffermenge enthalten können, u.U. bei der Suche ausgeschlossen werden können. Dies führt u.U. zu einer erheblichen Verringerung der zu durchsuchenden Datenmenge.

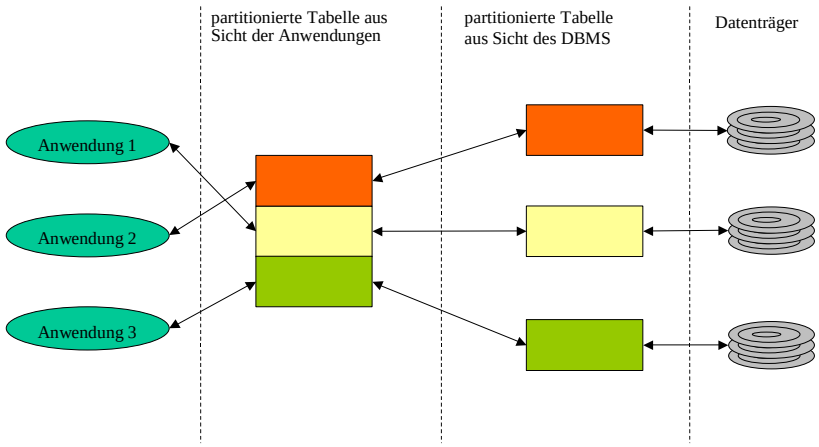


Abbildung 3.3: Partitionierung von Tabellen

Gegenüber Anwendungsprogrammen wird die Partitionierung verborgen (Transparenz). DBMS behandeln sie bezüglich der Speicherung i.d.R. wie eigenständige Tabellen. Jeder Partition wird ein eigenes Segment zugeordnet. Zur Ausnutzung aller Vorteile, die hash- und ausdrucksbasierte Partitionierungsstrategien bieten, ist es notwendig, dass die an der Anfrageausführung beteiligten Komponenten Kenntnis über die Partitionierungsstrategie besitzen und dass die Selektionsbedingungen der Anfragen zu diesem Schema „passen“. Bei Entwurf und Überarbeitung von Partitionierungsschemata müssen daher Informationen oder Annahmen über die gegen die entsprechenden Tabellen gerichteten Anweisungen (Workload) berücksichtigt werden.

Die Anwendung von Partitionierungsmechanismen kann zur Verbesserung der Administrierbarkeit beitragen. Administrationsmaßnahmen können zielgerichtet auf kleinere Einheiten angewendet werden. In gewisser Hinsicht kann Partitionierung auch zu einer Erhöhung der Datenverfügbarkeit beitragen. Fällt ein Teil einer unpartitionierten Tabelle, die innerhalb eines Table Space auf mehrere Dateien auf unterschiedlichen Datenträgern verteilt ist, durch einen Datenträgerfehler aus, so steht die gesamte Tabelle nicht mehr zur Verfügung. Für partitionierte Tabellen kann u.U. die Verarbeitung auf den noch verfügbaren Partitionen weiter ausgeführt werden. Einige DBMS-Produkte auch können so konfiguriert werden, dass durch Datenträgerfehler oder die Durchführung von Administrationsmaßnahmen ausgefallene Partitionen „übersprungen“ werden. Allerdings sollte vorab genau geprüft werden, ob die ggf. damit verbundenen Fehler bei der Erstellung der Treffermengen von Such- und Update-Operationen toleriert werden können (was etwa bei statistischen Auswertungen der Fall sein kann).

3.2.3 Tabellenübergreifende Clusterung

Vorrangig zur Unterstützung von Verbundoperationen (hier Equi Joins) bietet das DBMS-Produkt Oracle (als eines von relativ wenigen Produkten) die Möglichkeit der *tabellenübergreifenden Clusterung* von Daten. Das Konzept ist besonders bei Überlegungen zu Reorganisationen auf der Definitionsebene von Interesse. Hier können Tupel unterschiedlicher Tabellen, die über Schlüssel (*Cluster-Schlüssel*) in Beziehung (typischerweise 1:N) zueinander stehen, gemeinsam in einem Datenblock gespeichert werden. Der Zugriff auf die Tupel erfolgt über einen gemeinsamen Zugriffspfad, der als „normaler“ B^*-Baum-Index oder als Hash-Index realisiert sein kann. Der Speicherbereich, der die zu einem Cluster-Schlüsselwert bzw. Hash-Wert gehörenden Tupel aufnimmt, wird als *Bucket* bezeichnet. Die in einem Bucket gespeicherten Tupel werden auch als *Cluster-Gruppe* bezeichnet. *Abbildung 3.4* zeigt ein Beispiel, bei dem jeweils die persönlichen Daten eines Mobilfunkteilnehmers und seine Verbindungsdaten gemeinsam in einem Bucket gespeichert werden. Die Zahl der Tabellen, die an einer Cluster-Definition beteiligt sein können und deren Daten in einem Cluster abgespeichert werden können, ist theoretisch nicht begrenzt.

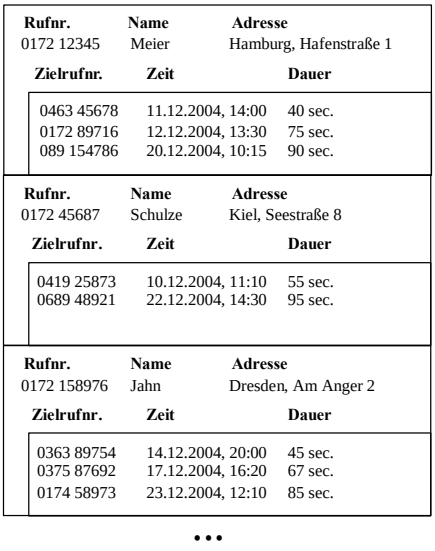


Abbildung 3.4: Tabellenübergreifende Clusterung von Tupeln

Anhand der in *Abbildung 3.5* dargestellten Beispiele sollen kurz Einsatzgebiete und Grenzen der Cluster-Bildung erläutert werden. Primärschlüsselattribute sind unterstrichen, Fremdschlüssel kursiv und Cluster-Schlüssel fett dargestellt.

Beispiel 1:

```
Filiale(Filialnr,PLZ,Ort,Straße,...);
Mitarbeiter(Manr,Name, Vorname,...,Filialnr);
Kunde(Kdnr,Name,Vorname,...,Filialnr);
```

Beispiel 2:

```
Filiale(Filialnr,PLZ,Ort,Straße,...);
Mitarbeiter(Manr,Name,Vorname,PLZ,Wohnort,...,Filialnr);
Kunde(Kdnr,Name,Vorname,PLZ,Wohnort,...,Filialnr);
```

Beispiel 3:

```
Flug(Flugnr,Start,Ziel,...);
Passagier(Panr,Name,Vorname, ...);
Buchung(Buchungsnr,Flugnr,Panr,Platznr,...);
```

Abbildung 3.5: Beispiele zur Cluster-Bildung

Die Beispiele 1 und 2 zeigen einen kurzen Ausschnitt aus einer Unternehmensdatenbank, die Daten von Filialen und deren Mitarbeitern sowie die Daten der von den einzelnen Filialen betreuten Kunden enthält. Die Filialnummer, die in allen Tabellen enthalten ist, ist ein typischer Kandidat für einen Cluster-Schlüssel. Die Daten der Mitarbeiter einer Filiale und die Daten der Kunden der Filiale würden dicht bei den Filialdaten gespeichert werden. Die Clusterung kann aber auch für Nicht-Schlüsselattribute, also für beliebige Attribute, vorgenommen werden.

In Beispiel 2 dienen die in den Tabellen in Form von Sekundärschlüsseln enthaltenen Ortsangaben zur Clusterung. Hier werden die Daten von Filialen, Kunden und Mitarbeitern, die am gleichen Ort ansässig sind, physisch dicht beieinander gespeichert. Die Objekte der im Cluster gespeicherten Tabellen müssen also nicht zwangsläufig hierarchisch zueinander strukturiert sein. M:N-Beziehungen können über den Cluster-Schlüssel allerdings nicht dargestellt werden. Beispiel 3 zeigt einen Ausschnitt aus einer Datenbank zur Buchung von Flügen. Die Tabelle Buchung dient dabei zum Ausdrücken der M:N-Beziehung zwischen Passagieren und Flügen über die beiden Fremdschlüsselattribute `Panr` und `Flugnr`. Es ist auch möglich, einen Cluster-Schlüssel aus einer Attributkombination zu bilden (mehrattributige Cluster-Schlüssel). Allerdings müssen dann alle Tabellen, die im Cluster untergebracht werden sollen, diese Attributkombination vollständig enthalten und nicht nur Teile des Schlüssels (wie bei M:N üblich und in Beispiel 3 gezeigt). Durch diesen gemeinsamen Cluster-Schlüssel ist die Anzahl der Hierarchieebenen, über die Datenobjekte geclustert werden können, auf zwei begrenzt (es gibt sozusagen keine Möglichkeit des „Clusterns innerhalb des Clusters“).

Die Vorteile der tabellenübergreifenden Clusterung sind dann besonders ausgeprägt, wenn Zugriffsoperationen hauptsächlich über den Cluster-Schlüssel erfolgen und wenn die logisch in Beziehung zueinander stehenden Daten auch häufig zusammen benötigt werden. Hier kann die physisch zusammenliegende Speicherung der Tupel die Anzahl der für den Verbund notwendigen Blockzugriffe gegenüber der in relationalen Systemen sonst üblichen getrennten Speicherung deutlich vermindern [GE98]. In [SD03] wird das Potenzial der Cluster-Bildung am Beispiel der Unterstützung kollektionswertiger Attribute gezeigt. Die Vermischung der Tupel unterschiedlicher Tabellen führt bei auf einzelne Tabellen des Clusters angewendete sequenzielle Suchoperationen aber auch dazu, dass meist deutlich mehr Datenblöcke durchsucht werden müssen, als bei der getrennten Speicherung der Tabellen. Es gilt die bekannte Eigenschaft: Man kann nur nach einem Kriterium clustern (entweder *innerhalb* der Tabelle oder *tabellenübergreifend*). In die Entscheidung darüber, ob eine tabellenübergreifende Clusterung sinnvoll ist, müssen, wie z.B. bei Entscheidungen über die Partitionierung von Tabellen auch, Informationen oder Annahmen über die gegen die zu clusternden Tabellen gerichteten Anweisungen (Workload-Charakteristika) berücksichtigt werden.

3.3 Interne Strukturen zur Datenspeicherung und Degenerierungen

3.3.1 Datenspeicherung als Heap

Die Speicherung von Informationen in Datenbanken erfolgt in Blöcken, aus denen komplexere interne Speicherungsstrukturen gebildet werden. Die Speicherung von Tupeln und Indexinformationen erfolgt in Form von Sätzen und Einträgen, die in Blöcken abgelegt werden.

Eine häufig von relationalen Systemen verwendete Struktur zur Speicherung von Daten ist die **Heap-Struktur**. Die einzelnen Sätze werden fortlaufend in Zugangsreihenfolge untergebracht. Zur Speicherung wird ein Segment angelegt, das

bei Bedarf um weitere Extents vergrößert wird (vgl. Abschnitt 3.1). Wird durch Löschoperationen oder tupelverkürzende Änderungsoperationen Speicher frei, so kann (oder soll) dieser meist nicht unmittelbar wieder belegt werden. Das führt sukzessive zu einer Vermischung von freiem und belegtem Speicher innerhalb der Blöcke. Der frei gewordene Speicher (*eingestreuter Freiplatz)* liegt meist in so kleinen Einheiten vor, dass er von der Freispeicherverwaltung häufig nicht sinnvoll wieder vergeben werden kann, bzw. dieser zunächst nicht wieder zur Verfügung gestellt wird. Um die Entwicklung von Freiplatzfragmenten zu verlangsamen, legt bspw. das DBMS-Produkt Oracle für jedes Segment Freispeicherlisten an, in die Blöcke eingeordnet werden, deren Belegungsgrad nach einer nahezu vollen Belegung unter eine definierte Schwelle (definiert über den Parameter PCTUSED) sinkt. Bei Einfügeoperationen wird versucht, Sätze zunächst in einem in einer Freiliste befindlichen Block unterzubringen, bevor neue Blöcke belegt werden. Mit Einführung des *Automatic segment-space management* wurden unter Oracle Bitmap-Listen eingeführt, die u.U. die Konfiguration von Freispeicherlisten und Parametern wie PCTFREE und PCTUSED überflüssig machen [Ora03a]. Eingestreuter Freiplatz führt zu einer *Verschlechterung der Speicherauslastung* und zur Erhöhung des I/O-Aufwands beim sequenziellen Suchen, da die Freiplatzfragmente mitgelesen werden müssen. Bei Zugriffen über Indexe hat in Datenbereiche eingestreuter Freiplatz kaum Auswirkungen. Allerdings ist eine vollständige Vermeidung eingestreuten Freiplatzes i.d.R. auch nicht anzustreben. Bei Einfüge- und Änderungsoperationen kann eine „Freispeicherreserve“ in den Datenblöcken durchaus positive Auswirkungen haben, die nachfolgend noch näher erläutert werden.

Wenn die Länge von Sätzen die Größe des in den Blöcken zur Datenspeicherung verfügbaren Platzes übersteigt und das DBMS dies zulässt, müssen die Sätze auf mehrere Blöcke verteilt (fragmentiert) werden. Für die *Fragmentierung von Sätzen* [Dor99b] sind unterschiedliche Vorgehensweisen denkbar. Einige davon soll das folgende (zugegebenermaßen konstruierte) einfache Beispiel verdeutlichen.

Wenn ein Satz (inkl. des Verweises auf das nachfolgende Fragment) das 1.75-fache des in einem Block zur Datenaufnahme verfügbaren Speichers belegt, so könnten vier Sätze (S1 ... S4), wie im folgenden Bild gezeigt, in sieben oder acht Speicherblöcken (B1 ... B7 bzw. B8) untergebracht werden (*Abbildung 3.6*).

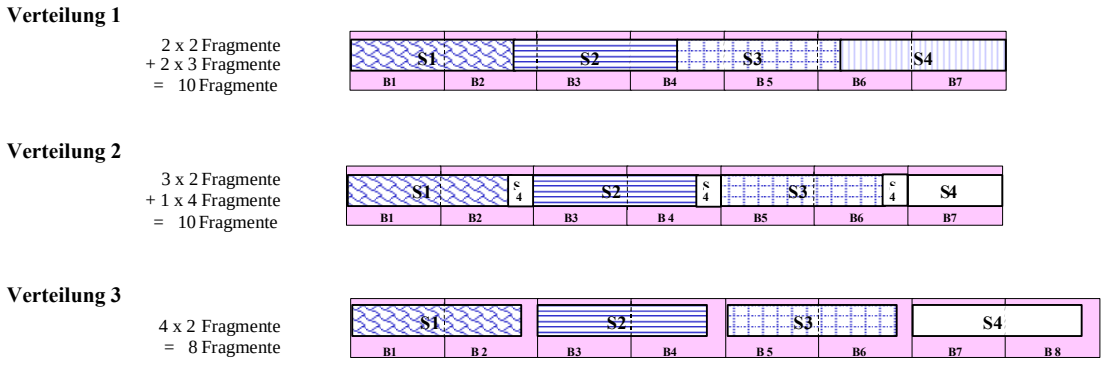


Abbildung 3.6: Verteilungsvarianten von Sätzen

Tabelle 3.1 zeigt einen Vergleich der dargestellten Varianten bezüglich benötigtem Speicherplatz in Blöcken, der Anzahl anfallender Blockzugriffe beim sequenziellen Suchen nach den vier Sätzen und der Anzahl Zugriffe auf Datenblöcke bei Punktanfragen unter Nutzung von Indexen, wobei eine Gleichverteilung der Zugriffe auf die Sätze angenommen wird.

Speicherplatz in Blöcken	Aufwand für sequenzielles Suchen	Aufwand für Punktzugriffe
7	7	2,5
7	7 bzw. 9	2,5
8	8	2

Tabelle 3.1: Vergleich der Varianten

Bei den ersten beiden Verteilungen wird der zur Verfügung stehende Speicher optimal ausgenutzt. Diese optimale Ausnutzung des Speichers führt allerdings dazu, dass gegenüber der dritten Verteilung eine höhere Anzahl Satzfragmente entsteht. Bei *Verteilung 1* werden zwei Sätze (sozusagen „ohne echte Not“) in drei anstelle der mindestens notwendigen zwei Fragmente aufgeteilt. Diese Erhöhung der Fragmentzahl wird beim sequenziellen Suchen ohne größere Auswirkungen bleiben. Voraussetzung dafür ist, dass es gelingt, die Verteilung bei Änderungs- und Löschoperationen aufrecht zu erhalten. Dies ist allerdings mit vertretbarem Aufwand kaum zu realisieren.

Bei *Verteilung 2* entspricht die Gesamtzahl der Fragmente der von Verteilung 1. Die Sätze werden nur anders aufgeteilt. Beim sequenziellen Suchen verursacht die Fragmentierung von Satz S4 bei Verteilung 2 entweder eine deutliche Erhöhung der Anzahl Blockzugriffe (wenn immer erst alle Fragmente eines Satzes gelesen werden, bevor der nächste Satz verarbeitet wird) oder es muss ein hoher Aufwand für das Zwischenspeichern und „Merken“ einzelner Satzfragmente betrieben werden.

Bei *Verteilung 3* wird mehr Speicher benötigt als bei den ersten beiden Verteilungen. Der Aufwand für die sequenzielle Verarbeitung ist höher als bei Verteilung 1. Bei gleich verteilten Zugriffen über Punktanfragen sind aber jeweils nur zwei Zugriffe auf Datenblöcke nötig.

Als Zugriffsmuster für eine Tabelle muss i.d.R. ein Mix aus sequenziellem Suchen und Punktanfragen (und Bereichsanfragen unter Nutzung von Indexen) angenommen werden. Auch der Aufwand für Einfüge-, Lösch- und Änderungsoperationen muss berücksichtigt werden. Deshalb verteilen Datenbank-Management-Systeme üblicherweise Sätze auf jeweils möglichst wenige Blöcke (ähnlich Verteilung 3). Das heißt, je nach Satzlänge werden zunächst ein Block oder mehrere Blöcke vollständig gefüllt. Ein verbleibendes Restfragment wird in einem weiteren Block, der dann nicht vollständig gefüllt ist, gespeichert. Der dadurch entstehende eingestreute Freiplatz wird toleriert. Um den Freiplatzanteil möglichst klein zu halten, speichern manche

DBMS-Produkte mehrere Restfragmente gemeinsam in einem Block, wenn das deren Länge erlaubt.

3.3.2 Indexierung bei getrennter Speicherung von Daten und Indexen

Zur Beschleunigung von Zugriffen auf einzelne Sätze bzw. kleinere Satzmengen werden durch **Indexierung** weitere Zugriffspfade geschaffen. Die Indexierung erfolgt dabei über einen Indexierungsschlüssel, der aus den Werten eines Felds bzw. einer Kombination von Feldern gebildet wird. Über den Index wird eine Zuordnung von Werten des Indexierungsschlüssels zu den Speicherorten der Sätze vorgenommen, die die entsprechenden Schlüsselwerte enthalten. Diese Zuordnung erfolgt über Wertepaare, die aus dem jeweiligen Wert des Indexierungsschlüssels und Verweisen auf die Speicherorte der die Schlüsselwerte enthaltenden Sätze gebildet werden. Dabei können logische Verweise verwendet werden, die z.B. dem Indexierungsschlüsselwert den korrespondierenden Schlüsselwert eines anderen (primären) Index zuordnen[5] oder physische Verweise (Zeiger), die oftmals als Tupelidentifikatoren (*TID*) oder Record Identifier (*RID*) bzw. Row Identifier (*ROWID*) bezeichnet werden. Tupelidentifikatoren enthalten typischerweise einen Verweis auf den physischen Speicherort eines Datensatzes (u.a. die Blocknummer). Die Speicherung der Indexe erfolgt meist von den Daten getrennt in jeweils eigenen Segmenten. Als Organisationsform für Indexe werden größtenteils Varianten von B-Bäumen (B^+-Bäume bzw. B^*-Bäume) verwendet. Bei Tabellen, deren Daten als Heap gespeichert werden, werden auf der Blattebene des Indexbaums Verweise auf die zu den Schlüsselwerten gehörenden Datensätze gespeichert, da die Datensätze selbst hier außerhalb des Baums abgelegt werden. Beim Zugriff auf einen Datensatz (Index Lookup) wird zunächst der Indexbaum von der Wurzel beginnend durchsucht. Anschließend wird auf den Datenblock zugegriffen, auf den der Zeiger in der Blattebene verweist (*Abbildung 3.7*).

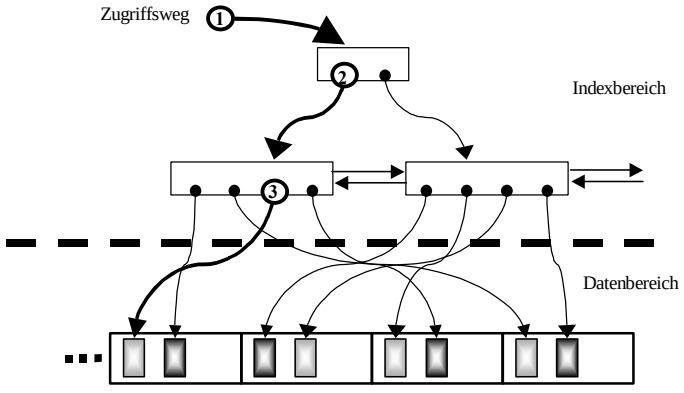


Abbildung 3.7: Blockzugriffe bei einem Index Lookup

[5] Derartige Verweiskonzepte werden bspw. bei indexorganisierten Tabellen, auf die noch eingegangen wird, verwendet.

Durch Änderungsoperationen können Datensätze verlängert werden. Dies kann dazu führen, dass die betroffenen Datensätze nicht mehr in den ursprünglichen Datenblöcken gespeichert werden können und komplett in andere Datenblöcke verschoben werden müssen. Bei der Verwendung physischer Verweise müssten diese entsprechend aktualisiert werden. Dies ist u.U. mit einem erheblichen Aufwand verbunden, besonders wenn die Tabelle nach mehreren Kriterien indexiert ist. Zur Vermeidung dieses aufwendigen Nachpflegens von Indexen wird am ursprünglichen Speicherort der betroffenen Datensätze jeweils ein Zeiger (Stellvertreter) auf den neuen Speicherort abgelegt (*Abbildung 3.8*). Im Zusammenhang mit Satzauslagerungen wird auch von *migrierten Tupeln* gesprochen.

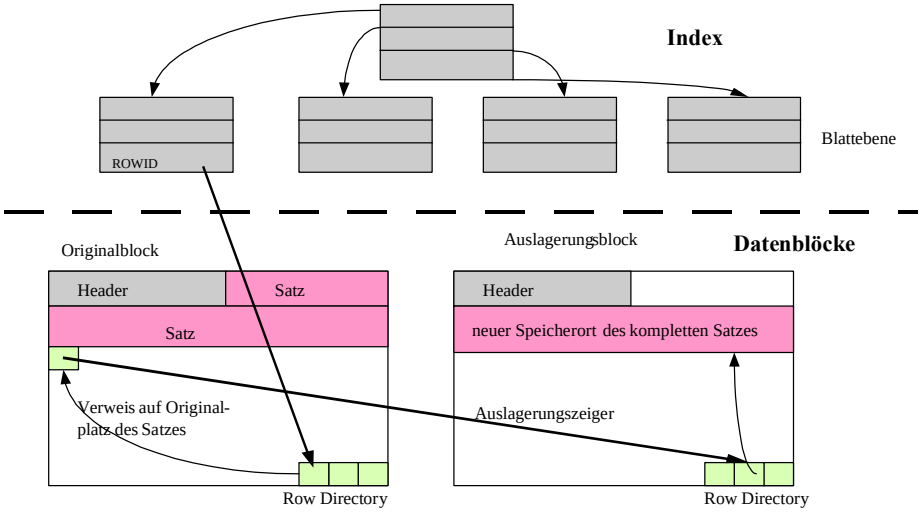


Abbildung 3.8: Satzauslagerung

Besonders beim Verschieben langer Datensätze entstehen im ursprünglichen Block Freispeicherlücken, die beim sequenziellen Suchen zu einer Aufwandserhöhung führen. Ein sofortiges Verfolgen der Auslagerungszeiger würde durch die damit verbundenen Sprünge bei sequenziellem Suchen ebenfalls zu einer u.U. erheblichen Aufwandserhöhung führen. Da nach dem relationalen Datenmodell Tupel aber prinzipiell ungeordnet vorliegen, ist ein Verfolgen der Auslagerungszeiger zur Einhaltung bestimmter Reihenfolgen nicht notwendig. Sortierungen werden oft erst vorgenommen, nachdem die Treffermenge einer Suchoperation zusammengestellt wurde. Wird aber über Index Lookups auf solche ausgelagerten Datensätze zugegriffen, so müssen die Zeiger verfolgt werden. Die damit verbundene Erhöhung des Aufwands (also Höhe des Indexbaums plus zwei Zugriffe) deutet der Zugriff Nummer 4 in *Abbildung 3.9* an.

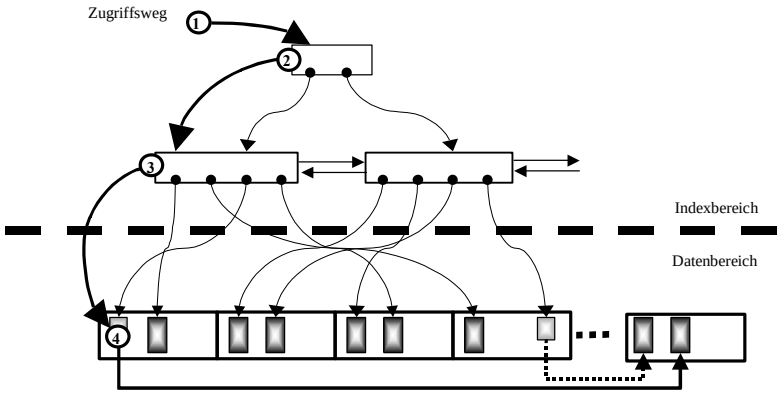


Abbildung 3.9: Index Lookup mit Auslagerung

Zur Unterstützung von Bereichsanfragen (Index Range Scans) werden die Blätter des Indexbaums in den meisten Implementierungen doppelt miteinander verkettet. Bei der Verarbeitung werden dann die einzelnen Sätze gemäß dem Ordnungskriterium der Tabelle gelesen (*Abbildung 3.10*).

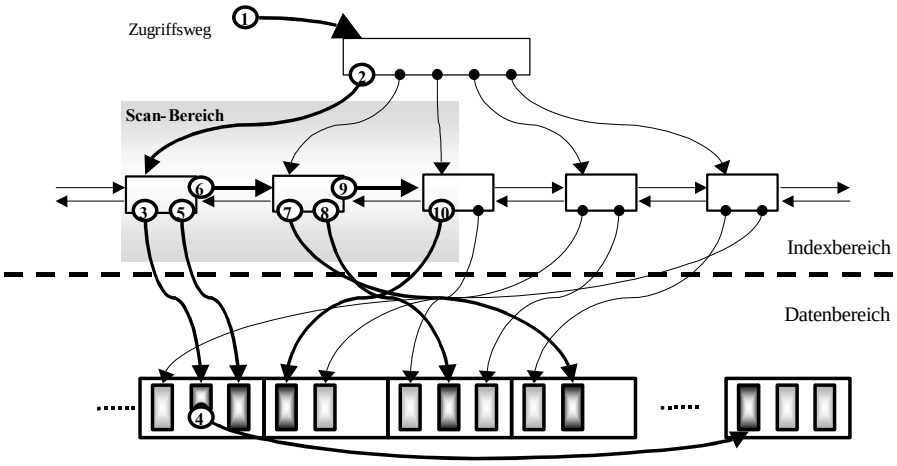


Abbildung 3.10: Index Range Scan mit Auslagerung

Die nacheinander ausgeführten Zugriffe im zu durchsuchenden Bereich werden durch die Nummerierung angedeutet. Der Zugriff mit der Nummer 4 zeigt wieder eine Aufwandserhöhung, die durch das Verfolgen eines vorhandenen Auslagerungszeigers anfällt. Im vorliegenden Beispiel sind also insgesamt zehn (logische) Blockzugriffe notwendig:

- vier auf Indexblöcke
- fünf auf „normale“ Datenblöcke
- einer auf den Auslagerungsblock

Zur Verringerung der Gefahr von Satzauslagerungen ist es sinnvoll, einen gewissen Speicheranteil in den Datenblöcken für tupelverlängernde Änderungsoperationen freizuhalten. Es ist also nicht immer erstrebenswert, allen Freiplatz in Datenbereichen mit einer Reorganisation zu beseitigen. DBMS-Produkte bieten teils

Steuerparameter an (z.B. Oracle mit dem Parameter PCTFREE), mit denen festgelegt werden kann, welcher Speicherplatzanteil in Datenblöcken als Reserve für satzverlängernde Änderungsoperationen freigehalten werden soll. Eine weitere Möglichkeit zur Verhinderung der Entstehung von migrierten Tupeln wäre die ausschließliche Verwendung von Datentypen fester Länge bei der Tabellendefinition. Allerdings sind Datentypen variabler Länge, insbesondere auch Zeichenketten variabler Länge, aus Gründen der ökonomischen Verwendung von Speicherplatz eingeführt worden. Würden beispielsweise nur Zeichenketten fester Länge verwendet werden, so würde jeweils Speicher für Zeichenketten der maximalen Länge belegt, unabhängig davon, wie lang die jeweils zu speichernde Zeichenkette wirklich ist. Das führt zu einer Verschwendung von Speicherplatz, die oftmals größere negative Auswirkungen hat als die sukzessive Entstehung von migrierten Tupeln. Diese Alternative kommt also kaum in Frage.

Durch die Speicherung von Daten in Zugangsreihenfolge bei Heap-Tabellen, durch Satzauslagerungen und durch die Möglichkeit, Daten nach unterschiedlichen Kriterien zu indexieren, entspricht die *physische Sortierung* der Daten i.d.R. nicht dem Ordnungskriterium eines jeweils betrachteten Index. Solche Indexe werden auch als *nicht geclusterte Indexe* (Non Clustered Index) [SHS05] bezeichnet. Ein Maß für den Grad der Sortierung von Daten nach dem Ordnungskriterium eines Index ist der sog. *Clustering Factor*. Werden Daten nach einem bestimmten Kriterium sortiert benötigt, so steigt der Sortieraufwand, wenn diese nach dem betrachteten Kriterium nur wenig vorsortiert gespeichert sind. Auch bei Bereichsanfragen unter Nutzung von Indexen (Index Scans) steigt der Aufwand, weil Datenblöcke evtl. mehrfach gelesen werden müssen.

In der Literatur wird zwischen dünn besetzten und dicht besetzten Indexen unterschieden [SHS05]. Bei *dünn besetzten Indexen* wird nicht für jeden Wert des Indexierungsschlüssels ein Eintrag im Index gespeichert. Liegen die Daten intern nach dem Indexierungsschlüssel sortiert vor, so reicht es aus, wenn im Index ein Eintrag je Datenblock vorhanden ist. Bei *dicht besetzten Indexen* wird für jeden indexierten Datensatz eine entsprechende Zuordnungsinformation gespeichert. Solche Indexe werden üblicherweise verwendet, wenn die Daten ausgelagert (z.B. als Heap organisiert) gespeichert werden. Weiterhin kann noch in eindeutige (unique) Indexe und nicht eindeutige (non unique) Indexe unterschieden werden. Bei einem *eindeutigen Index* kommt jeder Schlüsselwert nur einmal im Index vor und zu jedem Schlüsselwert existiert nur ein Datensatz, auf den verwiesen wird. Eindeutige Indexe werden üblicherweise zur Realisierung von Primärzugriffspfaden genutzt. Bei *nicht eindeutigen Indexen* können die Werte des Indexierungsschlüssels jeweils in mehreren Datensätzen vorkommen. Solche Indexe werden häufig zur Realisierung von Sekundärzugriffspfaden verwendet und daher auch als *Sekundärindexe* bezeichnet. Aus Gründen der Speicherökonomie werden die Schlüsselwerte in den Blättern bei nicht eindeutigen Indexen i.d.R. nur einmal abgelegt. Zu jedem Schlüsselwert wird dann eine Liste mit den Verweisen auf die den Schlüsselwert enthaltenden Datensätze gespeichert (*Abbildung 3.11*). Würde sich die zu einem Schlüsselwert gehörende Verweisliste über mehrere Blöcke auf der Blattebene

erstrecken, so wird in jedem der betroffenen Blätter der Schlüsselwert einmal mit einem jeweils eigenen Teil der Verweisliste gespeichert.

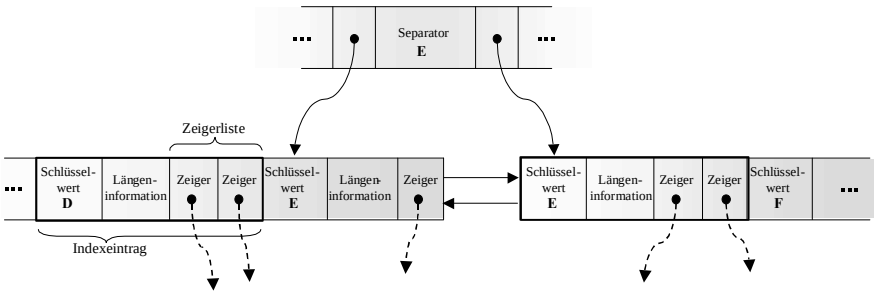


Abbildung 3.11: Indexeinträge eines Sekundärindex

Insbesondere durch Löschoperationen kommt es auch bei Indexknoten zu einer *Verschlechterung der Speicherausnutzung*. Aus Performance-Gründen werden Verschmelzungen von Indexknoten teilweise von DBMS nicht oder erst sehr spät durchgeführt. Daraus ergibt sich neben der Speicherplatzverschwendung (weniger als 50% des belegten Speichers werden auch tatsächlich genutzt) auch eine Erhöhung der Anzahl der zu lesenden Index-Knoten bei Index Range Scans. Die schlechte Speicherauslastung kann zu einer so stark überhöhten Anzahl Indexknoten führen, dass die *Höhe des Indexbaums* durch eine kompakte Speicherung um eine Ebene verringert werden könnte, was zu einer Reduzierung der für einen Datenzugriff notwendigen Blockzugriffe führt. Bei Einfügeoperationen haben nicht vollständig gefüllte Indexknoten allerdings wiederum den Vorteil, dass die Wahrscheinlichkeit sinkt, dass ein Aufspalten von Baumknoten notwendig wird. DBMS-Produkte bieten auch hier Steuerparameter an (z.B. Oracle mit dem Parameter `PCTFREE` bzw. `FILLFACTOR` bei Informix), mit denen festgelegt werden kann, welcher Speicherplatzanteil in den Indexblöcken beim Neuaufbau für spätere Einfügeoperationen frei gehalten werden soll. Wenn weiter viele Einfügeoperationen nach dem initialen Aufbau des Index zu erwarten sind, kann dadurch die Systemleistung für einen gewissen Zeitraum verbessert werden. Ist die Speicherplatzreserve erschöpft, muss der Index ggf. neu aufgebaut werden, wenn die Zahl notwendiger Splitting-Operationen auch weiterhin gering gehalten werden soll.

Das physische Löschen von Indexeinträgen erfolgt bei einigen DBMS-Produkten (z.B. DB2, Informix) asynchron. Den Verweisen auf die Datensätze wird meist noch ein Lösch-Flag zugeordnet. Zu löschende Indexeinträge werden über dieses Flag zunächst nur als „gelöscht“ markiert. Später, bspw. in lastarmen Zeiten, wird eine Wartung der Indexstrukturen vorgenommen, bei der u.a. als gelöscht gekennzeichnete Verweise bzw. Indexeinträge (*Ghost-Einträge*) entfernt werden [IBM02]. Ist eine automatisch ablaufende Wartung lange Zeit nicht möglich, z.B. weil der Index stark frequentiert wird, so sammelt sich eine große Anzahl von Ghost-Einträgen an, die den Index ebenfalls aufblähen.

3.3.3 Indexorganisierte Tabellen

Eine Verknüpfung von Daten- und Indexbereichen stellen **indexorganisierte Tabellen** (IOT) dar. Dabei werden die Daten in die Blattebene des Indexbaums integriert. Einfüge-, Änderungs- und Löschoperationen von Daten werden nach den für B*-Bäume üblichen Methoden vorgenommen. Dadurch sind die Tupel immer nach dem Indexierungsschlüssel sortiert. Deshalb werden solche Strukturen häufig auch als *geclusterte Indexe* (Clustered Index) bezeichnet. Die Sortierung ist insbesondere vorteilhaft bei Bereichsanfragen über dem Indexierungsschlüssel, bzw. wenn Daten häufig nach dem entsprechenden Schlüssel geordnet benötigt werden. Auch hier werden die einzelnen Blätter üblicherweise miteinander doppelt verkettet. Allerdings ist es durchaus wahrscheinlich, dass beim Blocksplitting in der Blattebene der neue Block nicht physisch unmittelbar hinter dem zu teilenden Block reserviert werden kann. Dies führt zu einer Erhöhung des Positionieraufwands der Lese-/Schreibköpfe. Um das Wachstum des Indexbaums zu verlangsamen, ist die Tupellänge entweder systemseitig begrenzt oder das DBMS (bspw. Oracle) bietet die Möglichkeit an, die Sätze zur Speicherung der Tupel aufzuteilen und seltener benötigte Teile in einen *Überlaufbereich* auszulagern (*Abbildung 3.12*). Durch Löschoperationen kommt es zur langsamen Verschlechterung der Speicherauslastung der Blattknoten. Neben dem höheren Speicherbedarf führt dies bei Bereichsanfragen zu einer Erhöhung der Anzahl zu durchsuchender Blöcke und in extremen Fällen zu einer höheren Ebenenzahl des Index, als eigentlich nötig.

Durch die notwendige Verkettung zwischen dem im Baum gespeicherten Teil eines Tupels und dem ausgelagerten Teil sind, wenn ein gesamtes Tupel gelesen werden soll, weitere Zugriffe für das Lesen des jeweiligen Überlaufblocks nötig. Daher sollte beim Anlegen von indexorganisierten Tabellen (durch den DBA) darauf geachtet werden, dass die Teile dem Überlaufbereich zugeordnet werden, auf die i.d.R. seltener zugegriffen wird. Die Vorgehensweise ähnelt der vertikalen Partitionierung von Tabellen. Werden bei Änderungsoperationen an den Tupeln Werte der Attribute geändert, die im primären Index, über den die IOT organisiert ist, enthalten sind, so müssen die Sätze physisch verschoben werden, um die Sortierreihenfolge aufrecht zu erhalten.

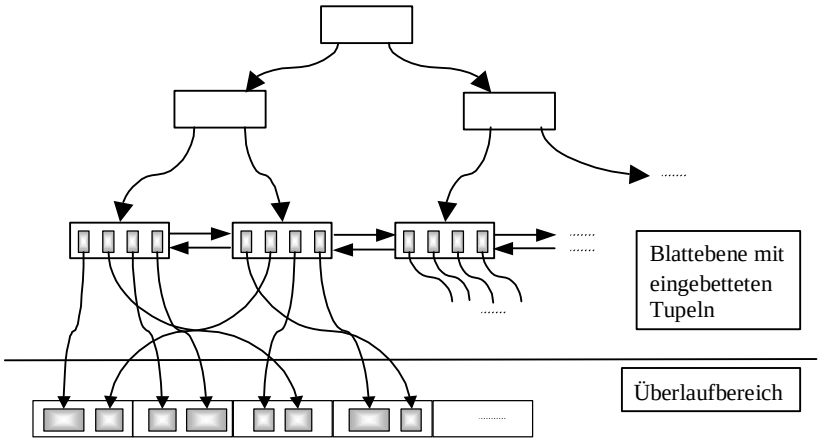


Abbildung 3.12: Indexorganisierte Tabelle mit Überlaufbereich

Auch beim Teilen von Blattknoten können sich die physischen Speicherorte von Sätzen durch die mit dem Blocksplitting verbundenen Umverteilungen ändern. Bezüglich der bisher betrachteten Adressierung von Datensätzen über physische Referenzen aus zusätzlichen Indexen heraus (TID, RID usw.) hieße dies, dass sich bei solchen Umverteilungen die Tupelidentifikatoren einer größeren Anzahl von Sätzen ändern würden. Das würde entweder zu einem sehr schnellen Anwachsen der Zahl migrierter Tupel oder zu einem erheblichen Aufwand für die Pflege der (anderen betroffenen) Indexe führen. Deshalb werden bei Sekundärindexen auf Tupel indexorganisierter Tabellen häufig logische Referenzen gespeichert (bspw. bei Adabas D bzw. MaxDB). Damit einher geht aber ein erhöhter Aufwand für das Auflösen der logischen Referenzen bei Datenzugriffen. Oracle verwendet hier sog. *logische ROWIDs*, die ihre Gültigkeit auch behalten, wenn ein Satz in der IOT verschoben wird. Eine solche logische ROWID beinhaltet aber auch die Nummer des Blocks, in dem der zugehörige Satz beim Aufbau des Index bzw. bei der letzten Indexwartung gespeichert war [Ora03a]. Damit kann auf Sätze, die seitdem nicht verschoben wurden, mit einem Blockzugriff (zusätzlich zu denen zum Durchsuchen des Sekundärindex) zugegriffen werden. Da im Laufe der Zeit durch die notwendigen Verschiebeoperationen immer mehr dieser Blocknummern ihre Gültigkeit verlieren, muss durch den DBA in gewissen Abständen eine Wartung des Index vorgenommen werden. Diese kann nach [Ora03a] im Online-Betrieb erfolgen.

3.3.4 Unterstützung von Verbundoperationen

Ein Konzept zur Unterstützung von Verbundoperationen ist die tabellenübergreifende Clusterung von Daten (vgl. Abschnitt 3.2.3). Zur Speicherung der Daten eines Clusters wird ein Segment angelegt. Im Unterschied zu Heap-Segmenten und Segmenten zur Datenspeicherung von indexorganisierten Tabellen können hier in einem Bucket (Block) Sätze mit verschiedenem Format (aus unterschiedlichen Tabellen) gespeichert werden. Über den Cluster-Schlüssel wird ein gemeinsamer Zugriffspfad angelegt. Dieser Zugriffspfad kann bei Oracle als B^*-Baum Index oder über eine Hash-Funktion realisiert sein. Im Index wird jeweils einem Wert des Cluster-Schlüssels die Adresse des Bucket zugeordnet, das die den Cluster-Schlüsselwert enthaltenden Sätze aufnimmt. Es werden keine Zuordnungen zu einzelnen Sätzen vorgenommen. Damit stellt der Zugriffspfad eine Ausprägung der dünn besetzten Indexe dar. Oracle verwendet kein erweiterbares Hashing. Wird der Zugriffspfad über eine Hash-Funktion realisiert, so muss beim Anlegen des Clusters angegeben werden, wie viele Buckets sofort im primären Bereich des Clusters angelegt werden sollen. Da Hash-Funktionen i.d.R. nicht eineindeutig sind, können in einem Bucket durchaus Sätze mit unterschiedlichen Cluster-Schlüsselwerten untergebracht werden. Deshalb wird der jeweilige Wert des Cluster-Schlüssels bei Hash-Clustern in den Datensätzen mit gespeichert.

Ist der Zugriffspfad als B^*-Baum Index organisiert, so wird jeweils beim Einfügen eines neuen Cluster-Schlüsselwerts ein neues Bucket angelegt [Sin00]. Da alle in einem Bucket gespeicherten Datensätze den gleichen Cluster-Schlüsselwert enthalten, wird dieser, um Speicher zu sparen, im Bucket nur einmal abgelegt [Ora03b]. Wächst

die Zahl der zu einem Cluster-Schlüsselwert gehörenden Tupel in Laufe der Zeit (stark) an, so reicht u.U. ein Datenblock zu deren Speicherung nicht mehr aus. In diesem Fall werden *Überlaufbereiche* angelegt. Das Verfahren ähnelt im Wesentlichen dem von Hash-Tabellen her bekannten *Separate Chaining*, bei dem je Bucket eine separate Überlaufkette angelegt wird. Durch das Durchsuchen der Überlaufketten steigt der Aufwand für Datenzugriffe (*Abbildung 3.13*), der im Idealfall, neben den für das Durchsuchen des Index notwendigen Zugriffen, nur einen Zugriff auf Datenblöcke betragen sollte. Überlaufbereiche können durch eine Reorganisation beseitigt bzw. verkleinert werden, wenn diese durch das Löschen von Tupeln nur noch teilweise gefüllt sind oder wenn bei einer Reorganisation die Bucket-Größe erhöht werden kann.

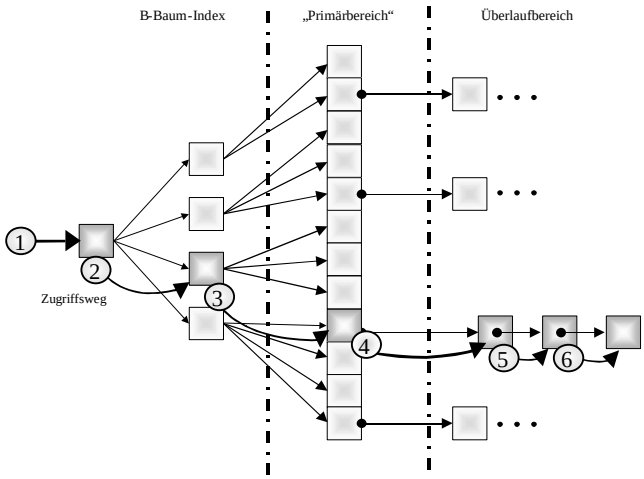


Abbildung 3.13: Datenzugriff bei einem Index-Cluster

Zusätzlich zum Cluster-Index können (wie üblich) noch weitere Indexe (zusätzliche Indexe) erzeugt werden, die sich auf einzelne im Cluster gespeicherte Tabellen beziehen. Bei der Indexierung wird hier wie bei Heap-Tabellen vorgegangen. In den Blättern der Indexbäume werden physische Verweise auf die Datensätze gespeichert. Durch satzverlängernde Änderungsoperationen kann es auch bei Clustern zum Verschieben der Sätze in andere Datenblöcke, der dem Bucket zugeordneten Überlaufkette kommen [Sin00].

Zusätzlich zum Cluster-Index können noch weitere Indexe (zusätzliche Indexe) erzeugt werden, die sich auf einzelne im Cluster gespeicherte Tabellen beziehen. Bei der Indexierung wird hier wie bei Heap-Tabellen vorgegangen. In den Blättern der Indexbäume werden physische Verweise auf die Datensätze gespeichert. Durch satzverlängernde Änderungsoperationen kann es auch bei Clustern zum Verschieben der Sätze in andere Datenblöcke, der dem Bucket zugeordneten Überlaufkette kommen [Sin00].

Dies führt bei Zugriffen über zusätzliche Indexe, wie bei Heap-Tabellen auch, neben der Entstehung von eingestreutem Freiplatz, zu einer Erhöhung des Suchaufwands durch die Verfolgung der Auslagerungszeiger. Ändern sich bei Tupeln die Werte von Attributen, über denen der Cluster-Schlüssel definiert ist, so müssen die zugehörigen Sätze physisch verschoben werden, damit die Cluster-Eigenschaft erhalten bleibt. Um

auch hier das Nachpflegen eventuell vorhandener zusätzlicher Indexe zu vermeiden, wird ebenfalls mit Auslagerungszeigern gearbeitet, die am ursprünglichen Speicherort abgelegt werden. Bei Zugriffen über den Cluster-Schlüssel existiert kein direkter Zusammenhang zwischen der Zahl der migrierten Tupel und dem Suchaufwand. Eine Erhöhung des Suchaufwands ergibt sich aber indirekt durch die mit den Auslagerungen (durch die Entstehung eingestreuten Freiplatzes) oftmals verbundene Verlängerung der jeweiligen Überlaufketten. Diese Verlängerung der Überlaufketten führt zu einer Erhöhung der Anzahl Datenblöcke, die der Cluster insgesamt belegt, und damit auch zu einer Erhöhung des Aufwands für sequenzielles Durchsuchen einzelner Tabellen des Clusters. Die Verwendung von Clustern ist aus praktischer Sicht zu empfehlen wenn:

- die Cluster-Schlüsselwerte der einzelnen Tupel nur selten geändert werden,
- die Daten der im Cluster gespeicherten Tabellen oft gemeinsam (über den Cluster-Schlüssel verbunden) benötigt werden,
- die im Cluster gespeicherten Tabellen wenig wachsen,
- die Cluster-Gruppen (sieh e Abschnitt 3.2.3) möglichst annähernd gleiche Größen haben und
- die einzelnen Tabellen des Clusters selten sequenziell durchsucht werden.

Eine *verallgemeinerte Zugriffspfadstruktur* (Generalized Access Path) zur Unterstützung von Verbundoperationen zwischen hierarchisch in Beziehung stehenden Datenbankobjekten bzw. zur Überprüfung referenzieller Integritäten, die ebenfalls auf B*-Baum-Variationen basiert, wird in [HR01] vorgeschlagen. Üblicherweise werden auf der Blattebene eines Index nur Verweise auf Datensätze *einer* Tabelle gespeichert. Das heißt, ein Index wird genau einer Tabelle zugeordnet. Bei der verallgemeinerten Zugriffspfadstruktur wird ein Index mehreren, typischerweise in hierarchischer Beziehung zueinander stehenden Tabellen zugeordnet. Einem Schlüsselwert im Index wird jeweils ein Verweis auf den Datensatz des in der Hierarchie übergeordneten Objekts und eine Liste mit Verweisen auf die Datensätze untergeordneter Objekte gespeichert. *Abbildung 3.14* zeigt ein Beispiel, für die Verknüpfung von Mitarbeiterdaten mit den Daten der Abteilungen, denen sie zugeordnet sind.

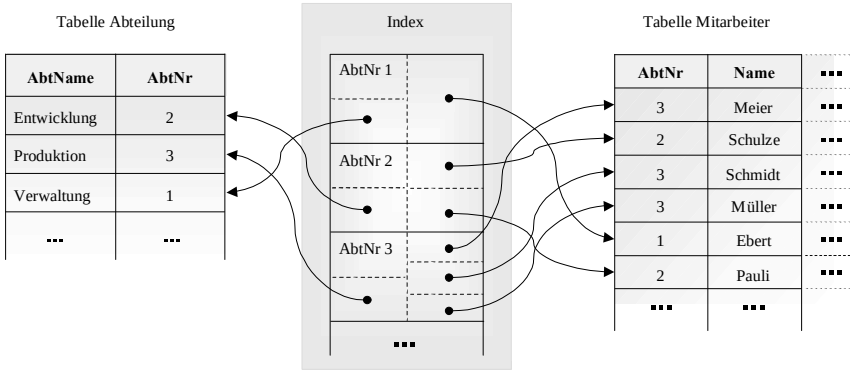


Abbildung 3.14: Verallgemeinerte Zugriffspfadstruktur zur Verknüpfung von Mitarbeiter- und Abteilungsdaten

Der Index ist hier nur schematisch dargestellt. Er kann beispielsweise als B^*-Baum realisiert und als Primärzugriffspfad für den Zugriff auf die Sätze der übergeordneten Objekte, als Sekundärzugriffspfad für den Zugriff auf die Daten der in der Hierarchie untergeordneten Objekte und als gemeinsamer Zugriffspfad (bspw. zur Unterstützung von Verbundoperationen) verwendet werden. Zu beachten ist, dass der Generalized Access Path wegen seiner Größe natürlich nicht immer optimal ist (bspw. beim Primärzugriff gegenüber einem reinen Primärindex). Der Index kann unabhängig von der Organisation der Datenbereiche und von anderen Indexen aufgebaut werden. Eine bestimmte physische Anordnung der Daten wird nicht erzwungen und muss somit auch nicht, wie beim oben betrachteten Cluster, gepflegt werden. Die Daten der über- und der untergeordneten Tabellen können, wie bei relationalen DBMS üblich, in physisch getrennten Bereichen gespeichert werden. Verbundoperationen werden, wie beim Cluster auch, durch den gemeinsamen Zugriffspfad auf die Daten der am Verbund beteiligten Tabellen unterstützt.

Der Aufwand für die reinen Datenzugriffe ist durch deren physisch getrennte Speicherung allerdings i.d.R. höher als beim Cluster. Durch weitere Verallgemeinerungen kann das Einsatzgebiet noch erweitert werden. So sind mehrere Verweislisten auf untergeordnete Objekte verschiedenen Typs, die in verschiedenen Tabellen gespeichert sind, denkbar. Werden für alle Tabellen Verweislisten verwendet, so können auch Verbundoperationen über in allen Tabellen enthaltene Schlüssel unterstützt werden.

3.4 Unterstützung komplexer Objekte

In einigen Anwendungsbereichen (z.B. im wissenschaftlich-technischen Bereich) mit großen Mengen komplex strukturierter Daten führt die Beschränkung auf Relationen mit atomaren Attributen oftmals dazu, dass logisch eng verknüpfte Daten, wie z.B. die eines komplexen Objekts, bei rein relationalen DBMS auf mehrere Tabellen verteilt werden müssen. Das führt u.U. zu einem erheblichen Aufwand bei der Rekonstruktion der Objekte. Um Anwendungen mit komplex strukturierten Daten besser unterstützen zu können, sind die Anbieter relationaler DBMS teilweise (bspw. Oracle bzw. IBM mit DB2 und Informix) dazu übergegangen, ihre Systeme um objektorientierte Konzepte zu erweitern und damit (zumindest ansatzweise) objektrelationale Datenbank-Management-Systeme (ORDBMS) anzubieten. Diese Systeme lassen die Modellierung komplexer Objekte ohne strenge Einhaltung der ersten Normalform für Relationen zu. Vorschläge für zukünftige Erweiterungen der SQL-Norm um zusätzliche Kollektionstypen werden in [Luf05] unterbreitet.

3.4.1 Abbildungsmöglichkeiten auf physische Speicherungsstrukturen

Objektrelationale Datenbank-Management-Systeme müssen die Verwaltung relational repräsentierter Daten in flachen Tabellen und die Speicherung und Verarbeitung komplexer und geschachtelter Datenobjekte parallel ermöglichen. Eine zentrale Anforderung ist dabei, dass die Integration neuer objektrelationaler Konstrukte und die neu gewonnene Erweiterbarkeit des DBMS die Performance bei der Verarbeitung

der bisherigen flachen Tabellen nicht beeinträchtigt. Durch die Beschränkung der Erweiterungen und Änderungen auf die oberen Architekturschichten Datensystem und Zugriffssystem und auf die Anfrageübersetzung ist dieses Ziel erreichbar [CCN+99, HR01]. Die Komponenten zur Puffer-, Block- und physischen Satzverwaltung können weitgehend unverändert bleiben. In [KLS02] wird eine Transformationsschicht beschrieben, die objektrelationale Sprachkonstrukte auf die Strukturen existierender DBMS abbildet. Die von den Systemen selbst angebotenen Möglichkeiten zur Anwendung von objektorientierten Konzepten werden intern ebenfalls oft auf bisherige relationale Strukturen abgebildet und die Anwender haben meist nur wenig Einfluss auf die physische Abbildung [Ska06]. Damit wird zwar zunächst das im Zusammenhang mit relationalen Systemen und Anwendungserfordernissen oftmals genannte Problem des *impedance mismatch* abgemildert, das Problem der aufwendigen Rekonstruktion komplexer Objekte aus den in flacher relationaler Form gespeicherten Daten bleibt jedoch bestehen. Es ist wünschenswert, auch die Abbildung komplexer Objekte auf die vorhandenen physischen Speicherungsstrukturen der logischen Strukturierung der Objekte entsprechend gestalten zu können. In mehreren Arbeiten (z.B. [Keß95, Mea97, Ska02, Ska06]) werden dazu verschiedene Abbildungsmöglichkeiten auf interne Satzstrukturen und Clusterungs-Strategien für komplexe Objekte untersucht und beschrieben. Darüber hinaus werden dort Sprachkonstrukte definiert (bspw. Physical Representation Definition Language – PRDL [Kis02, Ska06]), die Anwendern Einflussmöglichkeiten auf die physische Speicherungsstruktur geben.

Dabei werden als Freiheitsgrade

- die Aufteilung von Objekten auf mehrere physische Sätze,
- seitenüberspannende Sätze,
- der Speicherort physischer Sätze,
- Typen von Objektreferenzen sowie
- die physische Kollektionsstruktur und
- Clusterungs-Strategien

genannt.

Unterschiedliche Möglichkeiten der Abbildung von komplexen Objekten sollen am vereinfachten Beispiel eines Telekommunikationsdienstes dargestellt werden, auf das auch in den Erläuterungen in Kapitel 9 Bezug genommen wird. Der Telekommunikationsdienst „Universal Number“ bietet sog. Dienstkunden (z.B. Pannendienste, Versicherungen etc.) die Möglichkeit, unter einer landesweit einheitlichen („virtuellen“) Rufnummer (z.B. 0180-189765) erreichbar zu sein [Dor97]. Die einzelnen Anrufe sollen aber nicht in einem zentralen Call-Center bearbeitet werden, sondern von den Vermittlungsanlagen direkt an die den Nutzern der Dienste jeweils am nächsten liegende Niederlassung des Dienstkunden weitergeleitet werden. Hauptaufgabe des sog. Verkehrsführungsprogramms, das den Dienst realisiert, ist die Umsetzung der einheitlichen virtuellen Rufnummer in eine

zugehörige reale Rufnummer. Eine entsprechende Modellierung der Daten für diesen Dienst zeigt *Abbildung 3.15* in Form eines Objektdiagramms.

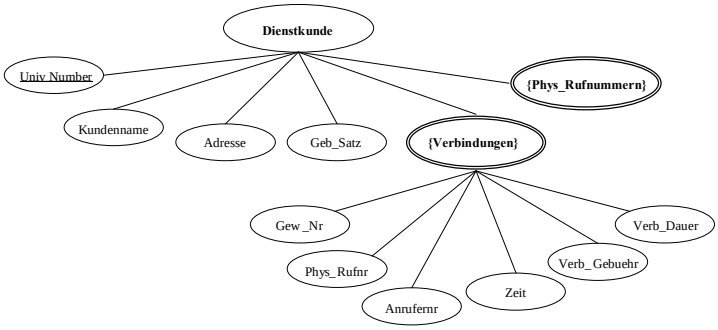


Abbildung 3.15: Objektdiagramm des Beispiels

Neben den üblichen Daten wie Kundenname, Adresse und Gebührensatz werden unseren Annahmen zufolge im Objekt `Dienstkunde` im Kollektionsattribut `Verbindungen` noch die Daten aller über das Telekommunikationsunternehmen abgewickelten Verbindungen gespeichert. Über die Telefonnummer des Anrufers (`Anrufernr`) und die Anrufzeit werden die einzelnen Verbindungen identifiziert. Weiterhin werden die vom Anrufer gewählte Nummer, die reale („physische“) Rufnummer, zu der der Anruf weitergeleitet wurde, die Verbindungsdauer sowie der tatsächlich angefallene Gebührensatz gespeichert. Im mengenwertigen Attribut `Phys_Rufnummern` werden den jeweiligen virtuellen die realen Rufnummern zugeordnet. Das auf der Datenbank arbeitende Verkehrsführungsprogramm für den Dienst „Universal Number“ bestimmt anhand der gewählten Nummer (z.B. 0180-189765) den entsprechenden Dienstkunden und eine passende verfügbare reale Rufnummer, mit der der Anrufer verbunden wird. Nach dem Anruf wird dieser im Kollektionsattribut `Verbindungen` verzeichnet.

Für eine zunächst rein relationale Umsetzung des Beispiels erfolgt die Darstellung in einem E/R-Diagramm ohne Nutzung erweiterter Konzepte wie mengenwertige und strukturierte Attribute (*Abbildung 3.16*).

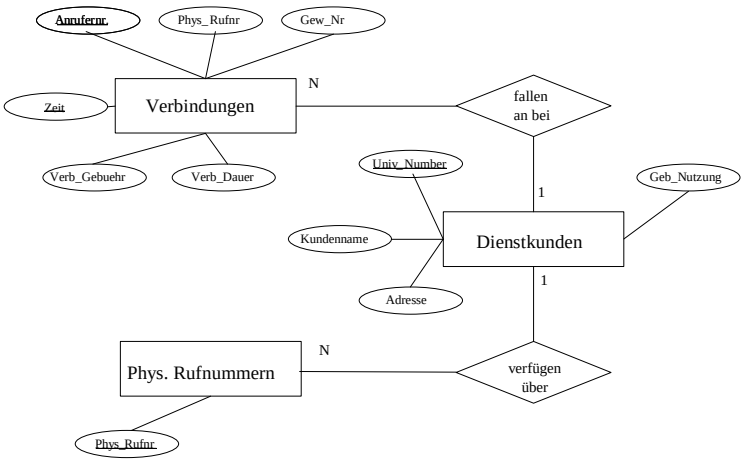


Abbildung 3.16: Informationsschema des Beispiels

Nach den üblichen Transformationsregeln ergibt sich daraus etwa das in *Abbildung 3.17* dargestellte relationale Schema. Primärschlüsselattribute sind unterstrichen und Fremdschlüsselattribute kursiv dargestellt. Das Attribut `Gew_Nr` enthält die vom Anrufer gewählte Nummer, die der `Univ_Number` des Dienstkunden entspricht.

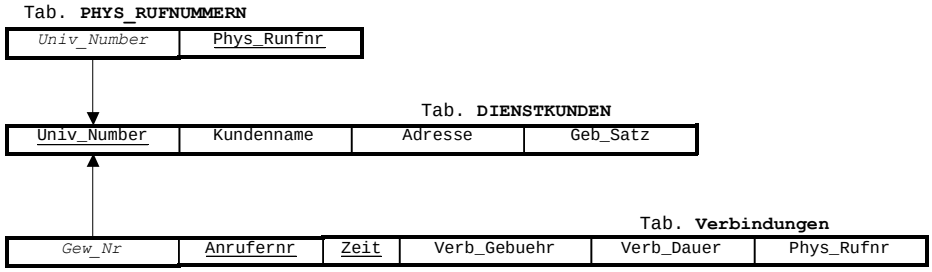


Abbildung 3.17: Relationales Schema für das Beispiel „Universal Number“

Die Abbildung der drei Tabellen kann auf die in *Abschnitt 3.3* beschriebenen Strukturen erfolgen.

Bei der Speicherung der Daten dieses Umweltausschnitts als komplexes Objekt in einer objektrelationalen Datenbank sind hingegen verschiedene physische Repräsentationen möglich.

Die Daten komplexer Objekte werden prinzipiell in Satzstrukturen gespeichert, die im Unterschied zu rein relationalen Implementierungen allerdings beliebig geschachtelt werden können. Die einzelnen Komponenten eines Satztyps, die selbst wieder Sätze sein können, werden entweder in die Satzstruktur *eingebettet* oder aus ihr heraus *referenziert* (*Abbildung 3.18*). Bei einer Referenzierung von Komponenten werden deren Daten physisch entfernt (ausgelagert) von der Satzstruktur des umfassenden Objekts (Hauptobjekt) gespeichert, die lediglich einen Verweis auf den Speicherort der Daten der jeweiligen Komponente (Subobjekt) enthält.

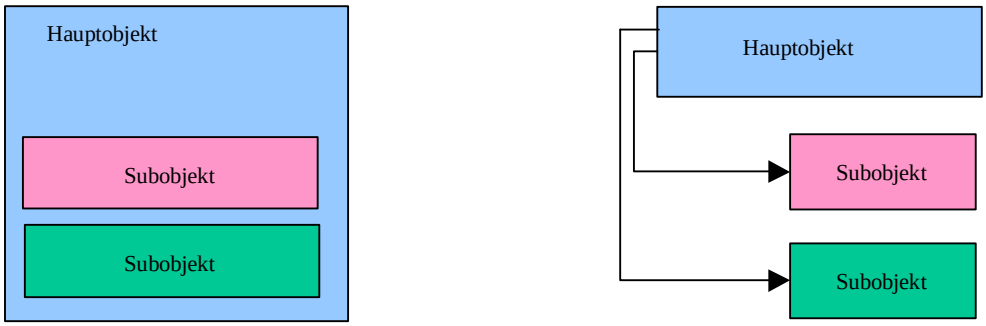


Abbildung 3.18: Eingebettete und referenzierte Speicherung komplexer Objekte

Im betrachteten Beispiel bietet sich besonders eine Auslagerung der Verbindungsdaten an, da die Anzahl anfallender Verbindungen jeweils sehr unterschiedlich und schwer abschätzbar ist und hier mit einer starken Zunahme des Umfangs der zu speichernden Daten gerechnet werden muss. Die Entscheidung, ob die Menge der realen Rufnummern, in die die Daten der jeweiligen Dienstkunden enthaltenden Sätze eingebettet gespeichert werden kann oder nicht, ist u.a. davon

abhängig, wie stark die Anzahl der Rufnummern schwankt und ob eine Obergrenze existiert. Ob physische Sätze auf die Größe eines Datenblocks beschränkt sind oder ob diese mehrere Blöcke *überspannen* dürfen muss hier evtl. ebenfalls berücksichtigt werden.

Neben der Festlegung der physischen Satzstruktur kann für die einzelnen Satztypen komplexer Objekte über die Angabe des jeweiligen Table Space der physische *Speicherort* auf den Sekundärspeichermedien grob festgelegt werden.

Referenzen auf Objekte können die physische Adresse des Speicherorts enthalten oder über logische Werte realisiert werden. Bei der Verwendung logischer Werte müssen diese mit Hilfe einer Indexstruktur in physische Adressen umgesetzt werden. Auch Mischformen, bestehend aus logischen Werten mit Hinweisen auf physische Speicherorte, sind möglich [Ska02].

Mehrere Sätze eines Typs können fortlaufend in Heap-, verketteten Listen-, Array- oder Baum-Strukturen abgelegt werden. Bei verketteten Listenstrukturen werden die Speicherbereiche, die die einzelnen Sätze belegen, durch Zeiger miteinander verbunden. Dies ermöglicht eine effiziente Zusammenfassung auch dann, wenn die einzelnen Sätze unterschiedliche Größen haben können oder wenn sich die Größe von Sätzen durch Update-Operationen u.U. erheblich ändern kann. Zur einfachen Anwendung der bei Arrays i.d.R. üblichen Methode der indexierten Adressierung muss sichergestellt werden, dass alle Sätze die gleiche Größe besitzen. Die einzelnen Sätze werden dann nacheinander in den einzelnen Elementen des Array abgelegt. Um den Speicherplatz für das Array vorab reservieren zu können und damit eine eingebettete Speicherung zu ermöglichen, muss auch die (maximale) Anzahl der Elemente bekannt sein. Durch die Verwendung der beschriebenen Strukturen als Satzkomponenten können verschiedene Arten von *Kollektionen* (wie in [Luf05] beschrieben) abgebildet werden. Die eingebettete Speicherung der Menge der realen Rufnummern könnte bspw. über ein Array erfolgen.

3.4.2 Clusterung der Daten komplexer Objekte

Zwei wichtige *Strategien zur Clusterung* von Sätzen sind die objektbezogene Cluster-Bildung und die objektübergreifende Cluster-Bildung. Bei der objektübergreifenden Cluster-Bildung werden physische Datensätze des gleichen Typs (der gleichen Tabelle) möglichst dicht gespeichert. Diese Speicherungsform entspricht dabei im Wesentlichen der bei relationalen DBMS üblichen Speicherungsform. In Tabellen gespeicherte Fremdschlüsselwerte können dabei als eine Art logischer Referenzen angesehen werden. *Abbildung 3.19* zeigt diese Form der Cluster-Bildung bezogen auf das Beispiel des Telekommunikationsdienstes. Die einzelnen Cluster sind dabei jeweils grau hinterlegt.

Müssen bspw. die Daten aller Dienstkunden oftmals gemeinsam, unabhängig von den realen Rufnummern, verarbeitet werden, so bewirkt die Auslagerung der realen Rufnummern eine Verringerung der zu verarbeitenden Datenmenge. Die physisch beieinanderliegende Speicherung der Sätze aller Dienstkunden führt weiterhin zu einer Verringerung der Positionierungsoperationen der Lese-/Schreibköpfe beim

sequenziellen Suchen. Unterscheidet sich die Zahl der realen Rufnummern bei den verschiedenen Dienstkunden stark und ist deren maximale Zahl je Dienstkunde ebenfalls nicht absehbar, so ist die von den übrigen Daten der Dienstkunden getrennte Speicherung auch einfacher zu realisieren, als die eingebettete Speicherung der realen Rufnummern.

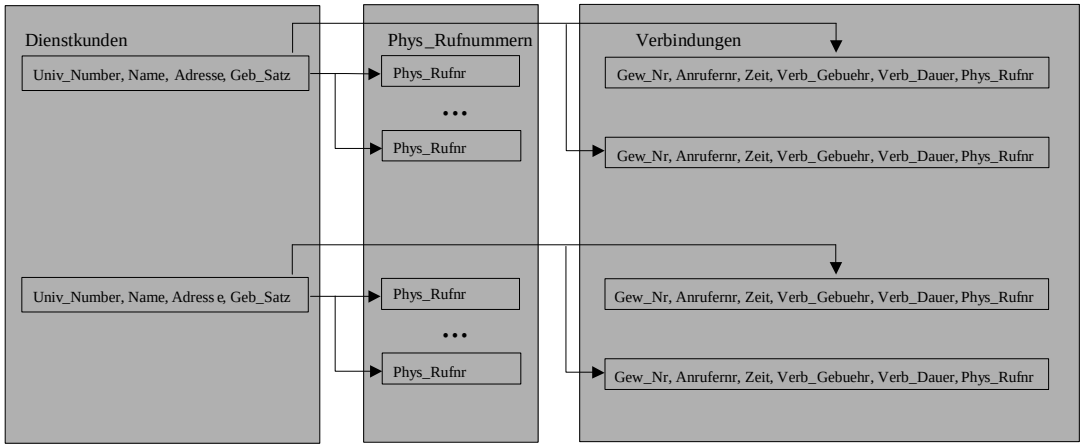


Abbildung 3.19: Objektübergreifende Cluster-Bildung

Werden die Daten der Dienstkunden und die zugehörigen realen Rufnummern allerdings zusammen benötigt, wie dies bspw. bei der Vermittlung von Anrufen der Fall ist, so verursacht die physische getrennte Speicherung Aufwand für die Verknüpfung der Daten. Hier bietet sich die objektbezogene Cluster-Bildung als wesentliche Möglichkeit objektrelationaler DBMS an (komplexe Objekte auf Modell- und *Speicherungsebene*). Dabei werden die physischen Sätze, die zu einem komplexen Objekt gehören, dicht gespeichert, obwohl unterschiedliche Satztypen vorliegen. Alle Sätze von Subobjekten (auch Mitgliedssätze oder sekundäre Sätze) werden in unmittelbarer Nähe des Satzes des jeweils umfassenden Objekts (auch Primärsatz) gespeichert. *Abbildung 3.20* zeigt die objektbezogene Cluster-Bildung am Beispiel des Telekommunikationsdienstes.

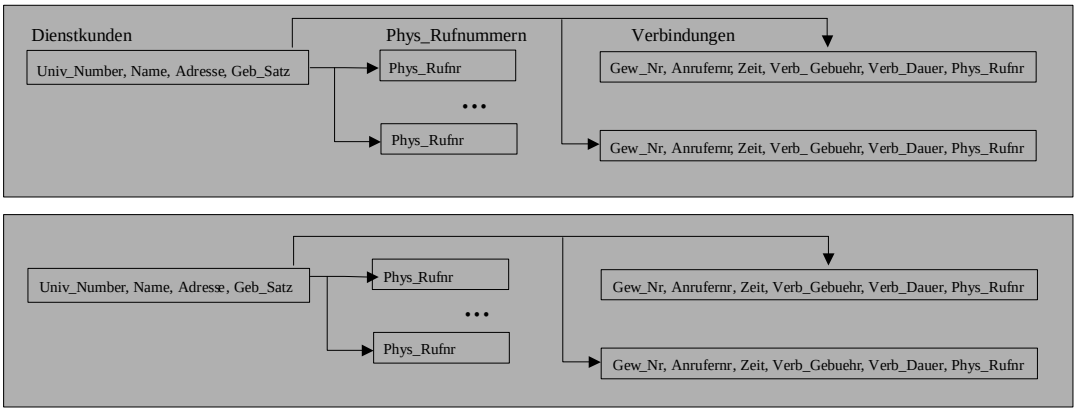


Abbildung 3.20: Objektbezogene Cluster-Bildung

Mit der von Oracle implementierten Cluster-Struktur kann diese Form der Cluster-Bildung realisiert werden.

Die verallgemeinerte Zugriffspfadstruktur und die Cluster-Struktur von Oracle (vgl. Abschnitte 3.2.3 und 3.3.4) unterstützen eine hierarchische Clusterung von Datenbankobjekten auf zwei Ebenen. In [Zou96, ZSL98] wird unter dem Begriff *Dynamic Hierarchical Clustering* eine Methode vorgeschlagen, die eine Clusterung der Sätze hierarchisch strukturierter Datenobjekte ermöglicht, die aus mehr als zwei Ebenen bestehen. Als Speicherungsstruktur werden B^*-Baum-Variationen verwendet, bei denen die Daten komplexer Objekte, inklusive der Daten von Subobjekten, auf der Blattebene gespeichert werden. Dadurch bleibt die physisch beieinanderliegende Speicherung der Daten komplexer Objekte auch bei der Anwendung von Lösch- und Einfügeoperationen erhalten. Um die Clusterung zu erreichen, werden zusammengesetzte (Pfad)Schlüssel gebildet, über denen der Indexbaum aufgebaut wird. Beispielsweise kann für Mitarbeiter, die in den Niederlassungen von Unternehmen beschäftigt sind, ein Schlüssel (*Look Up Key*) aus der Kombination `Unternehmen/Niederlassung/Mitarbeiter` gebildet werden. Zur Reduzierung der Größe der Schlüsselwerte wird eine als *„Enc“* bezeichnete numerische Codierung vorgenommen. Auf der obersten Ebene werden die Objekte (bspw. die Unternehmen) in der Reihenfolge ihres Einfügens nummeriert. Die zu einem Objekt gehörenden Subobjekte (hier bspw. die Niederlassungen) werden wiederum in der Einfügereihenfolge nummeriert. Diese Nummern werden als C*hild Number* bezeichnet. Die Nummerierung wird bis zur unteren Ebene (hier Mitarbeiter) fortgesetzt. Würden beispielsweise die in *Tabelle 3.2* aufgeführten Daten von Firmen, zugehörigen Niederlassungen und deren Mitarbeitern eingefügt, so würden sie wie ebenfalls in Tabelle 3.2 zusätzlich angegeben codiert und wie in *Abbildung 3.21* dargestellt abgespeichert werden. Die in Abbildung 3.21 unten rechts angedeutete Slot-Liste spiegelt auch die Einfügereihenfolge wieder. Dabei wird angenommen, dass die Slot-Liste vom Ende her in den Block „hineinwächst“.

Firma (Code)	**Niederlassung (Code)**	**Mitarbeiter (Code)**
Meier AG (1)	Berlin (1-1)	Müller (1-1-1)
		Schulze (1-1-2)
	München (1-2)	Franz (1-2-1)
Schmidt GmbH (2)	Jena (2-1)	

Tabelle 3.2: Beispieldaten für Dynamic Hierarchical Clustering

Den Datensätzen der Objekte werden intern noch die Look Up Keys hinzugefügt. Zusätzlich wird bei den Objekten, die Subobjekte besitzen können, noch der nächste zu vergebende Child Key (in *Abbildung 3.21* in Klammern dargestellt) gespeichert. Bei Einfügeoperationen erfolgt eine Top-Down-Suche. Wurde das dem einzufügenden Objekt übergeordnete Objekt gefunden, wird aus dessen Look Up Key und dem nächsten zu vergebenden Child Key der Look Up Key des einzufügenden Objekts gebildet und der Datensatz für das Objekt eingefügt.

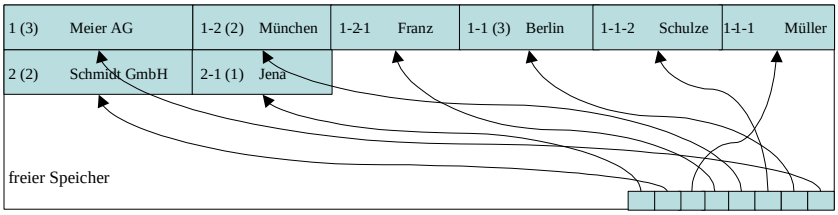


Abbildung 3.21: Datenspeicherung beim Dynamic Hierarchical Clustering (nach [Zou96])

Der Speicherbedarf eines Objekts und seiner Subobjekte kann den in einem Block auf der Blattebene verfügbaren Speicher übersteigen. Bei den für B-Bäume üblichen Einfügeverfahren würden die Daten neuer Subobjekte nach einer Splitting-Operation auf der Blattebene aber nicht mehr zwingend in den Block eingefügt werden, der auch die Daten des übergeordneten Objekts enthält. Dies würde dazu führen, dass beim Einfügen neben dem Block, der das übergeordnete Objekt enthält, noch ein weiterer Block, in den das Subobjekt eingefügt wird, geändert werden muss. Die Änderung des das übergeordnete Objekt enthaltenden Blocks ist zur Änderung des nächsten zu vergebenden Child Key notwendig. Durch eine leichte Modifikation der Vergleichsoperation beim Einfügen wird dafür gesorgt, dass neue Subobjekte immer in den Block eingefügt werden, der auch die Daten des übergeordneten Objekts enthält. Somit muss bei Einfügeoperationen, außer wenn aktuell ein Block-Splitting nötig wird, nur ein Block geändert werden.

Verallgemeinerte Zugriffspfade sowie die eben beschriebenen Speicherungsstruktur zur Cluster-Bildung werden nach dem derzeitigen Kenntnisstand von am Markt verfügbaren und im kommerziellen Einsatz verbreiteten Systemen nicht zur Verfügung gestellt. Allerdings könnten die vorgeschlagenen Konzepte durch relativ geringe Änderungen an vorhandenen Strukturen im Rahmen der Weiterentwicklung von (objektrelationalen) DBMS-Produkten umgesetzt werden. Weitergehende Probleme, etwa die adäquate Berücksichtigung bei der Anfrageoptimierung, Synchronisationsverfahren etc. seien hierbei in der Bewertung erst einmal außer Acht gelassen.

Bei der Festlegung der physischen Repräsentation komplexer Objekte ist es wichtig, dass diese die Abarbeitung der gegen die Objekte gerichteten Workload möglichst gut unterstützt. In [SD03] wird ein Ansatz zur Bewertung alternativer physischer Speicherrepräsentationen vorgestellt. Basis dafür ist die Quantifizierung des zur Workload-Abarbeitung anfallenden Aufwands in konkreten Systemumgebungen. Erläuterungen dazu enthält auch *Kapitel 9* dieser Arbeit.

4 Stand von Forschung und Entwicklung

Dieses Kapitel bietet einen Überblick über den gegenwärtigen Stand von Forschung und Entwicklung anhand von Publikationen und einer durchgeführten Produktstudie. Dabei werden Beispiele für Reorganisationen auf der internen Ebene und der Definitionsebene sowie für die in *Abschnitt 2.6* genannten Reorganisationsmethoden behandelt. Das Thema Reorganisation von physischen Speicherungsstrukturen, egal ob im Zusammenhang mit Datenbanken oder auch Dateisystemen, wurde in der Forschung in den vergangenen Jahren immer wieder mit unterschiedlicher Intensität und Zielstellung behandelt. Einige Beiträge entstanden dabei nicht direkt im Kontext von Datenbank-Management-Systemen, sondern im Zusammenhang mit der Entwicklung von Dateisystemen. Allerdings lassen sich die Ergebnisse und Methoden oft auch auf Datenbanken übertragen. Zahlreiche Beiträge in der Literatur, die im Zusammenhang mit dem Thema Reorganisation von Datenbanken oder Dateien bzw. Dateisystemen stehen, lassen sich in folgende, größtenteils disjunkte Gruppen gliedern:

- Die erste Gruppe von Beiträgen befasst sich mit dem Begriff Datenbankreorganisation und dem Herausarbeiten von Ebenen, auf denen Datenbankreorganisationen ansetzen können [NF76, SG79]. Diese Themen wurden bereits in Kapitel 2 der vorliegenden Arbeit behandelt.
- Eine weitere Gruppe der Beiträge und Bücher befasst sich damit, physische Speicherungsstrukturen und Operationen über diesen Strukturen formal zu beschreiben [BG82, EN02, HR01, SHS05]. Erste Ausführungen hierzu finden sich in Kapitel 3. In Kapitel 5 wird darauf aufbauend ein verallgemeinertes Speicher- und Verhaltensmodell (eInformationsschema genannt) entwickelt.
- Eine Gruppe von Beiträgen befasst sich mit Methoden zur Reorganisation von physischen Strukturen zur Speicherung von Daten und Indexen. Dabei liegen die Schwerpunkte auf Methoden zur Online-Reorganisation und auf Methoden, die parallel zum laufenden Betrieb quasi eine permanente Reorganisation des Datenbestands vornehmen. In [Wan00] wird z.B. ein System beschrieben, dass das Zugriffsverhalten von Anwendungsprozessen überwacht und ggf. zur Reduzierung von I/O-Aufwand eine Neuanordnung von Datenobjekten veranlasst.
- Die Bestimmung von Reorganisationszeitpunkten sowie die Festlegung von Reorganisationsintervallen unter Berücksichtigung von Kostenmodellen behandelt ebenfalls eine Gruppe von Arbeiten.
- Die nächste Gruppe bilden Beiträge, die sich mit Möglichkeiten auseinander setzen, durch die Verwendung geeigneter Speicherungsstrukturen und -konzepte auf der Definitionsebene den Aufwand für die Abarbeitung einer erwarteten Workload möglichst gering zu halten. Dabei werden auch teilweise Möglichkeiten zur Konvertierung der Strukturen beschrieben.
- Eine weitere Gruppe bilden Beiträge, die sich mit dem Thema Datenbankreorganisation aus praktischer Sicht (Einsatz der Reorganisation)

auseinander setzen und dabei durchaus auch auf spezielle Eigenschaften bestimmter DBMS-Produkte oder Anwendungen eingehen.

- Die letzte hier betrachtete Gruppe bilden Beiträge, die sich mit der Umstellung von Datenbankschemata (Restrukturierung, bzw. Schemaevolution) befassen. Dieses Thema wird in der Literatur [Ast02, KR04, MM87, NF76, Nav80, SC99] ausführlich behandelt und soll, wie bereits in Abschnitt 2.4 erwähnt, hier nicht näher betrachtet werden, da sich der Schwerpunkt der dortigen Betrachtungen wesentlich von dem der vorliegenden Arbeit unterscheidet.

Einige der Beiträge der Gruppen drei bis sechs werden in den folgenden Abschnitten exemplarisch näher betrachtet.

4.1 Methoden zur Reorganisation bei gleichzeitiger Nutzung

Offline durchzuführende Wartungsarbeiten, wie beispielsweise auch Datenbankreorganisationen, sind mit erheblichen Einschränkungen in der Verfügbarkeit der Daten verbunden. Deshalb lag und liegt ein Schwerpunkt in der Forschung auf der Entwicklung und Verbesserung von Methoden zur *Online-Reorganisation* von Datenbanken. In **[LS96]** findet sich ein Überblick über Arbeiten aus dem Umfeld von Gary H. Sockut, Edward Omiecinski, Betty Salzberg und Chendong Zou, die sich intensiv mit diesem Thema befassen.

4.1.1 In-Place-Reorganisation von indexorganisierten Tabellen

In **[ZS96a, ZS96c]** wird eine Methode zur *Online-Reorganisation von ausgedünnten B^+-Bäumen* mit in die Blattebene eingebetteten Daten[6] beschrieben. Das Verfahren beseitigt eingestreuten Freiplatz und ordnet die Blöcke auf der Blattebene auch physisch entsprechend der logischen Ordnung des Index. Die Reorganisation erfolgt in drei Phasen und wird *teilweise am Ort* (als In-Place-Reorganisation) und teils unter Verwendung bisher noch nicht genutzter Blöcke durchgeführt.

- In der ersten Phase erfolgt eine Verdichtung der Speicherung auf der Blattebene. Dazu werden entweder die Daten aus mehreren wenig gefüllten und entsprechend der Sortierreihenfolge des Index aufeinanderfolgenden Blöcken in einem Block zusammengeführt (*Abbildung 4.1*), oder es wird ein bisher unbenutzter Block als Zielblock verwendet (*Abbildung 4.2*), wenn sich in physischer Speicherreihenfolge zwischen zwei benachbarten Blattknoten ein freier Block befindet. Damit wird versucht, die physische Speicherreihenfolge der die Blattknoten enthaltenden Blöcke der logischen Reihenfolge im Index anzupassen. Dies dient der Reduzierung des Aufwands in der zweiten Phase der Reorganisation (s.u.).

[6] Indexorganisierte Tabellen entsprechen im Wesentlichen dieser Struktur.

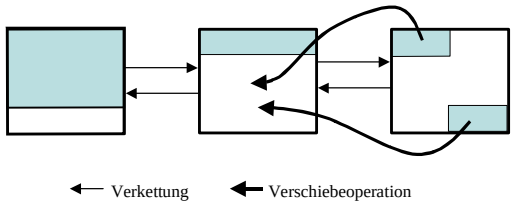


Abbildung 4.1: Verdichtung durch Umverteilung in benachbarte Blattknoten

Während der Reorganisation kann es durch den parallel laufenden Datenbankbetrieb dazu kommen, dass bereits reorganisierte und damit „abgehandelte“ Knoten wieder geteilt werden. Somit ist das Ergebnis nicht zwingend optimal, was aber toleriert wird.

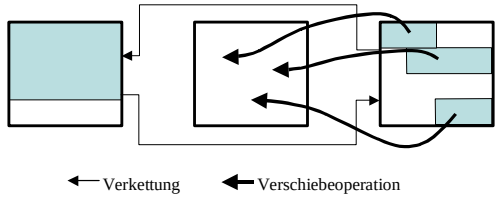


Abbildung 4.2: Verdichtung durch Verschieben in einen leeren Block

- Nachdem alle Blattknoten reorganisiert wurden, erfolgt in der zweiten Phase das Verschieben der Inhalte ganzer Blattknoten so, dass logisch aufeinanderfolgende Blattknoten auch physisch aufeinanderfolgend gespeichert werden (*Abbildung 4.3*).

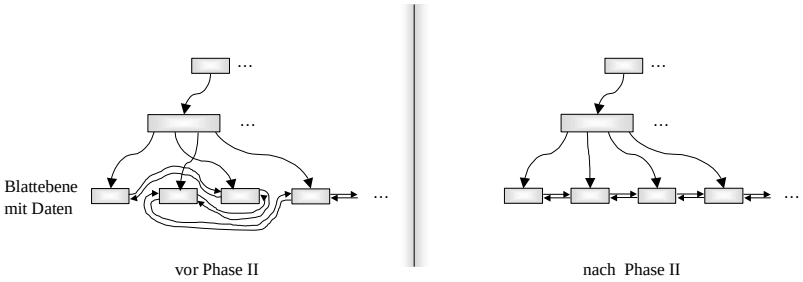


Abbildung 4.3: Ordnen der Blattknoten

- In der dritten Phase erfolgt schließlich die Reorganisation der weiteren inneren Ebenen. Da diese im Vergleich zur Blattebene nur wenig Speicher belegen, erfolgt hier ein kompletter Neuaufbau. Der alte obere Teil des Index bleibt bis zum Ende dieser Phase erhalten und wird für den normalen Datenbankbetrieb genutzt. Abschließend wird auf den neuen Index umgeschaltet. Zwischenzeitlich am Originalindex vorgenommene Änderungen werden in einem sog. *Side File* vorgemerkt und vor dem Umschalten am neuen Index nachvollzogen.

4.1.2 In-Place-Reorganisation von als Heap organisierten Datenbereichen

Bei *der Implementierung* von Möglichkeiten zur Online-Reorganisation von Datenbanken zu beachtende Aspekte werden in **[SD92]** erörtert. Zur Sicherung einer möglichst hohen Verfügbarkeit der Daten wird vorgeschlagen, während der Reorganisation die Sätze der zu reorganisierenden Tabelle einzeln und somit nach und nach zu verschieben (*Abbildung 4.4*). Der durch Verschiebeoperationen frei werdende Speicher soll, soweit möglich, als Ziel für nachfolgende Verschiebeoperationen verwendet werden. Alle mit dem Verschieben eines Satzes verbundenen Aktionen sind innerhalb einer Transaktion durchzuführen.

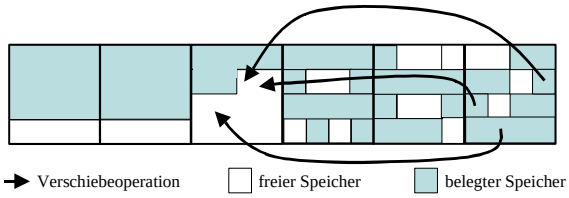


Abbildung 4.4: Beispiel für das Verschieben einzelner Sätze

Zu diesen Aktionen gehört auch die Aktualisierung von Indexen. Die Effizienz der Reorganisationsmethode könnte gesteigert werden, wenn in einer Transaktion größere Granulate als einzelne Tupel (also ganze Mengen) bewegt werden würden. Allerdings würde dies durch den höheren Bedarf an Ressourcen für den Reorganisationsprozess und durch größere zu sperrende Bereiche wieder zu stärkeren Behinderungen der Anwendungen (Sperrkonflikte, Performance-Einbußen) führen. Damit Reorganisationen parallel zum laufenden Datenbankbetrieb ausgeführt werden können, sind teilweise Anpassungen und Erweiterungen vorhandener Funktionalitäten nötig. So wäre die Realisierung der Verschiebeoperation durch eine Verknüpfung vorhandener Lösch- (`DELETE`) und Einfügefunktionen (`INSERT`) denkbar. Allerdings muss dann beispielsweise die Ausführung bestimmter Trigger ebenso unterdrückt werden wie die Prüfung von Constraints zur Sicherung referenzieller Integrität (sonst würden „scheinbare“ Inkonsistenzen entdeckt und evtl. behandelt). Weiterhin muss bei Suchoperationen sichergestellt werden, dass verschobene Sätze nicht „überlesen“ oder doppelt gefunden werden. Hier bestehen somit Ähnlichkeiten mit dem bekannten Phantomproblem beim Mehrbenutzerbetrieb.

Eine Methode zur *Clusterung von Sätzen*, die *am Ort* durchgeführt wird, wird in **[Omi85, SPO89]** präsentiert. Eingaben für den vorgestellten Algorithmus sind die zu reorganisierende Datenmenge und eine von einem Cluster-Algorithmus ermittelte Folge von sog. *logischen Seiten*. Eine logische Seite enthält jeweils die Identifikatoren der Sätze, die durch die Reorganisation gemeinsam in einem *physischen Block* untergebracht werden sollen. Eine Ordnung der logischen Seiten untereinander existiert nicht.

Die „Konstruktion“ von reorganisierten Blöcken erfolgt mit möglichst wenigen I/O-Operationen in einem eigenen Pufferbereich (Reorganisationspuffer). Die Abläufe zeigt *Abbildung 4.5*. Zunächst wird für jede logische Seite ein Wert für die Kosten errechnet, die anfallen würden, um alle physischen Blöcke zu lesen, die Datensätze

der logischen Seite enthalten. Für die logische Seite mit dem kleinsten Kostenwert werden die entsprechenden Blöcke gelesen und die benötigten Sätze in den zu erzeugenden physischen Block im Puffer kopiert. Danach werden für die verbliebenen logischen Seiten die Kostenwerte neu berechnet und das Verfahren wird wiederholt. Konnte der zu erzeugende Block vollständig gefüllt werden, so gilt seine Bearbeitung als abgeschlossen und er wird auf den Datenträger geschrieben.

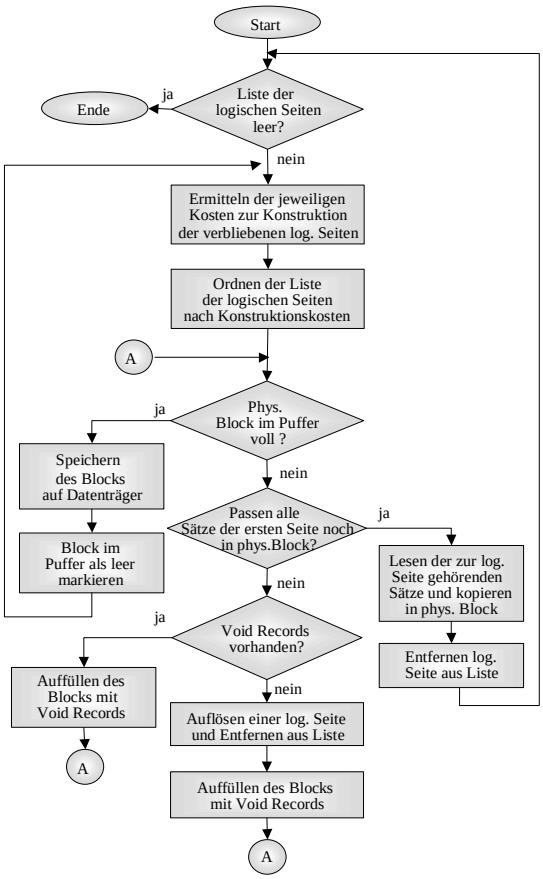


Abbildung 4.5: Konstruktion der reorganisierten Blöcke

Füllen die Datensätze der zuletzt betrachteten logischen Seite den Block nicht voll aus, so wird unter Berücksichtigung neu berechneter Kostenwerte für das Lesen der zu einer logischen Seite gehörenden Kostenwerte versucht, den Block mit den Sätzen weiterer ganzer logischer Seiten aufzufüllen. Gelingt dies nicht (bspw. weil im physischen Block nicht mehr genug Platz für die kleinste logische Seite vorhanden ist), so wird zunächst versucht, den Block mit Sätzen aufzufüllen, die mit keinen anderen Sätzen zu clustern sind (sog. *Void Records*) und sich möglichst im Puffer befinden. Existieren im bisher noch nicht reorganisierten Teil der Datei nicht mehr ausreichend viele Void Records, so wird eine logische Seite aufgelöst (entfernt) und die in ihr enthaltenen Sätze werden zu Void Records erklärt. Dem Auffüllen der Blöcke wird also Vorrang gegenüber der vollständigen Einhaltung der angestrebten Clusterung eingeräumt. Zur Integritätssicherung werden die im Reorganisationspuffer befindlichen Blöcke kurzzeitig gesperrt [OLS94].

Die durch die Verschiebung der Sätze notwendige Aktualisierung eventuell vorhandener Indexe erfolgt asynchron, ähnlich den in [OLA91] beschriebenen Verfahren (*vgl. auch Abschnitt 4.1.3*).

4.1.3 Aktualisierung von Sekundärindexen

In **[ZS96b, ZS98]** wird eine Methode zur *inkrementellen Änderung von physischen Verweisen (RID, TID) in Sekundärindexen* im Rahmen der Reorganisation von Tabellen entwickelt. Die Grundidee zur Reduzierung von physischem I/O-Aufwand während der Reorganisation basiert darauf, die Pflege der jeweiligen Verweise erst dann durchzuführen, wenn die entsprechenden Blöcke durch Operationen des regulären Datenbankbetriebs ohnehin in den Arbeitsspeicher gelesen werden. Durch die Reorganisation verursachte zusätzliche I/O-Operationen werden so zunächst vermieden. Die Pflege der Indexeinträge erfolgt in zwei Phasen. Die Vorgehensweise in der *ersten Phase* zeigt *Abbildung 4.6*.

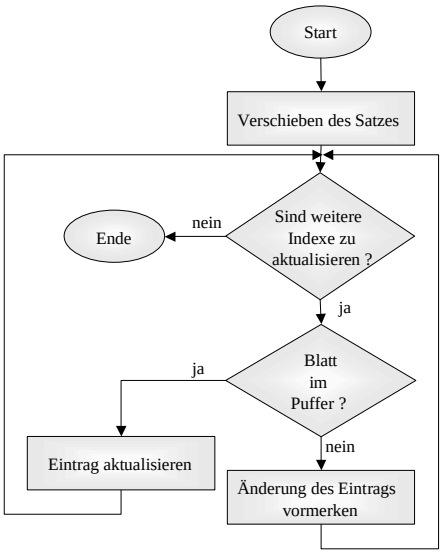


Abbildung 4.6: Inkrementelle Änderung der Verweise in Sekundärindexen – erste Phase

Wenn ein Satz durch den Reorganisationsprozess (oder eine Änderungsoperation) verschoben wird, werden nur die entsprechenden Einträge der Sekundärindexe sofort angepasst, die aktuell in Blöcken im Puffer gehalten werden. Befindet sich mindestens ein zu aktualisierender Block nicht im Puffer, so wird für den Satz ein Eintrag in eine sog. *Forward Address Table* eingefügt, die temporär die Zuordnung alter Verweise zu neuen Verweisen enthält. In einer für den jeweiligen Index angelegten sog. *Pending Changes Table* werden je Block die noch zu aktualisierenden Indexeinträge ebenfalls temporär vermerkt. Beide Tabellen werden im Arbeitsspeicher gehalten. Im Falle eines drohenden „Überlaufs“ werden durch einen separaten Prozess die aufgezeichneten Aktualisierungen der Sekundärindexe vorgenommen und dadurch die Tabellen geleert, bevor das Verschieben von Datensätzen fortgesetzt wird. Zur Sicherung der Wiederherstellbarkeit nach

Systemfehlern werden die während der Reorganisation durchgeführten Aktionen im Transaktionsprotokoll (Log) vermerkt.

In der *zweiten Phase* erfolgt die eigentliche Aktualisierung der betroffenen Sekundärindexe (*Abbildung 4.7*). Wird ein entsprechender Block in den Puffer geladen, so wird zunächst vom Puffer-Manager die entsprechende Pending Changes Table geprüft. Sind dort Aktualisierungsanforderungen für den Block verzeichnet, so werden die Verweise unter Nutzung der Forward Address Table geändert, bevor der Block „nach oben“ zur Verfügung gestellt wird.

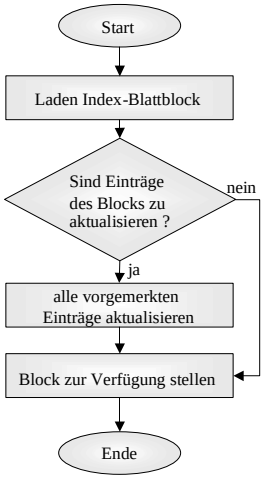


Abbildung 4.7: Inkrementelle Änderung der Verweise in Sekundärindexen – zweite Phase

Auch wenn im normalen Datenbankbetrieb in Tupeln enthaltene Werte indexierter Attribute geändert werden, ist eine Aktualisierung der betroffenen Indexe nötig. Die Verweise auf die zugehörigen Sätze müssen aus den Verweislisten (vgl. auch *Abschnitt 3.3.2*) der Indexeinträge mit den alten Attributwerten entfernt und in die Verweislisten der Indexeinträge der neuen Attributwerte eingefügt werden. Eine Methode, die die notwendigen Aktualisierungen *zeitverzögert* und *inkrementell* ausführt, wird in **[OLA91]** vorgeschlagen. Die an einem Index notwendigen Änderungen werden hier erst durchgeführt, wenn dieser im Rahmen von Operationen des normalen Datenbankbetriebs nach dem alten oder neuen Schlüsselwert sowieso durchsucht (d.h. die relevante Stelle „angefasst“) wird. Im Unterschied zum Verfahren aus [ZS96b, ZS98] werden hier die Änderungsanforderungen nicht bezüglich der Indexblöcke aufgezeichnet und ausgeführt, sondern (inkrementell) jeweils für die mit den Schlüsselwerten der Anfragen in Verbindung stehenden Indexeinträge. Je Index werden in einer aus Effizienzgründen im Arbeitsspeicher gehaltenen und als *Differential File* bezeichneten Tabelle für jeden geänderten Satz dessen RID sowie der alte und der neue Attributwert gespeichert. Um deren Größe zu begrenzen, wird je geändertem Satz nur ein Eintrag gehalten, unabhängig von der Zahl der am Satz vorgenommenen Änderungen.

Bei der Änderung des Werts eines Attributs wird jetzt nur noch der eigentliche Datensatz verändert. Für jeden zu aktualisierenden Index wird entweder ein neuer Eintrag in das entsprechende Differential File eingefügt oder ein bereits von einer

früheren Änderung her bestehender Eintrag aktualisiert. I/O-Operationen für die Aktualisierung der Indexe entfallen.

Die Implementierung von Suchoperationen, die Indexe nutzen, muss erweitert werden. Zunächst wird der Index durchmustert, bis der zum Suchschlüssel gehörende Indexeintrag gefunden ist. Anschließend erfolgt im Differential File eine Suche nach Einträgen, die ebenfalls den Suchschlüsselwert enthalten. Werden solche Einträge gefunden, dann werden am zum Suchschlüssel gehörenden Indexeintrag die notwendigen Änderungen vorgenommen (nachgeholt). Anschließend werden die Datensätze zur Verfügung gestellt, auf die im Indexeintrag verweisen wird. Wurden die Einträge für den alten und den neuen Schlüsselwert aktualisiert, wird der Eintrag für den Datensatz aus dem Differential File entfernt.

Für Suchoperationen ergibt sich bei der Anwendung der Methode eine Erhöhung des I/O-Aufwands um die Operation zum Rückschreiben des geänderten Indexblocks. Bei beiden Verfahren ist allerdings zu berücksichtigen, dass durch die verzögerte Pflege von Indexen aus „eigentlich reinen“ Lese-Transaktionen implizit Änderungstransaktionen werden können.

4.1.4 Clusterung von Daten unter Berücksichtigung von Zugriffsmustern

Bei einer Datenanforderung erfolgt jeweils ein Zugriff auf den die Daten enthaltenden Block. Wenn der benötigte Block nicht im Puffer gehalten wird, so muss er von Sekundärspeichermedien gelesen und in den Puffer gebracht werden. Dieser Fall wird als *Page Fault* (Fehlseitenbedingung) bezeichnet. Vor dem Lesen muss u.U. erst (gemäß der im konkreten Fall verwendeten Ersetzungsstrategie) Pufferspeicher zur Verfügung gestellt werden. In **[BF77]** wird ein selbstorganisierendes Dateisystem vorgeschlagen, dass eine Verringerung des Auftretens von Fehlseitenbedingungen zum Ziel hat. Dabei wird angenommen, dass mit einer I/O-Operation nicht nur der jeweils benötigte Block, sondern eine Menge von Blöcken gelesen wird (vorausschauendes Lesen – *Prefetching*). Dadurch werden bei jeder I/O-Operation eventuell auch Blöcke „sinnlos“ mit in den Puffer gebracht, auf deren Inhalte bis zum Auslagern der Blöcke nicht zugegriffen wird.

Die Idee hinter der als *Permutation Clustering* bezeichneten Methode liegt darin, die physischen Speicherorte der im Puffer befindlichen referenzierten Blöcke durch Austauschen mit den Speicherorten von unreferenzierten Blöcken so zu verändern, dass die von einem Prozess (z.B. dem DBMS) referenzierten Blöcke beim Rückschreiben auf die Sekundärspeichermedien physisch beieinanderliegend gespeichert werden. Damit kann zukünftig das Lesen benötigter Blöcke beschleunigt werden. Für die Clusterung werden bei dieser Methode keine Dateninhalte, sondern die Zugriffshäufigkeit auf bestimmte Daten verwendet. Die Methode wird kontinuierlich auf alle Blöcke angewendet, auf die ein Prozess während seiner Laufzeit zugreift. Werden Blöcke aus- und später wieder eingelagert, so werden sie eventuell erneut den Austauschoperationen unterzogen. Veränderungen im Zugriffsmuster spiegeln sich somit in sich verändernden physischen Speicherreihenfolgen von Blöcken wider. Wenn zur Indexierung von Datenbeständen

physische Referenzen verwendet werden, ist die Anwendung dieser Vorgehensweise allerdings problematisch. Die Referenzen müssten oft angepasst werden oder es müssen zusätzliche Indirektionen (bspw. Umsetzungstabellen) benutzt werden.

Der Ansatz, Muster für Zugriffe auf Daten zu bestimmen und anschließend die Daten so anzuordnen, dass der I/O-Aufwand bei Datenanforderungen reduziert wird, wird auch in **[GKM96, MK94, Wan00]** verfolgt. Das s*elf-adaptive on-line reclustering framework* [MK94], die *MCR tool box*[7] [GKM96] bzw. das Gesamtsystem beim *Dynamic Online Data Clustering* [Wan00] bestehen jeweils aus drei wesentlichen Komponenten (*Abbildung 4.8*). Die Systeme unterscheiden sich dabei hauptsächlich in Details der Implementierung.

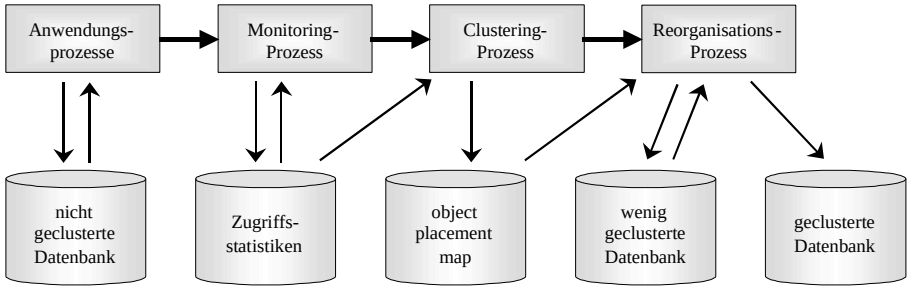


Abbildung 4.8: Komponenten der MCR tool box (nach [GKM96])

Ein *Monitoring-Prozess* sammelt Informationen über die von Anwendungen getätigten Datenzugriffe und erstellt daraus Statistiken. Aus diesen lassen sich Zugriffsmuster ableiten, auf deren Basis ein *Cluster-Analysealgorithmus* eine Object Placement Map bestimmt. Dies ist eine Folge von logischen Seiten (siehe auch [Omi85, SPO89] in *Abschnitt 4.1.2*), in denen die Zuordnung von Datenobjekten zu den Blöcken vorgenommen wird. Beispielsweise werden Objekte in einem Block untergebracht, die mit hoher Wahrscheinlichkeit gemeinsam innerhalb einer Transaktion benötigt werden. Ziel ist es dabei, Datenzugriffe mit minimalem I/O-Aufwand zu ermöglichen. Ein *Reorganisationsprozess* nimmt schließlich die Umordnung der Datenobjekte gemäß den Vorgaben des Cluster-Analysealgorithmus vor.

In [Wan00] werden gegenüber den anderen Systemen Erweiterungen vorgeschlagen. So wird beispielsweise für jede logische Seite geprüft, ob mit dem Aufbau des mit ihr verbundenen physischen Datenblocks tatsächlich ein Nutzen verbunden ist, bevor der eigentliche Reorganisationsprozess gestartet wird.

4.1.5 Reorganisation in eine Kopie

In **[SBC97]** wird eine Methode zur Online-Reorganisation von Datenbankobjekten beschrieben, die dort als *Fuzzy Reorganization* bezeichnet wird und eine *reorganisierte Kopie* der Originaltabelle erstellt. Am Ende der Reorganisation werden die entsprechenden Einträge im Datenbankkatalog geändert und dadurch die

[7] MCR steht hier für Monitoring, Clustering, Reorganization.

Originaltabelle und die ihr zugeordneten Indexe durch die Kopien ersetzt. Ziel der Reorganisation ist die Beseitigung von migrierten Tupeln und eingestreutem Freiplatz sowie die Wiederherstellung der internen Sortierung der Daten nach dem Ordnungskriterium eines Clustered Index. Die bei derartigen Methoden zur Reorganisationsdurchführung anfallenden I/O-Kosten werden in *Kapitel 7* noch näher betrachtet. Die Reorganisation erfolgt in mehreren Schritten (*Abbildung 4.9*).

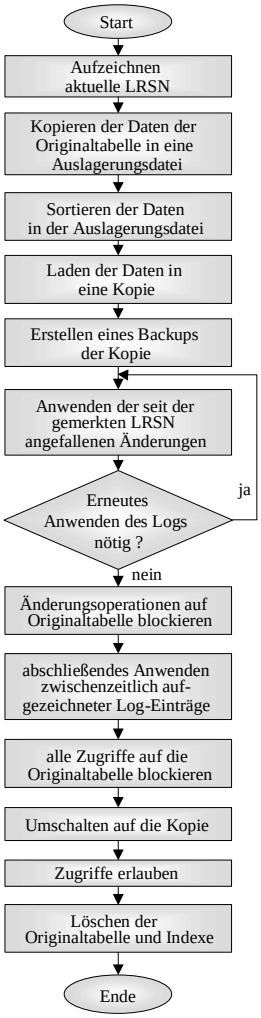


Abbildung 4.9: Ablauf der Reorganisation unter Nutzung einer Kopie

Die Methode erlaubt während der meisten Zeit der Reorganisation lesenden und schreibenden Zugriff auf die Daten der Originaltabelle. Dort vorgenommene Änderungen werden anhand des Transaktions-Logs später in der Kopie nachvollzogen. Zur Synchronisation wird zu Beginn der Reorganisation die aktuelle Log Record Sequence Number (LRSN) festgehalten. Im zweiten Schritt werden die Daten der Originaltabelle in eine Datei ausgelagert, dort sortiert und in eine Kopie der Originaltabelle in der Datenbank geladen. Weiterhin werden gemäß dem Vorbild der Originaltabelle Indexe angelegt. Um für den Fall späterer Fehler eine Recovery zu ermöglichen, wird noch ein Backup der Kopie erstellt. Anschließend werden die im Log aufgezeichneten Änderungen in der Kopie nachvollzogen.

Um die im Log aufgezeichneten Operationen auf die Kopie anwenden zu können, muss eine Zuordnung der RIDs der Sätze der Originaltabelle zu den korrespondierenden RIDs der Kopie erfolgen. Dieses Problem wird durch die Verwendung einer *Mapping-Tabelle* gelöst. Abhängig von der Intensität der Änderungen an der Originaltabelle kann das Anwenden der im Log aufgezeichneten Operationen mehrfach wiederholt werden, um den abschließenden Aktualisierungslauf möglichst kurz zu halten. Vor diesem abschließenden Aktualisierungslauf wird auf die Beendigung aktiver Änderungsoperationen gewartet und es werden keine neuen Änderungen zugelassen. Während des letzten Laufs sind nur lesende Zugriffe auf die Originaltabelle möglich. Danach werden für einen kurzen Zeitraum alle Benutzerzugriffe auf die Originaltabelle unterbunden. Die neue Tabelle nimmt den Platz der alten ein, Benutzerzugriffe werden wieder ermöglicht und die alte Tabelle und die ihr zugeordneten Indexe werden gelöscht.

4.1.6 Kontinuierliche Reorganisation

Bei den bisher beschriebenen Methoden finden Datenbankreorganisationen i.d.R. zyklisch und auf Veranlassung des DBA statt. In [Soe80, Soe81] werden für (Netzwerk-)Datenbanken die Möglichkeiten *kontinuierlich parallel zum normalen Datenbankbetrieb laufender Reorganisationen* untersucht. Das in [BF77] beschriebene selbstorganisierende Dateisystem führt ebenfalls kontinuierlich Reorganisationen durch. Der Beitrag wurde allerdings in eine andere Gruppe eingeordnet, weil dort die Clusterung von Datenblöcken im Vordergrund steht. Das Ziel kontinuierlich durchgeführter Reorganisationen ist die Vermeidung von Unterbrechungen des Datenbankbetriebs durch Reorganisationen. Teile der Datenbanken werden jeweils zu lastarmen Zeiten neben dem laufenden Datenbankbetrieb reorganisiert.

Die Reorganisation wird bei der in **[Soe80, Soe81]** beschriebenen Methode von einem eigenen, asynchron laufenden Reorganisationsprozess mit niedriger Priorität ausgeführt. Dieser durchläuft nacheinander die Strukturen der Datenbank und beseitigt vorhandene Degenerierungen. Statistische Informationen über einen evtl. höheren oder niedrigeren Reorganisationsbedarf unterschiedlicher Bereiche der Datenbank werden nicht berücksichtigt. Der Reorganisationsprozess wird zwischenzeitlich angehalten, wenn Operationen von Anwendungsprozessen auszuführen sind. Allerdings werden kritische Operationen (beispielsweise das Verändern von Verweisen) vor der Unterbrechung erst beendet, damit die Integrität der Strukturen der Datenbank gewahrt bleibt. In [Soe81] werden Kostenbetrachtungen angestellt und die Wirksamkeit eines solchen Reorganisationsprozesses anhand von Messungen an einem Simulationsmodell untersucht. Von Interesse ist hierbei besonders, bis zu welchen Grenzen der Reorganisationsprozess in der Lage ist, mit der Entwicklungsgeschwindigkeit von Degenerierungen Schritt zu halten.

Parallel zum normalen Betrieb stattfindende Reorganisationen werden in der Literatur auch für Dateisysteme betrachtet. In **[RO92]** wird ein Entwurf für Dateisysteme

beschrieben, der dahingehend optimiert ist, in Anwendungsumgebungen, in denen überwiegend kleine Dateien bearbeitet werden müssen, den für Schreibvorgänge anfallenden Aufwand zu reduzieren. Dabei wird davon ausgegangen, dass sich durch die zunehmenden Möglichkeiten, größere Teile von Dateien im Puffer zu halten, das Verhältnis zwischen physischen Lese- und Schreiboperationen deutlich zu Gunsten der Schreiboperationen verschiebt. Der Vorschlag ist als *Log-Structured File System* (LFS) bekannt.

Schreiboperationen finden bei solchen File-Systemen stets in einem Bereich mit einer physisch zusammenhängenden Menge freier Blöcke (dem *Log-Bereich - Abbildung 4.10*) statt.

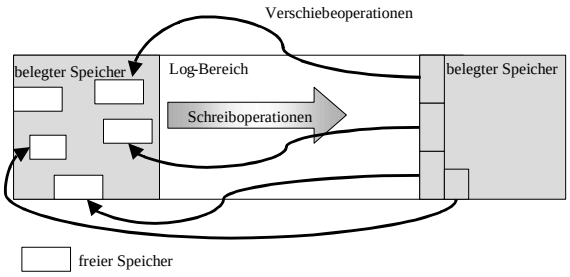


Abbildung 4.10: Log-Structured File System

Zur Beschleunigung von Schreiboperationen werden mehrere „kleine“ Schreiboperationen zu einem größeren Schreibauftrag zusammengefasst. Das Schreiben erfolgt dann sequenziell in den Log-Bereich, der sich langsam füllt. Dabei wird auch nicht versucht, zu schreibende Blöcke einer Datei jeweils physisch in der Nähe eventuell bereits vorhandener Blöcke der Datei unterzubringen. Durch Lösch- und Änderungsoperationen werden in den bereits gefüllten Bereichen ehemals belegte Blöcke freigegeben. Es kommt zu Fragmentierungen des freien Speichers und zu einer physisch gemischten Speicherung der Blöcke verschiedener Dateien. Damit sich aber die erwarteten Geschwindigkeitsvorteile bei Schreiboperationen ergeben, ist es wichtig, dass immer ein ausreichend großer physisch zusammenliegender Bereich freier Blöcke für Schreiboperationen zur Verfügung steht. Deshalb wird der Speicherbereich parallel zur Nutzung reorganisiert. Reorganisationen werden immer dann durchgeführt, wenn das entsprechende Externspeichermedium nicht benutzt wird.

4.2 Bestimmung von Reorganisationszeitpunkten bzw. –intervallen

4.2.1 Parametrierbare Verfahren

Das Problem der Bestimmung optimaler Reorganisationsintervalle wird in der Literatur mehrfach behandelt. Bereits in **[Tue78, YDT76]** wird jeweils eine Lösung zur näherungsweisen Bestimmung von Reorganisationsintervallen für Dateien mit linearem Wachstum präsentiert. Ziel ist es, die *Gesamtkosten* für Zugriffe auf die Datei möglichst gering zu halten. Dabei wird davon ausgegangen, dass die Zugriffskosten zu einem bestimmten Zeitpunkt bekannt sind. Weiterhin muss bekannt

sein, wie sich die Zugriffskosten durch an der Datei vorgenommene Änderungen entwickeln. Dem allgemeinen Trend folgend, dass mehr Daten in eine Datei eingefügt als gelöscht werden, wird zunächst von einem linearen Wachstum der Datei und der Zugriffskosten ausgegangen.

Zwei Wachstumsraten beschreiben die Steigerung der Zugriffskosten. Die erste Wachstumsrate beschreibt die Steigerung, die sich lediglich aus der größeren zu verarbeitenden Datenmenge ergibt. Hier wird angenommen, dass die Datei kontinuierlich reorganisiert wird, also alle Speicherungsstrukturen permanent vollständig gepflegt werden. Über diese Wachstumsrate wird die Kostenentwicklung im Idealfall dargestellt. Die zweite Wachstumsrate beschreibt das tatsächliche Anwachsen der Zugriffskosten (ohne Reorganisation) in einem bestimmten Zeitraum. Mit zyklisch durchzuführenden Reorganisationen wird versucht, den Verlauf der real anfallenden Kosten der idealen Entwicklung anzunähern.

Auch die Reorganisationskosten müssen bekannt sein. Der ideale Zeitpunkt für eine Reorganisation ist immer dann erreicht, wenn zu einem Zeitpunkt die Summe aus Zugriffskosten im Idealfall und Reorganisationskosten den tatsächlich anfallenden Zugriffskosten entspricht. Dadurch ergibt sich für den Verlauf der tatsächlichen Zugriffskosten eine „Sägezahnkurve“ (*Abbildung 4.11*), bei der die anfallenden Kosten jeweils unmittelbar nach einer Reorganisation dem Idealfall entsprechen. Hierbei wird auch berücksichtigt, dass die Reorganisationen nicht in festen Intervallen erfolgen können, weil mit steigender Dateigröße auch die Kosten für die Reorganisationen anwachsen (wie in Abbildung 4.11 angedeutet).

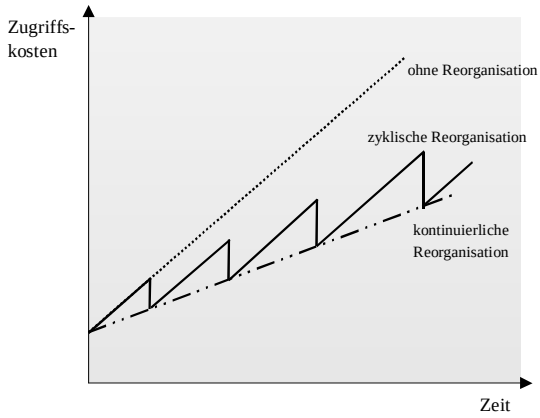


Abbildung 4.11: Entwicklung der Zugriffskosten (nach [Tue78, YDT76])

Als problematisch an dieser Methode stellt sich insbesondere die Ermittlung der Wachstumsraten für Zugriffs- und Reorganisationskosten dar, da diese über einen längeren Zeitraum verfolgt werden müssen und mit Unregelmäßigkeiten zu rechnen ist.

Neben den Zugriffskosten wird in **[YDT76]** auch die Entwicklung der Gesamtkosten (Zugriffs- und Reorganisationskosten) betrachtet, die in bestimmten Zeiträumen anfallen. Eine Reorganisation zu einem bestimmten Zeitpunkt senkt zwar die Kosten für Datenzugriffe, die Gesamtkosten sind jedoch, bedingt durch die von der Reorganisation verursachten Kosten, zunächst höher. Eine Reduzierung der

Gesamtkosten durch die Reorganisation tritt erst zeitverzögert bzw. über die Zeit verteilt ein. Die sich daraus ergebende Kostenentwicklung ist in *Abbildung 4.12* skizziert.

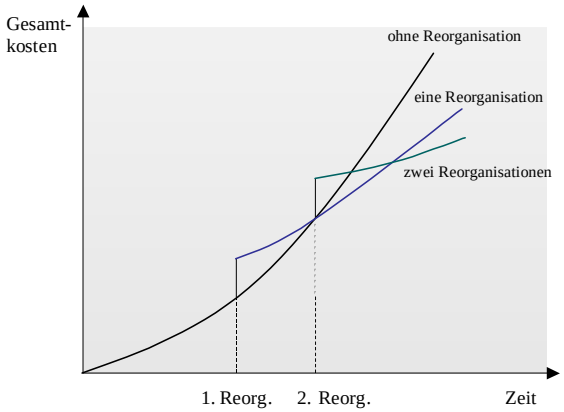


Abbildung 4.12: Entwicklung der Gesamtkosten (nach [YDT76])

Im Zusammenhang mit der Beschreibung des *Dynamic Online Data Clustering-Systems* für Objektdatenbanken werden in **[Wan00]** auch Methoden zur Bestimmung von Überwachungs- und Reorganisationsintervallen vorgeschlagen. Dort wird die Fehlseitenrate (vgl. Abschnitt 4.1.4) der Anwendungen als Maß für die Systemleistung angesehen. Diese soll verringert werden, indem Datenobjekte, die mit hoher Wahrscheinlichkeit gemeinsam innerhalb von Anwendungstransaktionen benötigt werden, im gleichen Datenblock gespeichert (geclustert) werden.

Es wird davon ausgegangen, dass eine Datenbank zyklisch drei Phasen durchläuft. In der ersten Phase befindet sich die Datenbank im *ungeclusterten* Zustand. Die Fehlseitenrate entspricht der eines Systems ohne Möglichkeiten zum Clustern von Daten. Während der *Reorganisation* erhöht sich die Fehlseitenrate der Anwendungstransaktionen gegenüber dem ungeclusterten Zustand durch den mit der Reorganisation verbundenen zusätzlichen Aufwand. Nach der Reorganisation liegt die Datenbank im *geclusterten* Zustand vor und die Fehlseitenrate liegt unter der im ungeclusterten Zustand. Durch Einfüge- und Änderungsoperationen geht die Clusterung im Laufe der Zeit (der Nutzungsdauer) wieder verloren und es muss erneut reorganisiert werden etc.

Anhand von drei unterschiedlichen Modellen für diesen Zyklus werden Berechnungsformeln vorgestellt, mit denen u.a. der Zeitraum bis zur Amortisation der Reorganisationskosten bestimmt werden kann. Die drei vorgestellten Modelle unterscheiden sich in den Annahmen über den Verlauf der Verschlechterung der Fehlseitenrate während der Nutzungsdauer nach der Clusterung.

Beim *Step Function Model* wird davon ausgegangen, dass die durch die Reorganisation verringerte der Fehlseitenrate während der gesamten Nutzungsdauer konstant bleibt und an deren Ende sprungartig auf den Wert im ungeclusterten Zustand ansteigt. Dieses Modell und die damit verbundenen Berechnungen sind zwar einfach, aber nicht realitätsnah. Beim *General Model* kann eine beliebige

differenzierbare Funktion für den Verlauf und die Erhöhung der Fehlseitenrate angenommen werden.

Beim *Straight-Line Model* wird unterstellt, dass die Erhöhung der Fehlseitenrate während der Nutzungsdauer linear erfolgt. *Abbildung 4.13* zeigt, mit Beispielwerten versehen, den beim Straight-Line Model angenommenen Verlauf der Fehlseitenrate über einen Zyklus. Es wird angenommen, dass die durch die unterbrochene Linie dargestellte „normale“ Fehlseitenrate (im Beispiel 10%) auftritt, wenn die Datenbank im ungeclusterten Zustand belassen (d.h. nicht reorganisiert) wird.

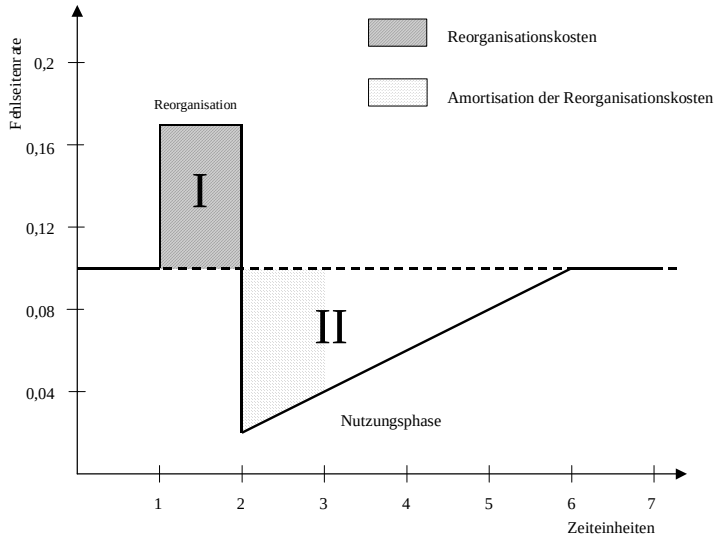


Abbildung 4.13: Änderung der Fehlseitenrate im Straight-Line Model (nach [Wan00])

Durch den durch die Reorganisation verursachten Mehraufwand steigt die Fehlseitenrate der Anwendungsprozesse im Beispiel auf einen Wert von ca. 17%. Unmittelbar nach Abschluss der Reorganisation sinkt die Fehlseitenrate auf einen Wert von 2% und steigt im Laufe der nächsten vier Zeiteinheiten wieder auf einen Wert von 10% an. In diesem Zeitraum ergibt sich der Nutzen der Reorganisation, von dem ein Teil zur Amortisation der Reorganisationskosten aufgebraucht wird. Die Ermittlung des Nutzens kann über einen Vergleich der Flächen **I** und **II** erfolgen.

Für die drei Modelle werden in [Wan00] Berechnungsformeln zur Bestimmung von Reorganisationszyklen vorgestellt, bei deren Einhaltung der Nutzen maximiert wird. Voraussetzung für die Anwendung der Berechnungsverfahren ist allerdings das Vorliegen von Erfahrungswerten. Der DBA muss erst im Rahmen (möglichst) mehrerer Durchläufe durch Messungen die Länge der Reorganisationsphase und die Länge der Nutzungsphase bestimmen. Dies setzt eine Gleichmäßigkeit und Kontinuität der Abläufe voraus, die bei realen Anwendungssystemen allerdings nur schwer zu finden sein dürfte. Weiterhin müssen die Fehlseitenraten im ungeclusterten Zustand, während der Reorganisation und unmittelbar nach der Reorganisation gemessen werden.

4.2.2 Verfahren mit Berücksichtigung von Workload-Informationen

In **[Bat82]** wird eine Methode zur Bestimmung von Reorganisationsintervallen und Dateiparametern für indexsequenzielle Dateien und für auf Hash-Verfahren basierende Dateien beschrieben. Die Methode verwendet Beschreibungsinformationen (sog. File Descriptor) über die jeweilige Datei, die u.a.

- den Dateityp,
- die Anzahl Sätze, die in einem Block gespeichert werden können,
- die Anzahl Sätze in der Datei,
- die Wahrscheinlichkeit, dass ein Satz in einen Überlaufblock eingefügt werden muss,
- die Speicherkosten, die je Block in einer Zeiteinheit anfallen sowie
- die Kosten für das Lesen und Schreiben einzelner Blöcke

enthalten. Weiterhin wird die Anzahl an Einfüge-, Lösch-, Änderungs- und Suchoperationen (Workload-Informationen) berücksichtigt, die in einem bestimmten Zeitraum ausgeführt werden. Ziel ist es, die gesamten Verarbeitungskosten (inkl. Reorganisationen) möglichst gering zu halten. Die Ermittlung der Verarbeitungskosten erfolgt dabei über Kostenfunktionen, die die Dateibeschreibungsinformationen nutzen. Da sich die Beschreibungsinformationen ändern, wird jeweils der Kostendurchschnitt des aktuellen und des vorangegangenen Intervalls ermittelt. Als Stellgrößen werden die *Länge eines Reorganisationsintervalls* oder ein *Loading Factor* (ähnlich PCTFREE) verwendet. Ausgehend von einem vorgegebenen Loading Factor wird die Länge der Reorganisationsintervalle berechnet. Umgekehrt kann zu einem fest vorgegebenen Reorganisationsintervall der Loading Factor, der bei der Reorganisation verwendet werden muss, bestimmt werden.

4.3 Physische Redefinition von Datenbankobjekten

4.3.1 Änderung der physischen Organisationsform

Eine Methode zur Konvertierung der Organisationsform von Datenbeständen wird in **[Omi88, Omi89]** beschrieben. Hier werden in B^+-Bäume eingebettete Datenbestände in Hash-Dateien umgewandelt. Eine solche Konvertierung kann beispielsweise notwendig werden, wenn sich die gegen den Datenbestand gerichtete Workload von einem Mix aus Punkt- und Bereichsanfragen bzw. sequenziellem Suchen hin zu nahezu ausschließlich auftretenden Punktanfragen geändert hat. Die Umwandlung erfolgt *online* und *am Ort* (*in place*). Dazu werden die Blattknoten (Datenblöcke) des Baums nacheinander bearbeitet. Für die einzelnen Sätze eines Blocks wird über die Hash-Funktion jeweils das Bucket errechnet, in das der Satz einzufügen ist. Anschließend werden die Sätze in die Hash-Datei übertragen und aus dem Baum gelöscht.

Um Nutzertransaktionen dirigieren zu können, die während der Konvertierung auf den Datenbestand zugreifen, wird der Schlüsselwert des letzten konvertierten Satzes gemerkt (*Abbildung 4.14*). Besitzen die von einer Nutzeranfrage benötigten Sätze einen größeren Schlüsselwert (als den gemerkten), so muss die Anfrage auf den noch nicht konvertierten Teil der Baums zugreifen, sonst auf die Hash-Datei. Die in Bearbeitung befindlichen Einheiten werden kurzzeitig mit Lesesperren versehen, um Änderungen durch parallel laufende Anwendungen zu verhindern.

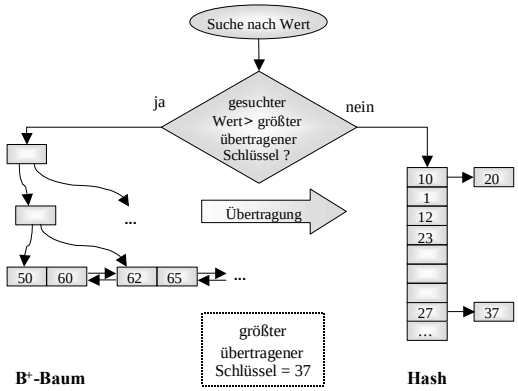


Abbildung 4.14: Konvertierung eines B^+-Baums in eine Hash-Struktur

In [Omi89] wird auch eine Methode zur Konvertierung in umgekehrter Richtung vorgestellt. Dabei werden die Buckets der Hash-Datei der Reihenfolge nach verarbeitet. Bei jedem Konvertierungsschritt wird der Inhalt eines Buckets, inklusive des Inhalts einer eventuell vorhandenen Überlaufkette, bearbeitet (*Abbildung 4.15*).

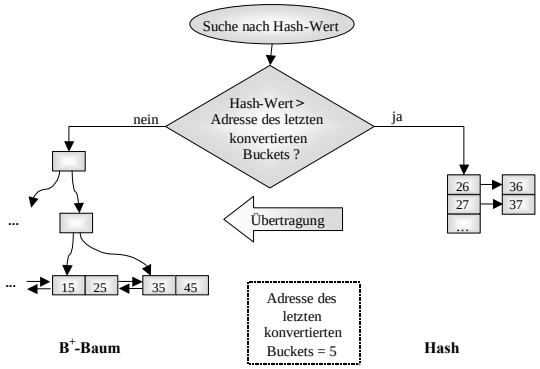


Abbildung 4.15: Konvertierung einer Hash-Struktur in einen B^+-Baum

Die aktuell zu bearbeitenden Sätze werden nach dem Suchschlüssel, über dem der B^+-Baum aufgebaut werden soll, sortiert und mit den für B^+-Bäume üblichen Verfahren in den Baum eingefügt. Um hier parallel zur Reorganisation ablaufende Zugriffe von Benutzertransaktionen zu dirigieren, wird zunächst der Hash-Wert für die benötigten Datensätze errechnet. Ist dieser größer als die Adresse des letzten konvertierten Buckets, so muss (noch) in der Hash-Datei gesucht werden, sonst (schon) im B^+-Baum.

4.3.2 Umverteilung bei der Nutzung von Partitionierungskonzepten

Zu Reorganisationen auf der Definitionsebene zählt auch die Umverteilung von Daten. Änderungen an Partitionierungsschemata oder Veränderungen der Workload können solche Umverteilungen erforderlich machen. In **[AON96]** werden zwei verschiedene Methoden zum Online-Verschieben von Daten und zum Aufbau von Indexen im Umfeld paralleler Datenbanksysteme gemäß der Shared-Nothing-Architektur betrachtet. Prinzipiell ähneln die Vorgehensweisen aber denen von lokalen Datenbanken. Ausgangspunkt der Betrachtungen ist die Verlagerung eines Datenbestands (z.B. einer Partition oder einer Tabelle) von einem Verarbeitungsknoten auf einen anderen Verarbeitungsknoten. Bei den betrachteten Ansätzen steht eine weitestgehend uneingeschränkte Verfügbarkeit der Daten während des Verschiebens im Vordergrund.

Der erste Ansatz wird als OAT (One-At-a-Time page movement) bezeichnet. Der Ablauf ist in *Abbildung 4.16* skizziert.

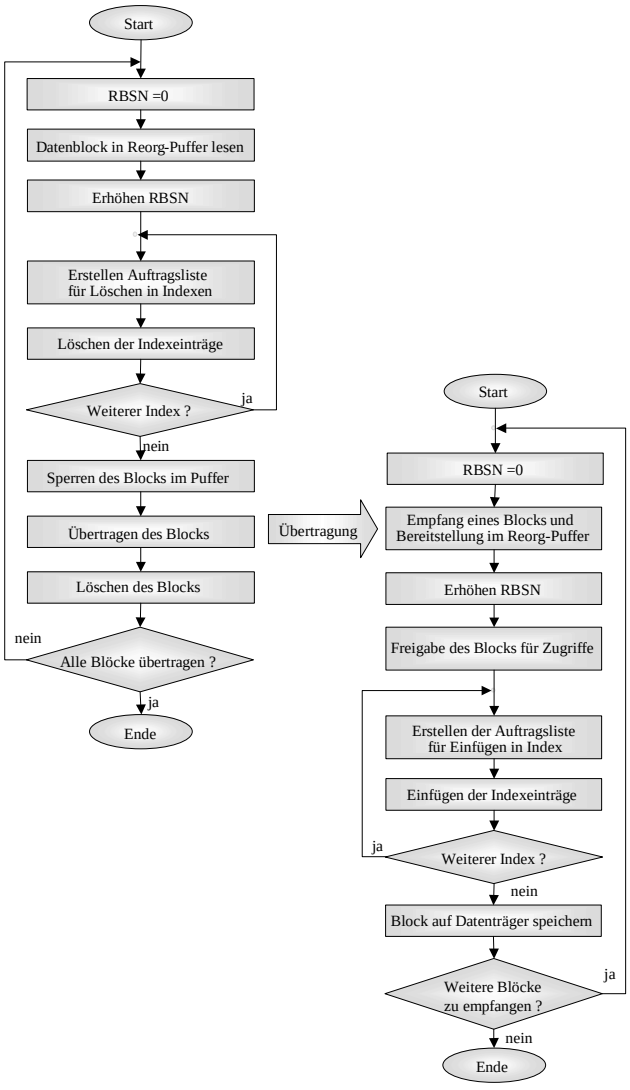


Abbildung 4.16: Verschieben von Daten auf einen anderen Knoten mit der OAT-Methode

Bei diesem Ansatz werden die Übertragung der Daten und der Indexaufbau *inkrementell* durchgeführt. Zum Verschieben wird auf beiden Verarbeitungsknoten ein Puffer (*Reorg-Puffer*) verwendet. Die Blöcke werden nur während der eigentlichen Übertragung zwischen den Systemen für Benutzerzugriffe kurzzeitig gesperrt. Um die Konsistenz der Ergebnisse parallel laufender Datenbankoperationen zu gewährleisten, wird eine *Reorg Buffer Sequence Number* (RBSN) geführt. Aus Effizienzgründen werden zum Löschen von Indexeinträgen auf dem Ursprungssystem und zum Einfügen auf dem Zielsystem nach den jeweiligen Schlüsseln sortierte Auftragslisten genutzt. Nachdem alle Indexe aktualisiert wurden, wird der Block aus dem Puffer auf den Zieldatenträger geschrieben. Diese Vorgehensweise wird wiederholt, bis alle Datenblöcke übertragen wurden.

Die Anwendung dieser Methode erfordert auch Anpassungen von Such- und Änderungsoperationen, die den Inhalt des Reorg-Puffers berücksichtigen müssen. Suchoperationen müssen u.U. teilweise auf dem Quell- und auf dem Zielsystem durchgeführt werden. Zur Vermeidung inkonsistenter Anfrageergebnisse (z.B. durch „vergessene“ oder doppelt gelesene Sätze) werden die RBSNs verwendet. Anhand der aktuellen RBSN von Quell- und Zielsystem kann der globale Transaktionsmanager mit den in *Tabelle 4.1* dargestellten Entscheidungsregeln ermitteln, ob das Anfrageergebnis korrekt ist. Wird festgestellt, dass Blöcke fehlen, so kann der globale Transaktionsmanager die Subtransaktion auf dem Zielsystem erneut starten. Wurden Blöcke doppelt gelesen, so muss die Transaktion abgebrochen und erneut gestartet werden.

Bei der zweiten betrachteten Methode (BULK-Methode) wird zunächst der gesamte Datenbestand auf das Zielsystem übertragen. Anschließend werden dort die Indexe aufgebaut. Während der Übertragung stehen Daten und Indexe auf dem Quellsystem uneingeschränkt zu Verfügung.

Quellsystem	**Zielsystem**	**Ergebnis der Prüfung**
$RBSN_q$	$RBSN_z = RBSN_q - 1$	Treffermenge korrekt
$RBSN_q$	$RBSN_z = RBSN_q$	Block doppelt gelesen
$RBSN_q$	$RBSN_z < RBSN_q - 1$	ein oder mehrere Blöcke fehlen

Tabelle 4.1: Prüfung der Korrektheit der Treffermenge

Änderungen, die auf dem Quellsystem an den Daten vorgenommen werden, werden auf dem Zielsystem im Anschluss an den Aufbau der Indexe nachvollzogen. Anschließend werden Nutzertransaktionen auf den Datenbestand des Zielsystems umgelenkt. Abschließend wird der Originaldatenbestand auf dem Quellsystem gelöscht. Vorteile der BULK-Methode liegen u.a. auch darin, dass I/O-Operationen verwendet werden können, die mehrere Blöcke mit einer Operation lesen bzw. schreiben können. Dies führt zu einer Beschleunigung der notwendigen I/O-Operationen und vermindert die Gefahr der Fragmentierung von Daten und Indexen beim Neuaufbau auf dem Zielsystem. Ein weiterer Vorzug der BULK-Methode gegenüber der OAT-Methode ist, dass keine Änderungen an den Operationen der

normalen Datenbankverarbeitung notwendig sind. Der größte Nachteil der BULK-Methode ist allerdings, dass sie während der Reorganisation durch die Erstellung einer vollständigen Kopie von Datenbereichen und Indexen einen wesentlich höheren Speicherbedarf als die OAT-Methode aufweist.

Das Thema der Umverteilung von Daten zwischen den Verarbeitungsknoten (Systemen) in Shared-Nothing-Systemen wird auch in **[LKO+00]** behandelt. Ausgangspunkt sind *bereichsweise partitionierte Tabellen*, deren Partitionen jeweils als lokale B^+-Bäume mit eingebetteten Daten (IOT) organisiert sind und auf verschiedenen Systemen gehalten werden. Im Laufe der Zeit kann es unter den Systemen zu *Schieflagen* (signifikanten Unterschieden) bezüglich des zu verwaltenden Datenvolumens oder bezüglich der Anfragebelastung kommen. Durch das Verschieben von Teilen von Partitionen (und entsprechendes Ändern der Partitionierungsbereiche) sollen vorhandene Schieflagen beseitigt werden.

Um den laufenden Datenbankbetrieb möglichst wenig zu beeinflussen wird vorgeschlagen, jeweils ganze Teilbäume von einem stärker belasteten System auf ein (vom verwalteten Partitionierungsbereich her) benachbartes System zu verschieben (*Abbildung 4.17*). Das Verschieben auf das jeweilige Nachbarsystem erfolgt nach der BULK-Methode. Nachdem die zu verschiebenden Daten auf das Zielsystem übertragen und dort für die übertragenen Daten ein eigener Indexbaum aufgebaut wurde, wird dieser einfach als Teilbaum am unteren bzw. oberen Ende in den Indexbaum der auf dem Zielsystem verwalteten Partition eingefügt. Auf dem Ausgangssystem wird der Teilbaum entfernt. Das Einfügen bzw. Entfernen ganzer Teilbäume kann realisiert werden, ohne dass dazu größere Bereiche der Indexe auf dem Quell- und Zielsystem über längere Zeiträume hinweg gesperrt werden müssen.

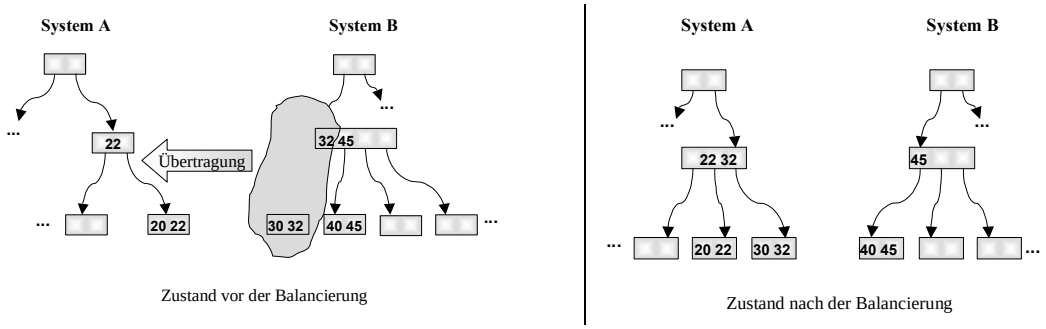


Abbildung 4.17: Lastbalancierung durch Verschieben ganzer Teilbäume

Der Beitrag **[GBG04]** befasst sich ebenfalls mit dem Problem der Lastbalancierung bei bereichspartitionierten Tabellen in parallelen Datenbanken. Hier wird vorgeschlagen, eine dynamische Lastbalancierung nach Einfüge- bzw. Löschoperationen zu veranlassen, wenn definierte Schwellwerte über- oder unterschritten wurden. Die oben beschriebene Vorgehensweise des Verschiebens von Daten zwischen benachbarten Systemen kann u.U. Kettenreaktionen (das Fortsetzen von Umverteilungsoperationen über weitere Systeme) auslösen. Um dieses Problem zu vermeiden, wird eine kombinierte Vorgehensweise vorgeschlagen, die auch ein Umordnen der Reihenfolge der Verarbeitungsknoten ermöglicht.

Nach Einfüge- oder Löschoperation wird (bspw. anhand von Informationen über Datenverteilungen) geprüft, ob eine Umverteilung von Daten erforderlich ist. Ist das der Fall, so wird zunächst versucht, eine Lastbalancierung mit den benachbarten Systemen durchzuführen. Ist das nicht möglich, weil beispielsweise die benachbarten Systeme ebenfalls stark belastet sind, so wird das System mit der geringsten Last gesucht. Dieses versucht, die von ihm verwalteten Daten an seine Nachbarsysteme abzugeben. Ist dies erfolgt, so wird das System zum neuen Nachbar des überlasteten Verarbeitungsknotens und übernimmt von diesem die Hälfte der Daten. *Abbildung 4.18* zeigt dies anhand eines Beispiels.

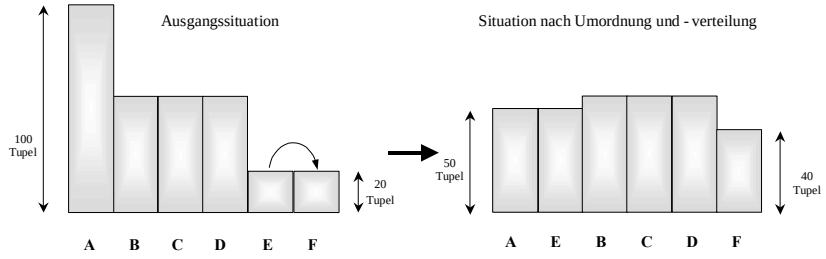


Abbildung 4.18: Umordnen der Verarbeitungsknoten (nach [GBG04])

Hier wird der Wertebereich der Partition F so erweitert, dass sie den gesamten Wertebereich der Partition E aufnimmt. Das dadurch frei werdende System E wird jetzt (bezüglich des verwalteten Datenbereichs) zum Nachbar von System A und übernimmt die Hälfte der bisher auf A gespeicherten Daten. Dieses Umordnen ist durch Änderungen an Metadaten über die Partitionierungsbereiche ohne größeren Aufwand realisierbar.

Abbildung 4.18 zeigt auch mögliche Kosteneinsparungen durch die Umordnung. Um die im Beispiel rechts gezeigte Datenverteilung zu erreichen, müssen beim Umordnen von Verarbeitungsknoten 70 Tupel bewegt werden. Um das gleiche Ergebnis mit mehreren nacheinander ablaufenden Umverteilungen („Kettenreaktion“) zwischen benachbarten Verarbeitungsknoten zu erreichen, müssten 250 Tupel bewegt werden.

In **[SWS+05]** wird ebenfalls das Thema des Zusammenführens von B-Baumstrukturen im Online-Betrieb behandelt. Allerdings wird dies hier als Mischoperation (B-Tree Merging) und nicht über das „Umketten“ von Teilbäumen realisiert. Solche Operationen können z.B. beim Zusammenführen der Indexstrukturen von bisher getrennten Partitionen notwendig sein, wenn die Wertebereiche der Indexschlüssel nicht disjunkt sind (z.B. weil bisher nach der Round-Robin-Strategie oder einem anderen Schlüssel partitioniert wurde). Im Beitrag wird zwischen einem sog. *Main Index* (Hauptindex), der erhalten bleiben soll und einem *Secondary Index*, der in den Hauptindex integriert werden soll, unterschieden.

Um den I/O-Aufwand für das Zusammenführen der Indexe zu reduzieren, werden die Mischoperationen mit Operationen des normalen Datenbankbetriebs verknüpft. Nachdem eine Mischoperation initialisiert wurde, wird (bis zum vollständigen Abschluss der Operation) bei jedem Zugriff auf einen Blattknoten des Hauptindex eine Systemtransaktion zum Einfügen von Werten aus dem zu integrierenden Index

gestartet (*Abbildung 4.19*). Nach Abschluss dieser Systemtransaktion wird die Operation des normalen Datenbankbetriebs fortgesetzt.

Die Systemtransaktion prüft zunächst, ob für den betrachteten Blattknoten bereits eine Mischoperation stattgefunden hat. Dafür werden die LSNs (Log Sequence Numbers) der Indexknoten verwendet. Diese geben Auskunft darüber, mit welchem Eintrag im Log die letzte Änderung am Knoten verknüpft ist.

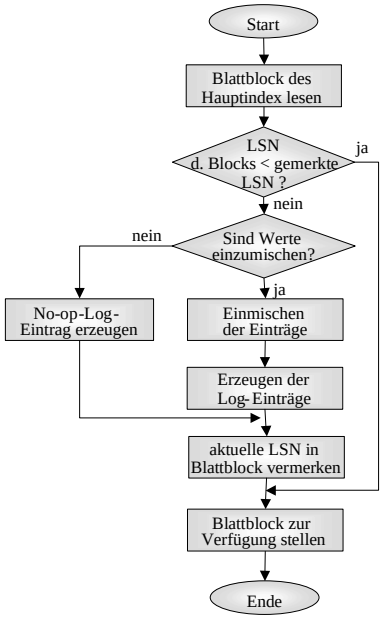


Abbildung 4.19: Mischen von Indexeinträgen

Bei der Initialisierung der gesamten Mischoperation wird die aktuelle LSN gemerkt. Besitzt ein Blattknoten eine höhere LSN als die gemerkte, so wurde für diesen Knoten bereits eine Mischoperation durchgeführt und die Systemtransaktion wird beendet, anderenfalls wird das Mischen ausgeführt. Durch die Einführung von sog. *No op Log Records* wird sichergestellt, dass auch für Knoten, bei denen kein Mischen notwendig ist, die LSN geändert wird. Um notwendige Mischoperationen auch für Indexknoten auszuführen, die im Rahmen des normalen Datenbankbetriebs über längere Zeiträume hinweg nicht benutzt werden, wird ein Hintergrundprozess mit niedriger Priorität eingesetzt.

4.4 Stand bei Produkten – Techniken und Werkzeuge

In einigen Beiträgen werden in Verbindung mit konkreten DBMS-Produkten Methodiken zur Reorganisationsdurchführung und zur Verlängerung von Reorganisationsintervallen vorgestellt. Die Erläuterungen lassen sich aber durchaus verallgemeinern. Die angeführten Beiträge stellen nur einige Beispiele dar, die sich mit dem Thema Datenbankreorganisation auseinander setzen. Die geschilderten Techniken sowie weitere, teilweise auch auf spezielle Anwendungsumgebungen bezogene Lösungen werden bspw. im Internet oder auch in Leitfäden für Datenbankadministratoren behandelt.

Mit der Beschleunigung (offline durchgeführter) systembasierter Reorganisationen von Oracle-Datenbanken befasst sich **[Nob95]**. Um die durch die Reorganisation verursachte Unterbrechung des Datenbankbetriebs so kurz wie möglich zu halten, wird die Ausnutzung möglichst vieler zur Verfügung stehender Ressourcen während der Reorganisationsdurchführung empfohlen. Weiterhin sollten die auszuführenden Aktionen weitgehend parallelisiert werden. Der Neuaufbau von Datenbankobjekten und der Import von Daten kann wesentlich beschleunigt werden, wenn bei diesen Vorgängen die üblichen Mechanismen zur Transaktionssicherung deaktiviert werden. Allerdings ist dabei zu beachten, dass vor und baldmöglichst nach der Reorganisation eine Sicherung der betroffenen Datenbankobjekte erfolgen muss, um die Wiederherstellbarkeit zu gewährleisten. Die meisten der vorgestellten Techniken wurden mittlerweile in verfügbare Werkzeuge zur Datenbankadministration integriert. Diese bieten dem DBA eine handhabbare Schnittstelle, bei der er bspw. nur noch entscheiden muss, *ob* Reorganisationsaufgaben parallel erledigt werden sollen. Die Werkzeuge versuchen dann, einen in der konkreten Systemumgebung sinnvollen Parallelitätsgrad zu ermitteln. Weiterhin wird bestimmt, in welchem Umfang Ressourcen benötigt werden und in welcher Reihenfolge die Reorganisationsaufgaben durchzuführen sind. Entsprechende Vorschläge werden dem DBA unterbreitet. In **[Ash00]** wird auf einige der in [Nob95] beschriebenen Techniken im Zusammenhang mit der Beschreibung der Produkte *SpaceManager* und *LiveReorg* von Quest Software eingegangen. Durch den Verzicht auf Mechanismen zur Transaktionssicherung und die Veränderung von Konfigurationsparametern im Rahmen der Reorganisation sowie die Parallelisierung von Operationen beim Neuaufbau von Datenbankobjekten kann die Dauer von Reorganisationen auch hier erheblich verkürzt werden. Es werden Empfehlungen zur Einstellung der Parameterwerte im SpaceManager gegeben.

In **[HL03]** werden am Beispiel von Oracle Techniken zur Verringerung von Fragmentierungen in Datenbanken beschrieben. Ziel ist die Verlängerung der Zeitintervalle zwischen Datenbankreorganisationen. Es wird empfohlen, innerhalb eines Table Space jeweils nur Datenbankobjekte mit gleichen Extent-Größen zu speichern. Damit können durch Löschungen frei gewordene Bereiche für andere Datenbankobjekte genutzt werden. Die Administrierbarkeit von großen Datenbankobjekten kann vereinfacht werden, wenn für sie jeweils eigene Table Spaces verwendet werden. Weiterhin werden Vor- und Nachteile der Speicherung von Daten in als Heap organisierten Tabellen und indexorganisierten Tabellen gegenübergestellt. Empfehlungen zur Festlegung von Parametern wie `PCTFREE` und `PCTUSED` werden gegeben. Zur Beschleunigung von Massenlöschungen (bspw. im Rahmen einer turnusmäßigen Archivierung von Daten [Ste02]) wird die horizontale Partitionierung von Tabellen und Indexen angeregt. Entsprechende Partitionierungsschemata vorausgesetzt, können solche Löschoperationen einfach über das Ausgliedern und anschließende Löschen ganzer Partitionen realisiert werden.

In den letzten Jahren wurden die in die Produkte integrierten Verfahren zur automatischen Verwaltung von Sekundärspeicher deutlich verbessert. Insbesondere kritische Probleme, wie das baldige Erschöpfen zur Verfügung stehenden Speichers,

können größtenteils vermieden werden. Aber auch eine vorausschauende Planung des Speicherplatzbedarfs von Datenbankobjekten, die sonst aufwendig „von Hand“ erfolgen muss(te), wird z.B. bei Oracle 10g über „Create Table Estimator“ und „Growth Trend Report“ unterstützt. Damit können die Speicherverwendung optimiert, der Reorganisationsbedarf verringert und Zeitintervalle zwischen Datenbankreorganisationen verlängert werden [Ora03d, Ora05].

Da Reorganisationen von Datenbankobjekten aber trotzdem nicht gänzlich zu vermeiden sind, haben Hersteller von DBMS-Produkten und Drittanbieter in der Vergangenheit daran gearbeitet, Verfahren zur Online-Reorganisation in ihre Produkte zu integrieren [BHL+02, Ora03d]. Dabei werden *In-Place-Verfahren* ebenso angeboten wie Verfahren, die eine reorganisierte *Kopie* erstellen und nach Abschluss der Reorganisation auf die Kopie umschalten. Auf konkrete Implementierungen der Verfahren wird in *Kapitel 7* noch näher eingegangen.

Werkzeuge zur Unterstützung von Reorganisationsentscheidungen und zur Reorganisationsdurchführung, wie z.B. REORGCHK/REORG für DB2 [IBM02], Oracle Enterprise Manager [Ora02, Ora03d], BMC Space Expert [BMC04, BR02] oder DBArtisan [EMB03], überwachen Kennzahlen über den Zustand physischer Speicherungsstrukturen (Anteil eingestreuten Freispeichers, Anteil migrierter Tupel, Speicherauslastung von Table Spaces, etc.). Über- oder unterschreiten bestimmte Kennzahlen vom System vorgegebene oder vom DBA festgelegte Schwellwerte, so wird dies typischerweise signalisiert, damit der DBA über die weitere Vorgehensweise entscheiden kann. Was keines der Werkzeuge liefert, sind quantifizierte Aussagen, welcher Nutzen durch eine Reorganisation bezüglich der Systemleistung erreicht werden könnte, da keines der Werkzeuge Informationen über die gegen die Datenbankobjekte gerichtete Workload in die Analysen einbezieht. Dieses Thema wird in *Kapitel 6* näher betrachtet.

Im Rahmen von [Hei04] wurden die genannten Werkzeuge auf ihre Leistungsfähigkeit hin untersucht. Ein Überblick über die Ergebnisse findet sich in [Dor05a]. Als Basis dienten die DBMS-Produkte Oracle9i/10g und DB2 UDB V8.1. Diese Systeme wurden einerseits wegen ihrer Verbreitung ausgewählt, andererseits auch wegen der Verfügbarkeit im Rahmen der durchgeführten Untersuchung. Da die Untersuchungen auch praktische Tests einschließen sollten, wurden nur Produkte berücksichtigt, die von den Herstellern zu Testzwecken zur Verfügung gestellt wurden. In *Tabelle 4.2* werden die betrachteten Werkzeuge verglichen. Dazu werden folgende Kriterien herangezogen.

- Verfügbarkeit für unterschiedliche Betriebssystemplattformen sowie unterschiedliche DBMS-Produkte. (♦ = beschränkt auf ein bestimmtes System, ✓ = Verfügbarkeit für unterschiedliche Plattformen)
- Reorganisationsmöglichkeiten von Datenbankobjekten. (♦ = Reorganisation offline möglich, ✓ = Reorganisation online möglich, × = Reorganisation nicht möglich, keine Angabe = Funktionalität bzw. Konzept wird vom Werkzeug bzw. vom DBMS-Produkt nicht oder nicht direkt angeboten)

- Reorganisationsgranulate (✓ = frei zusammenstellbar, ♦ = auf einzelne Tabellen bzw. Indexe beschränkt)
- verfügbare Reorganisationsmethoden (✓ = ja, × = nein, ♦ = Reparatur migrierter Tupel, keine Angabe = unbekannt)
- Besonderheiten der Tools (z.B. zeitgesteuerte Reorganisationsausführung, Durchführbarkeitstest ✓ = ja, × = nein, ♦= nur Prüfung, ob genügend Speicherplatz zur Verfügung steht)
- Basis der Reorganisationsempfehlung
- Änderbarkeit der Speicherungsparameter der Datenbankobjekte (✓ = ja, × = nein)

Produkt	Oracle 9i/10g	DBArtisan für Oracle	Quest Central for Databases	BMC Space Expert	IBM-Tools für DB2
Betriebssystemplattformen	✓	♦	✓	✓	✓
Unterstützung verschiedener DBMS	♦	✓	✓	✓	♦
Reorganisationsmöglichkeiten von Datenbankobjekten:					
Heap-Tabellen ohne Indexe	♦	♦	✓	♦	♦
Heap-Tabellen mit Primärschlüssel	✓	♦	✓	♦	✓
Indexe	✓	✓	✓	♦	✓
Partitionen von Tabellen	✓		✓		✓
Indexpartitionen	✓		✓		
Indexorganisierte Tabellen	✓	✓	✓	♦	
Table Spaces	♦	♦	♦	♦	×
Reorganisationsgranulate	✓	✓	✓	✓	♦
Reorganisationsmethoden:					
Export/Import	✓	×	✓	✓	✓
internes Umkopieren	✓	✓	✓	✓	✓
In-Place-Methode	✓		♦		✓
Besonderheiten:					
zeitgesteuerte Ausführung	✓	✓	✓	✓	✓
Assistent zur Job-Erstellung	✓	✓	✓	✓	✓
Durchführbarkeitstest	✓	♦	♦	✓	×
Basis der Reorganisationsempfehlung:					
statische Kennzahlen	✓	✓	✓	✓	✓
Verfahren mit Workload-Berücksichtigung	×	×	×	×	×
Änderbarkeit der Speicherparameter	✓	✓	✓	✓	×

Tabelle 4.2: Vergleichsübersicht für die betrachteten Werkzeuge

Erwartungsgemäß sind die von den DBMS-Herstellern mitgelieferten Werkzeuge nur für das jeweilige eigene Produkt verfügbar, während die Produkte von Drittanbietern i.d.R. für unterschiedliche Plattformen verfügbar sind. Dies macht die Werkzeuge der Drittanbieter für den Einsatz in heterogenen Systemumgebungen besonders interessant. Sie ermöglichen die Administration unterschiedlicher Basissysteme mit größtenteils einheitlichen Methoden und Vorgehensweisen. Dabei ist allerdings zu beachten, dass Funktionalitäten oder Einstellmöglichkeiten (bspw. auch die des Vorhandenseins von In-Place-Methoden) teilweise vom Basissystem abhängig sind und damit natürlich im Detail Unterschiede auftreten können.

Die geringere Menge an unterstützten Typen von Datenbankobjekten bei den Werkzeugen der IBM für DB2 ist darin begründet, dass DB2 in der betrachteten Version einige der aufgeführten Objekttypen nicht kennt.

Dass gesamte Table Spaces mit den Werkzeugen von IBM für DB2 (REORGCHK und REORG) nicht zusammenhängend analysiert und reorganisiert werden können liegt daran, dass die Werkzeuge keine freie Zusammenstellung der Granulate ermöglichen. Ist eine freie Zusammenstellung des Granulats möglich, so können beliebige zu reorganisierende Objekte einer Datenbank bzw. eines Table Space in einem Reorganisations-Job zusammengefasst werden.Weitere Unterschiede zwischen den Werkzeugen finden sich in der Tiefe der Durchführbarkeitstests, der Handhabbarkeit und der Übersichtlichkeit der Darstellung der Analyseergebnisse.

Zusammenfassend gesehen bieten die Werkzeuge eine gute Unterstützung für die Planung und Durchführung von Datenbankreorganisationen. Unterschiede bestehen in der Möglichkeit für den DBA, die entsprechenden Schwellwerte für die Kennzahlen selbst festlegen zu können, in Umfang und Tiefe der durchgeführten Reorganisationsbedarfsanalysen, der Präsentation der Ergebnisse, der Handhabbarkeit und darin, ob und wie umfassend für die erstellten Reorganisations-Jobs Durchführbarkeitsprüfungen angestellt werden. Auch die Realisierung von Methoden zu Online-Reorganisationen ist unterschiedlich, was natürlich auch an den von den unterschiedlichen Basissystemen (hier konkret: Oracle versus DB2) zur Verfügung gestellten Funktionalitäten liegt.

4.5 Zusammenfassung und kritische Würdigung

Die in den vorangegangenen Abschnitten angestellten Betrachtungen zu unterschiedlichen Aspekten des Themengebiets Datenbankreorganisation zeigen, dass sich der überwiegende Teil der Veröffentlichungen mit Methoden und Verfahren zur Online-Reorganisationsdurchführung befasst. Dabei werden zwei prinzipielle Vorgehensweisen aufgezeigt, unabhängig davon, ob die Reorganisation zur Beseitigung von Degenerierungen, zur Konvertierung von Speicherungsstrukturen oder zur Umverteilung von Daten ausgeführt wird.

Bei der einen Vorgehensweise erfolgt die Reorganisation *inkrementell* (also nach und nach) in *Einheiten, die klein genug sind*, dass die mit der Reorganisation verbundene Sperrung von Teilen des Datenbestands den laufenden Datenbankbetrieb nur wenig behindert. Solche Reorganisationen erfolgen meist zu großen Teilen *am Ort (in*

place), so dass kaum zusätzlicher Speicher zur Verfügung gestellt werden muss. Allerdings kann das notwendige Aufzeichnen der Operationen im Log u.U. einen erheblichen zusätzlichen Aufwand verursachen. Teilweise müssen erweiterte Sperrkonzepte entwickelt und angewendet werden, um Behinderungen des normalen Datenbankbetriebs gering zu halten. Auch Anpassungen und Erweiterungen „normaler“ Datenbankoperationen sind oftmals notwendig. Dabei werden beispielsweise Änderungsoperationen an Indexen mit Suchoperationen verknüpft. Dies kann u.U. zur Änderung von Workload-Charakteristika führen. So kann z.B. die Ausführung einer lediglich aus Retrieval-Operationen bestehenden Workload Änderungen an Datenbankinhalten vornehmen, Log-Einträge erzeugen, evtl. Sperrkonflikte verursachen usw., was sicherlich nicht immer akzeptabel ist.

Bei der anderen Vorgehensweise wird eine (soweit möglich) degenerierungsfreie *Kopie* erstellt, während Benutzerzugriffe noch auf den Originaldatenbestand erfolgen. Anschließend werden, ähnlich den Vorgehensweisen beim Online-Backup, die zwischenzeitlich am Originaldatenbestand erfolgten Änderungen an der Kopie nachvollzogen. Zum Schluss wird das Original durch die Kopie ersetzt. Diese Methode benötigt mehr Speicherplatz als die In-Place-Verfahren. Allerdings steht der Reorganisationsprozess nicht in direkter Konkurrenz zu Benutzertransaktionen. Daher sind auch Änderungen an der Implementierung von Datenbankoperationen i.d.R. nicht notwendig. Weiterhin kann davon ausgegangen werden, dass die Tabelle und die zugehörigen Indexe am Ende der Reorganisation frei von Degenerierungen sind. Dies kann bei den In-Place-Methoden nicht immer gewährleistet werden.

Die Betrachtung verschiedener DBMS-Produkte sowie einiger Werkzeuge von Drittanbietern hat gezeigt, dass Funktionalitäten zur Online-Reorganisation von Datenbanken auch hier verfügbar sind. Auch die Lokalisierung von Degenerierungen stellt i.d.R. keine Schwierigkeit dar. Eine Quantifizierung des durch eine Reorganisation erreichbaren Nutzens, besonders bezüglich der Systemleistung, wird aber nicht vorgenommen, weil keine Workload-Informationen verwendet werden. Hier finden sich vereinzelt Ansätze in der Literatur.

Die parametrierbaren Verfahren berücksichtigen für einzelne Dateien einstellbare Größen wie Wachstumsraten, Reorganisationskosten, Entwicklung von Verarbeitungskosten usw. Aus diesen Größen wird errechnet, in welchen Intervallen Reorganisationen durchgeführt werden sollten, um Verarbeitungskosten insgesamt möglichst gering zu halten. Als problematisch erscheint dabei allerdings aus heutiger Sicht bei großen Anwendungssystemen (mit teilweise vielen tausend Tabellen) allein schon die Ermittlung der Erfahrungswerte für die jeweiligen Parameter.

Die Grundidee aus [Bat82], einen eventuell vorhandenen Reorganisationsbedarf aus Informationen über den Zustand von Speicherungsstrukturen und Workload zu bestimmen, wird von uns aufgegriffen. Allerdings soll hier nicht vordergründig die Dauer von Reorganisationsintervallen bestimmt, sondern der zum Zeitpunkt der Analyse erreichbare *Nutzen* der Datenbankreorganisation *quantifiziert* werden, ohne dabei auf Erfahrungswerte zurückgreifen zu müssen.

Weiterhin ergeben sich einige neue Herausforderungen gegenüber der in [Bat82] betrachteten Situation, die berücksichtigt werden müssen. Die dort beschriebene

Methode zielt auf die *Minimierung der gesamten Verarbeitungskosten*. Ob dieses „hehre“ Ziel bei Anwendungen im Produktivbetrieb konsequent verfolgt werden kann, erscheint fraglich. Verfügbarkeitsanforderungen und Speicherkosten können dafür sorgen, dass die automatisch bestimmten Reorganisationsintervalle oder Speicherungsparameter nicht eingehalten werden können. Die *Workload-Informationen* beruhen auf Annahmen bzw. Erfahrungen aus der Vergangenheit. Dies setzt voraus, dass entsprechende Informationen über längere Zeiträume hinweg gesammelt wurden (und auch für die absehbare Zukunft weiter Aussagekraft besitzen).

Die *Vorhersage wahrscheinlicher zukünftiger Zustände* von physischen Speicherungsstrukturen, ausgehend von aktuellen Werten und Workload-Informationen, ist besonders bei großen Datenbanksystemen ebenfalls kaum durchführbar. Es ist durch die Vielfalt an Speicherungsstrukturen und deren Kombinationsmöglichkeiten sowie vorhandene Unterschiede in der Implementierung von Operationen äußerst schwierig, für existierende DBMS mit vertretbarem Aufwand die genaue Entwicklung von Degenerierungen vorherzusagen. Die Menge der Einflussfaktoren ist einfach zu groß. Neben Strukturen und Operationen spielen beispielsweise auch die zu speichernden Daten selbst eine Rolle. Unklar ist bspw. meist auch die genaue zu erwartende Verteilung von Schlüsselwerten, die Zugangsreihenfolge von Sätzen, die Reihenfolge der auszuführenden Änderungsoperationen usw. Ob und wie ein DBMS vorhandene Freispeicherlücken wieder verwendet, ist im Ergebnis (Datenbankzustand) ebenfalls kaum allgemein gültig vorhersagbar.

Die jeweils aktuelle Ermittlung des Zustands der physischen Speicherungsstrukturen aus statistischen Werten liefert hier nach unserer Meinung hinreichend genaue Ergebnisse. Trends könnten dabei ermittelt werden, wenn über einen Zeitraum hinweg (wie in [Wil04] beschrieben) mehrere zeitpunktbezogene Zustandsinformationen (Snapshots von Statistikdaten) gesammelt und gespeichert werden.

Ein weiteres Problem bei der in [Bat82] beschriebenen Methode besteht darin, dass die zur Workload-Abarbeitung verwendeten *Grundoperationen* und deren Ausführungshäufigkeit durch die Verwendung deskriptiver Sprachschnittstellen (z.B. SQL) nicht offensichtlich sind und somit nicht direkt angegeben werden können. Wie die einzelnen Anweisungen unter Verwendung von Grundoperationen realisiert werden, ist das Ergebnis einer jeweils durchgeführten Anfrageoptimierung, das vom Optimierer und von den Gegebenheiten der konkreten Systemumgebung abhängig ist. Die von uns vorgeschlagene und in den nächsten Kapiteln näher beschriebene Methode beruht größtenteils auf Informationen, die aktuell in der jeweiligen konkreten Systemumgebung gesammelt werden.

5 Entwurf eines Speicher- und Verhaltensmodells

Um eine einheitliche Grundlage für die Bewertung des Nutzens von Datenbankreorganisationen zu schaffen, wird in diesem Kapitel ein Speicher- und Verhaltensmodell beschrieben (vgl. auch [Dor00]). Das Modell berücksichtigt dabei verbreitete Speicherungsstrukturen und das Verhalten der über diesen Strukturen implementierten Operationen. Die Berücksichtigung des Verhaltens von Operationen ermöglicht auch die Ermittlung des Nutzens von Datenbankreorganisationen bezüglich der Systemleistung. Die Informationen über die physischen Speicherungsstrukturen und deren Zustand werden in einem vereinheitlichten Schema gehalten, das als *eInformationsschema* bezeichnet wird. Es bildet die Basis für die weiteren Betrachtungen zu Nutzen und Kosten von Datenbankreorganisationen. Dabei steht das *„e“* schlicht für *„erweitert“*, um Verwechslungen mit dem Informationsschema der SQL-Norm zu vermeiden. Ein auf dieser Basis entwickeltes Datenbankschema könnte zum Beispiel als Grundlage für ein systemunabhängiges Werkzeug zur Reorganisationsbedarfsanalyse, ähnlich dem in [Wil04] entwickelten Prototyp, verwendet werden. Nach einführenden Bemerkungen in *Abschnitt 5.1* und der Betrachtung möglicher Informationsquellen für das eInformationsschema in *Abschnitt 5.2* wird das Speichermodell in *Abschnitt 5.3* vorgestellt. In *Abschnitt 5.4* werden gängige Operationen und deren Verhalten beschrieben, die auf die Speicherungsstrukturen angewendet werden können. Dazu werden auch Kostenfunktionen zur Abschätzung des mit der Operationsausführung verbundenen Aufwands entwickelt.

5.1 Allgemeine Bemerkungen

Obwohl bezüglich der Leistungsfähigkeit und Funktionalität zwischen einzelnen DBMS-Produkten teils große Unterschiede bestehen, weisen die physische Speicherorganisation und die Implementierung von Grundoperationen (Planoperatoren [HR01]) Ähnlichkeiten auf. Als problematisch erweist sich allerdings die unterschiedliche Begriffswelt der einzelnen Systeme. So kennen z.B. Oracle und Informix den Begriff des *Table Space* (genaue Schreibweise unter Informix *tblspace* [IBM05a]), benutzen ihn allerdings in unterschiedlicher Bedeutung. Während in der Oracle-Begriffswelt ein Table Space ein Bereich zur Speicherung von (auch mehreren) Datenbankobjekten (Tabellen, Indexen) ist, sind *tblspaces* unter Informix mit Segmenten gleichzusetzen. Das Pendant zum Table Space bei Oracle wird in der Begriffswelt von Informix als *dbspace* bezeichnet. *Tabelle 5.1* zeigt als Beispiel die Gegenüberstellung einer Auswahl der in *Kapitel 3* eingeführten Begriffe und deren Verwendung (Bedeutung) bei gängigen DBMS-Produkten sowie – als Vorgriff – im Speichermodell (eInformationsschema).

Die Elemente des Speichermodells finden sich größtenteils in verbreiteten relationalen DBMS. Es ist so strukturiert, dass Konzepte unterschiedlicher Systeme abgebildet werden können. Bieten manche Systeme bestimmte Konzepte nicht an, so ist es trotzdem möglich, die Strukturen des Systems im Modell darzustellen. Zum Beispiel besteht bei einem System ohne Möglichkeiten zur horizontalen

Partitionierung eine Tabelle immer aus genau einer Partition. Bei verbreiteten DBMS-Produkten wird jede Partition üblicherweise in einem eigenen Speicherbereich untergebracht. In den Datenbankkatalogen verbreiteter DBMS-Produkte werden logische Beschreibungsinformationen zu Partitionen (bspw. das Partitionierungsschema) und die Informationen zu den physischen Bereichen zur Abspeicherung der Partitionen (Segmente) getrennt und es werden i.d.R. auch unterschiedliche Begriffe verwendet. Da zwischen Partitionen und den Segmenten, in denen sie untergebracht werden, eine 1:1-Beziehung besteht, erfolgt im Speichermodell eine Zusammenfassung. Dem Begriff der Partition wird dabei der Vorzug gegeben. Dies erscheint für später mögliche Erweiterungen (z.B. um Informationen zum Partitionierungsschema) günstiger.

Zu beachten ist dabei auch, dass es mit dem vorliegenden Modell (natürlich) *nicht möglich* ist, *alle Eigenschaften* der betrachteten Produkte (im Sinne einer strukturellen und funktionalen Obermenge) *abzubilden*. An einigen Stellen werden Annahmen und Festlegungen getroffen, die auch einschränkenden bzw. vereinfachenden Charakter besitzen, um ein unnötiges Aufblähen des Modells zu vermeiden. Im eInformationsschema sind auch nicht alle Informationen enthalten, die üblicherweise in Datenbankkatalogen gespeichert werden, da nicht alle Informationen für unsere Zwecke von Bedeutung sind. Das gilt bspw. für Integritätsbedingungen auf der Schemaebene, Zugriffsrechte etc.

eInformationsschema	DB2	Informix	Oracle	MS SQL-Server
Datenbank	*/*	*/*	Datenbank	*/*
/	Datenbank (Schema)	Datenbank	Schema	Datenbank (Schema)
Tabelle	Tabelle	Tabelle	Tabelle	Tabelle
Index	Index	Index	Index	Index
Cluster	*/*	*/*	Cluster	*/*
Partition	Partition	Fragment	Partition	Partition
Segment	*/*	tblspace	Segment	Partition
Table Space	Table Space	dbspace, blobspace, ...	Tablespace	Dateigruppe
Datei	Container	Chunk	File	File
Extent	Extent	Extent	Extent	Extent
Block	Seite	Seite	Block	Seite

Tabelle 5.1: Begriffsverwendung bei verschiedenen DBMS-Produkten

Einige der im Modell gemachten Aussagen (z.B. zum Verhalten der Operationen) beruhen auf Annahmen, die sich aus der Beobachtung des Verhaltens der Systeme bzw. aus den Beschreibungen in der Systemliteratur ergaben.

5.2 Informationsquellen

Bei der Abschätzung des Nutzens von Datenbankreorganisationen werden die Auswirkungen von Degenerierungen auf die im laufenden Datenbankbetrieb anfallenden *Speicher- und die Verarbeitungskosten* betrachtet. Reorganisationsbedarfs- und -nutzenanalysen müssen sich daher u.a. an aktuellen Informationen über den Zustand der Speicherungsstrukturen orientieren.

Als Informationsquelle bieten sich *Dienstprogramme* (Werkzeuge) zur Überwachung der Systemleistung bzw. zur Prüfung der Integrität von Speicherungsstrukturen (z.B. `Statspack` bzw. das Paket `DBMS_REPAIR` bei Oracle [DW00, Ora03e], `onperf` bzw. `oncheck` bei Informix [IBM05a] oder der `snapshot monitor` bzw. `db2dart` bei DB2 [IBM04, IBM02]) an. Solche Werkzeuge liefern oft auch in Form kurzer Übersichten Informationen über den Zustand der betrachteten Strukturen. Allerdings ist das Format der Ausgaben nicht einheitlich. Weiterhin werden die Angaben in Textform geliefert. Zur Nutzung solcher Informationen muss der Text analysiert werden. Das setzt aber voraus, dass der genaue Aufbau der Übersichtstexte bekannt ist. Deshalb erscheinen solche Dienstprogramme zur Informationsgewinnung weniger geeignet.

Eine wesentlich umfassendere und vielversprechendere Informationsquelle stellt die direkte Nutzung der Informationen aus dem *Datenbankkatalog* dar, aus dem auch die Dienstprogramme einen Teil der benötigten Informationen gewinnen. Neben den die Modellebene betreffenden Informationen werden dort auch statistische Informationen gespeichert, die primär zum Zweck der kostenbasierten Anfrageoptimierung [Ioa96] gesammelt werden. Diese Statistikdaten enthalten für unsere Zwecke meist ausreichende Informationen über den Zustand physischer Speicherungsstrukturen und stellen damit einen erheblichen Teil der für Reorganisationsbedarfs- und -nutzenanalysen relevanten Informationen dar. Allerdings ist hier zu berücksichtigen, dass die im Katalog gehaltenen Statistikdaten aus Performance-Gründen oft nicht vollständig automatisch und zeitnah gepflegt werden. Zum Sammeln aktueller Statistikinformationen existieren meist spezielle Anweisungen (z.B. `ANALYZE ...` bei Oracle, `UPDATE STATISTICS` bei Informix oder `runstats` bei DB2), die dann vor der Auswertung der Informationen aus dem Datenbankkatalog ausgeführt werden müssen. DBMS-Produkte bieten üblicherweise auch zu Informationen über den Zustand der im Arbeitsspeicher gehaltenen Strukturen des Systems (z.B. Pufferstatistiken, Pufferinhalte, aktive Sitzungen usw.) eine relationale Schnittstelle. Bei Oracle werden die Sichten als sog. *dynamische Performance-Views* [Ora03a] bzw. bei Informix als *System Monitoring Tables* [IBM05b] bezeichnet. Diese stehen zur Laufzeit als Teil des Katalogs zur Verfügung. Ein großer Vorteil der im Datenbankkatalog gehaltenen Informationen ist, dass sie in relationaler Form vorliegen und somit größtenteils mit normalen SQL-Sprachmitteln abgefragt und verarbeitet werden können. Ein Eingriff in das DBMS bzw. die genaue Kenntnis interner Strukturen unterschiedlicher Systeme sind nicht erforderlich. Deshalb basieren unsere Betrachtungen auch auf Katalogdaten.

In [Hel01, Wil04] wird beispielhaft am DBMS-Produkt Oracle die Transformation der Statistikdaten aus dem konkreten Katalog eines DBMS-Produkts in das

eInformationsschema dargestellt. Leider unterscheidet sich die Struktur der Datenbankkataloge bei den einzelnen Systemen deutlich [SSP+00]. Deshalb müssen für unterschiedliche Produkte jeweils spezielle Transformationsschichten geschaffen werden. Wird das eInformationsschema über Sichten realisiert, sind „lediglich“ aktuelle Informationen über den Zustand physischer Speicherungsstrukturen verfügbar. In der konkreten Implementierung werden die Daten des eInformationsschemas in eigenen Tabellen gespeichert. Dadurch besteht die Möglichkeit, mehrere Datenbankzustände (z.B. mit einem Zeitstempel versehen) festzuhalten. Daraus können bei Bedarf auch Informationen über die Entwicklung von Degenerierungen gewonnen werden, die als Basis für Prognosen (z.B. für Reorganisationsintervalle) verwendet werden können.

5.3 Elemente des Speichermodells - eInformationsschema

Ein Teil der für Reorganisationsbedarfs- und -nutzenanalysen benötigten Informationen kann aus dem SQL-Normkatalog [ISO03a, ISO03b] über die Sichten des Informationsschemas (Norm: Information Schema) gewonnen werden. Diese Sichten stellen eine standardisierte Schnittstelle zu den in den jeweiligen Datenbankkatalogen enthaltenen Informationen auf der Modellebene dar. Deshalb wird hier auch der Teil der Norm berücksichtigt, der in Zusammenhang mit der betrachteten Problematik steht. Ein Vorteil der Orientierung am Normkatalog ist dessen Unabhängigkeit von konkreten DBMS-Produkten. Da die SQL-Norm physische Implementierungsaspekte aber bewusst nicht berücksichtigt, sind die Möglichkeiten begrenzt. Es muss ein um Informationen zur physischen Abspeicherung erweitertes Speichermodell entworfen werden. Hier soll ein Überblick über die im eInformationsschema enthaltenen Elemente und deren Verbindung zu Elementen der SQL-Norm gegeben werden.

Abbildung 5.1 zeigt die relevanten Elemente des SQL-Normkatalogs und deren Beziehungen untereinander in Form eines Entity-Relationship-Diagramms.

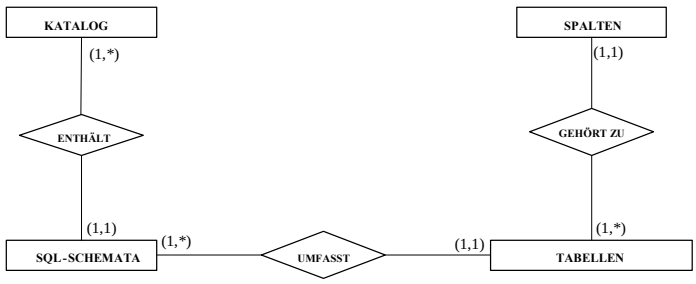


Abbildung 5.1: Darstellung der relevanten Elemente des SQL-Normkatalogs

Eine *SQL-Umgebung* (Norm: SQL-environment) stellt die umfassendste Einheit dar. Sie besteht aus einer Instanz eines DBMS, welche eine Menge von Katalogen verwaltet. Diese *Kataloge* (Norm: catalogs) besitzen einen Namen und stellen benannte Mengen von Schemata dar. Über *Schemata* (Norm: SQL-schemas) können Kataloge in separate Namens- und Verwaltungsräume unterteilt werden. Dynamisch

generierte Schemaelemente (Views) und aktive Elemente (z.B. Trigger) werden bei der Behandlung der Reorganisationsproblematik nicht berücksichtigt. Ein Schema besteht hier grundsätzlich aus einer Menge von *Tabellen* (Norm: Tables), welche nach bestimmten Kriterien in dem jeweiligen Namens- und Verwaltungsraum zusammengefasst wurden. Bei der Transformation ins eInformationsschema werden aus der Sicht TABLES des SQL-Normkatalogs nur die Einträge berücksichtigt, die Informationen über Tabellen zur Datenspeicherung (Norm: Base tables) enthalten. Eine Tabelle wird durch eine Menge von *Spalten* (Norm: columns) gebildet. Jede Spalte besitzt einen Datentyp. Die SQL-Norm unterscheidet in *predefined, constructed* und *user-defined types*. Für Reorganisationsbedarfsanalysen sind allerdings viele der typspezifischen Informationen (z.B. Genauigkeit von numerischen Werten, verwendeter Zeichensatz, Genauigkeit von Zeitangaben) nicht von vorrangigem Interesse. Informationen über die interne Abspeicherung von Daten (z.B. Länge eines Datenwerts) der entsprechenden Typen sind in der Norm wiederum nicht enthalten, da sie abhängig von der Implementierung des jeweiligen DBMS-Produkts sind. Bei einigen Datentypen (bspw. Zeichenketten) ist die Länge einer Spalte nicht nur vom Typ abhängig, sondern auch von der konkreten Spaltendefinition. Deshalb sind Datentypen im eInformationsschema nicht enthalten. Die bei Reorganisationsbedarfsanalysen benötigten Längeninformationen werden dort einfach direkt bei den Beschreibungsdaten der jeweiligen Spalten gespeichert.

Abbildung 5.2 zeigt die für die Reorganisationsbedarfsanalyse nötigen Elemente des eInformationsschemas und deren Beziehungen untereinander in Form eines Entity-Relationship-Diagramms. Nach den üblichen Transformationsregeln lässt sich aus dem E/R-Diagramm ein relationales Schema ableiten, dessen Elemente hier kurz beschrieben werden. Eine spätere Erweiterung sollte kein Problem darstellen. Dabei wurde die in *Kapitel 3* eingeführte Terminologie weitgehend übernommen. Eine detaillierte Übersicht über die im Speichermodell des eInformationsschemas enthaltenen Elemente findet sich in *Anhang A.*

Als *Datenbank* soll hier, analog zur üblichen Definition, eine unter der Verwaltung eines DBMS stehende Menge von Daten bezeichnet werden. Datenbanken werden im eInformationsschema, im Unterschied zur SQL-Norm, allerdings nicht in weitere Namensräume (Schemata) unterteilt. Dies entspricht auch der von einigen DBMS-Produkten (z.B. Informix, MS-SQL-Server) verwendeten Auffassung, bei der Datenbanken den Schemata aus der Norm entsprechen. Aber auch bei Systemen, die Konzepte zur Unterteilung von Datenbanken in Schemata bieten, ist für unsere Zwecke eine solche Unterteilung nicht sinnvoll. Dies liegt u.a. daran, dass

- Datenbankobjekte einer zu reorganisierenden Einheit (z.B. eines Table Space) unterschiedlichen Schemata angehören können und trotzdem gemeinsam betrachtet werden müssen oder
- Datenbankadministratoren typischerweise für die Wartung aller in einer Datenbank enthaltenen Objekte, unabhängig von deren Schemazugehörigkeit, verantwortlich sind.

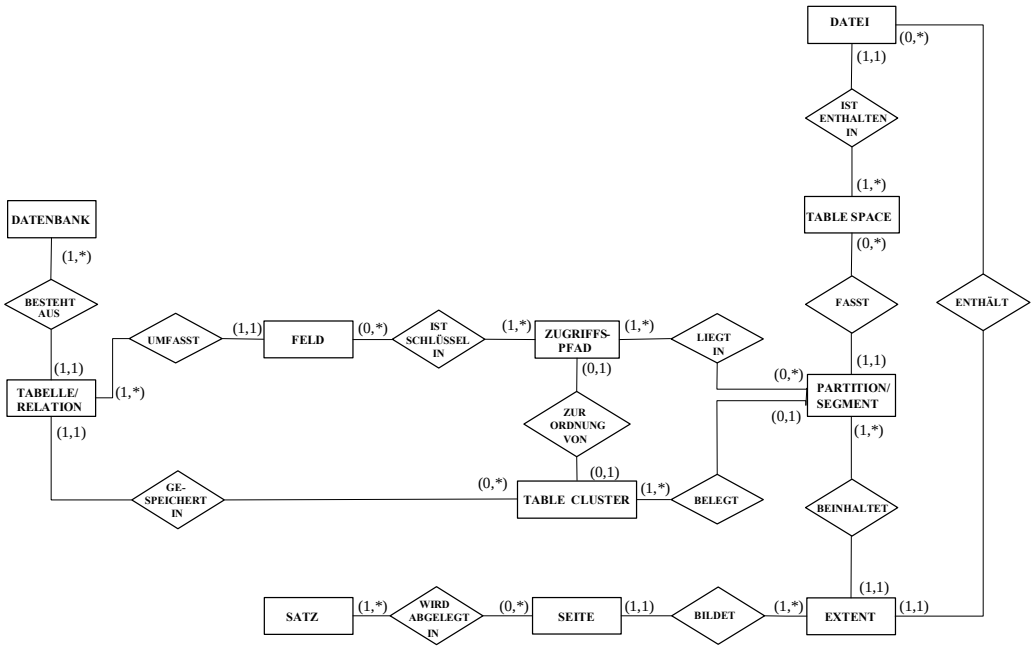


Abbildung 5.2: Darstellung der Elemente des Speichermodells

Wichtig ist bei der Abbildung von Informationen aus den Datenbankkatalogen der unterschiedlichen Systeme lediglich, dass gleichnamige Datenbankobjekte unterschiedlicher Namensräume trotzdem eindeutig identifiziert werden können. Deshalb verwenden die Systeme üblicherweise anstelle von Namen intern auch numerische Identifikatoren. Dies wird auch im eInformationsschema so angewendet.

Als beschreibende Eigenschaften von Datenbanken werden hier vor allem Größen angesehen, welche systemweit gelten (Blockgröße bzw. Länge der Verwaltungsinformationen je Block, Länge von ROWIDs usw.). Diese Informationen befinden sich normalerweise nicht in den Datenbankkatalogen, da sie mehr oder weniger vom verwendeten DBMS und seinen Konfigurationsmöglichkeiten ”vorgegeben” sind und bereits beim Zugriff auf den Katalog bekannt sein müssen. Sie sind allerdings notwendig, um anhand des Modells Reorganisationsbedarfsanalysen für unterschiedlichste DBMS in unterschiedlichen Konfigurationen durchführen zu können.

Eine Datenbank besteht aus einer Menge von *Tabellen* (Relationen). Eine Tabelle besteht aus einer festen Menge von *Feldern* (Spalten) und einer variablen Anzahl von *Sätzen* (Tupeln). Abhängig vom Typ der einzelnen Felder, der u.a. die Länge des für die Ablage des Attributwerts notwendigen Speicherplatzes angibt, können zwei Typen von Tabellen unterschieden werden:

- Tabellen mit *Sätzen fester Länge*
  Bei diesen Tabellen haben alle Sätze die gleiche Länge, d.h. alle Felder haben Datentypen fester Länge.
- Tabellen mit *Sätzen variabler Länge*
  Bei Tabellen, deren Sätze variable Länge haben, kann zur Abschätzung des benötigten Speicherplatzes im Prinzip die gleiche Vorgehensweise wie bei Sätzen

fester Länge verwendet werden. Der Unterschied liegt allerdings darin, dass mit durchschnittlichen bzw. geschätzten Werten für die Länge dieser Felder gearbeitet werden muss.

Für einen möglichst direkten Zugriff auf einzelne Sätze bzw. Satzmengen können bestimmte Felder zum Aufbau von *Zugriffspfaden* (Indexen) verwendet werden. Abhängig von den Datentypen der im Indexierungsschlüssel enthaltenen Felder können Indexe mit Schlüsseln fester oder variabler Länge unterschieden werden. Auch hier werden bei Schlüsseln variabler Länge Durchschnittswerte verwendet. Weiterhin wird der Typ eines Index (B^*-Baum, Cluster-Index, Hash usw.) festgehalten.

Tabellen werden in sog. *Table Clustern* gespeichert. Um eine tabellenübergreifende Clusterung zu realisieren, können in einem solchen Table Cluster auch mehrere Tabellen untergebracht werden. Bei tabellenübergreifender Clusterung erfolgt die Zuordnung der Sätze der einzelnen Tabellen über spezielle Zugriffspfade.

Ein Table Cluster wird in einer oder mehreren *Partitionen* abgespeichert. Jede Partition wird physisch in einem *Segment* untergebracht. Von Interesse sind für das eInformationsschema hauptsächlich die Informationen zum Zustand der Segmente. Wegen der bestehenden 1:1-Beziehung zwischen Partitionen und Segmenten werden beide Elemente zusammengefasst. Für eventuelle spätere Erweiterungen, bspw. um Informationen zum Partitionierungsschema (also Informationen der logischen Ebene), mit denen u.a. die Einhaltung erwünschter Datenverteilungen überwacht werden kann, wird dem Begriff *Partition* der Vorzug gegeben. Wenn Tabellen oder Indexe nicht partitioniert werden, werden sie jeweils in einer Partition untergebracht. Im eInformationsschema kann aber auch eine gemischte Speicherung von Tabellen- und Indexinformationen (wie bspw. bei Informix und DB2 möglich) abgebildet werden.

Die Partitionen werden in *Table Spaces* untergebracht. Ein Table Space ist eine logische Speichereinheit des Modells, die eine oder mehrere *Dateien umfasst.* Eine Datei entspricht einer auf Betriebssystemebene ansprechbaren Einheit (z.B. eine Datei im Dateisystem oder ein Raw Device). Der Speicherplatz für einzelne Partitionen wird in Form von *Extents* reserviert. Dabei werden zwei Arten von Extents unterschieden:

- *First Extents* werden jeweils beim Anlegen eines Datenbankobjekts reserviert.
- Muss für eine Partition neuer Speicher reserviert werden, so werden weitere *Next Extents* reserviert.

Die Größe von First und Next Extents kann unterschiedlich sein. Weiterhin kann für Next Extents oft ein Faktor angegeben werden, um den ein Next Extent gegenüber seinem Vorgänger vergrößert wird. Ein Extent besteht wiederum aus mehreren physisch zusammenliegenden *Blöcken*. Innerhalb der Blöcke werden die einzelnen Sätze (Tupel) oder Teile von Sätzen der Tabelle gespeichert. Da Reorganisationsvorgänge *innerhalb* von Blöcken (wie das Zusammenführen von freiem Speicher) i.d.R. implizit und mehr oder weniger sofort vom DBMS veranlasst werden, werden individuelle Informationen über einzelne Blöcke und die darin gespeicherten Sätze später vernachlässigt. Wichtige Beschreibungsinformationen

über Sätze können auch als Eigenschaften anderer Elemente angesehen werden. So lässt sich z.B. die Größe eines Satzes aus dem Speicherbedarf der Spalten einer Tabelle ermitteln. Die Anzahl Sätze kann als Eigenschaft von Partitionen aufgefasst werden.

5.4 Verhaltensmodell

Das Verhaltensmodell beschreibt die auf die im Speichermodell enthaltenen Elemente angewendeten Operationen. Dabei werden Such- und Änderungsoperationen betrachtet. Die dargestellten Implementierungen sind an Vorgehensweisen angelehnt, die in der Literatur ([Bur98, EN02, IBM05a, Ora03b]) beschrieben werden. Stellenweise beruht das beschriebene Verhalten auch auf Annahmen, die durch die Beobachtung der Systeme nahe gelegt werden. Die verwendeten Begriffe basieren zu einem Teil auf [BG82]. Im Rahmen von Untersuchungen am Lehrstuhl [Gan05, Wie05] wurde das Verhalten von Join-Operationen, Sortieroperationen und Änderungsoperationen modelliert, und es wurden Kostenfunktionen zur Abschätzung des bei der Ausführung dieser Operationen anfallenden I/O-Aufwands abgeleitet. Gegenstand der Betrachtungen war auch die Abschätzung der Selektivität von Anfragen, die besonders auch bei der Ermittlung des Aufwands zur Abarbeitung von Join-Operationen von Bedeutung ist.

In [Kli05] werden das Verhalten von Zugriffsoperationen untersucht und Kostenfunktionen zur Schätzung des anfallenden I/O-Aufwands abgeleitet. In umfangreichen Messungen wurde die Gültigkeit der Kostenfunktionen am Beispiel des DBMS-Produkts Oracle in den Versionen 9i und 10g nachgewiesen. Die in [Gan05, Kli05, Wie05] abgeleiteten Kostenfunktionen basieren alle auf dem Speichermodell des eInformationsschemas und den dort verwendeten Formelzeichen. Überprüfungen der Kostenfunktionen wurden zwar anhand eines konkreten DBMS-Produkts vorgenommen, trotzdem lassen sich die Ergebnisse zu großen Teilen auch auf andere Systeme übertragen. Die von Oracle verwendete Implementierung von Operationen weicht nur wenig von in der Literatur beschriebenen Verhaltensweisen ab. Zu einigen wichtigen Operationen, auf die in der vorliegenden Arbeit größtenteils auch weiterhin zurückgegriffen wird, werden neben kurzen Beschreibungen zum Verhalten auch Kostenfunktionen zur näherungsweisen Berechnung des bei der Operationsausführung anfallenden I/O-Aufwands mit angegeben. Weitere Beschreibungen und Kostenfunktionen sind in *Anhang C* aufgeführt.

Schätzungen der während der Abarbeitung von Operationen anfallenden Kosten werden auch im Rahmen der kostenbasierten Anfrageoptimierung angestellt. Eine einfache Variante zur Ermittlung des bei der Abarbeitung von Datenbankoperationen anfallenden Aufwands wäre die Nutzung der *Kostenschätzungen, die vom Anfrageoptimierer* geliefert werden. Problematisch ist hier allerdings, dass diese bei vorhandenen DBMS-Produkten „von außen“ kaum im Detail nachvollziehbar sind. Die Hersteller legen die im Anfrageoptimierer verwendeten Berechnungsvorschriften i.d.R. nicht offen. Unklar bleibt insbesondere, wie vorhandene Degenerierungen der Speicherungsstrukturen im jeweiligen Kostenmodell berücksichtigt werden (oft vermutlich nicht). Wie z.B. die Anzahl migrierter Tupel in die Kostenermittlung für

einen Datenzugriff über einen Index mit einfließt oder inwiefern der aktuelle Grad der physischen Sortierung von Tupeln nach dem Ordnungskriterium des Index (Clustering Factor) bei Index Scans berücksichtigt wird, bleibt unklar, ebenso wie verwendete Annahmen über die Einflüsse von Blockpufferung und vorausschauendem Lesen, die teilweise in die Kostenermittlungen einfließen. Im Rahmen der Anfrageoptimierung werden i.d.R. auch keine Kostenschätzungen für Änderungsoperationen angestellt. Dies ist größtenteils sicherlich auch darin begründet, dass absolute Kostenwerte, wie sie idealerweise für Quantifizierungen des Nutzens von Datenbankreorganisationen benötigt werden, für die Anfrageoptimierung nicht so stark von Bedeutung sind [Mak03]. Dort geht es vorrangig darum, den Ausführungsplan zu finden, der die im Vergleich niedrigsten Kosten erwarten lässt, wie hoch diese absolut auch immer sein mögen. Bei der in [Dor03] beschriebenen Vorgehensweise führt die Verwendung der Kostenschätzungen des Anfrageoptimierers unter ungünstigen Umständen zu fehlerhaften Berechnungen von Workload-Anteilen, mit denen ermittelte Mehraufwandswerte bei der Quantifizierung des Nutzens von Datenbankreorganisationen (bezogen auf die Systemleistung) dann gewichtet werden.

Die in verschiedenen Beiträgen in der Literatur [Dun02, KE04, Mak03, SAC+79] aufgeführten Kostenfunktionen berücksichtigen in Speicherungsstrukturen vorhandene Degenerierungen ebenfalls oft nicht. Abhilfe schafft hier die Verwendung eines *erweiterten (eigenen) I/O-Kostenmodells*, in das die Quantifizierung der Auswirkungen vorhandener Degenerierungen einbezogen ist. Das Modell wurde teilweise im Rahmen der oben genannten Arbeiten entwickelt. Unter Nutzung aktueller Statistikdaten lassen sich damit die bei der Abarbeitung einer Workload anfallenden Blockzugriffe nachvollziehbar abschätzen. Die vorgenommenen Erweiterungen können bei Bedarf in existierende Kostenmodelle von DBMS-Produkten eingebaut und die Funktionen zur Mehraufwandsberechnung hinzugefügt werden, wodurch auch systemspezifische Eigenschaften besser berücksichtigt werden können.

Im Rahmen der Quantifizierung des Nutzens von Datenbankreorganisationen (*Kapitel 6*) werden I/O-Kosten betrachtet, weil diese einen erheblichen Anteil an dem bei Datenzugriffen anfallenden Gesamtaufwand ausmachen und mit Datenbankreorganisationen maßgeblich beeinflusst werden können. Den Nutzen einer Datenbankreorganisation bezüglich der Verarbeitungskosten stellt die prozentuale Verringerung des zur Workload-Abarbeitung anfallenden I/O-Aufwands dar.

Das Verhaltensmodell gliedert sich in zwei Gruppen von Operationen. Die Zugriffsoperationen bilden die erste Gruppe (*Abbildung 5.3*).

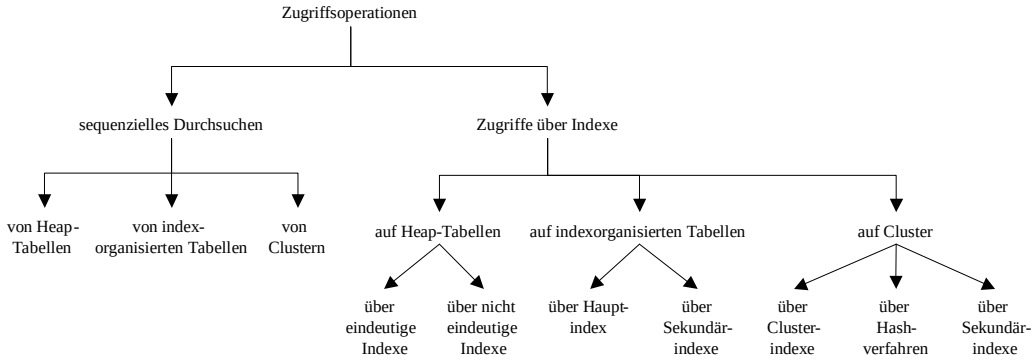


Abbildung 5.3: Übersicht über die Zugriffsoperationen

Die zweite Gruppe wird durch die Update-Operationen gebildet. Eine Übersicht zeigt *Abbildung 5.4*. Da die Implementierung der Operationen für unterschiedliche Speicherungsstrukturen verschieden erfolgen muss, wurde eine Untergliederung bezüglich der Speicherungsstrukturen vorgenommen.

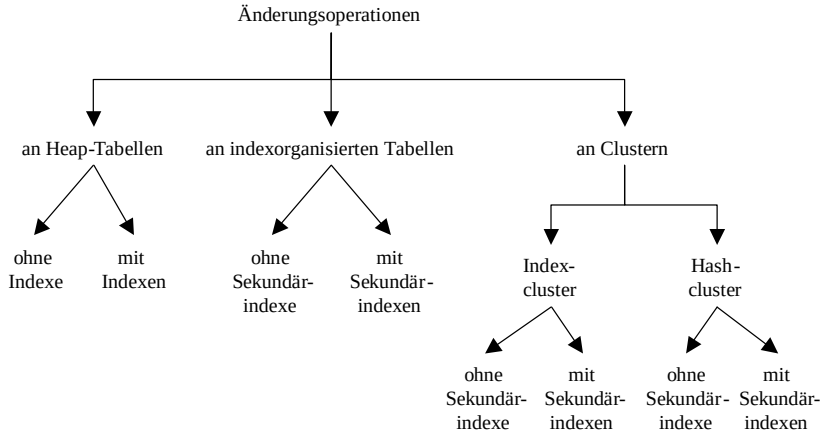


Abbildung 5.4: Übersicht über die Änderungsoperationen

Bei der Modellierung des Verhaltens der Operationen werden hauptsächlich die Eigenschaften betrachtet, die in engem Zusammenhang mit der Reorganisationsproblematik stehen. Bei den Kostenschätzungen werden nur die während der Ausführung der Operationen anfallenden I/O-Kosten berücksichtigt. Bezüglich des Auftretens von Degenerierungen, Schlüsselwerten usw. werden Gleichverteilungen angenommen. Die Berücksichtigung von „Schieflagen“ bei der Verteilung von Degenerierungen würde Erweiterungen der Statistiken erfordern und zu einer deutlichen Erhöhung des Aufwands für die Statistikerstellung und des Umfangs jener Daten führen. Hier müsste abgewogen werden, ob der zu erwartende Genauigkeitsgewinn diesen Aufwand rechtfertigt.

In den weiteren Ausführungen dieser Arbeit wird weiterhin größtenteils davon ausgegangen, dass Tabellen und Indexe nicht weiter unterteilt (partitioniert) wurden. Bei Abweichungen von dieser Annahme wird an der entsprechenden Stelle darauf hingewiesen.

5.4.1 Zugriffsoperationen

5.4.1.1 Sequenzielles Suchen

Beim sequenziellen Suchen (*Sequential Scan*) werden nacheinander alle belegten Datenblöcke (unabhängig von ihrem Füllungsgrad) eines Segments durchsucht (*Abbildung 5.5*). Bei als Heap organisierten Segmenten werden nacheinander die einzelnen Extents in der Reihenfolge abgearbeitet, in der sie reserviert wurden. Innerhalb der Extents werden die einzelnen Blöcke entsprechend der physischen Speicherreihenfolge verarbeitet. In den Verwaltungsdaten zu einem Segment (Segment Header) befindet sich üblicherweise auch eine sogenannte *High Water Mark* (HWM), über die der letzte mit Daten belegten Blöcke erkannt wird.

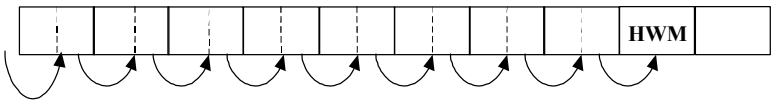


Abbildung 5.5: Sequenzielles Suchen

Die für eine solche sequenzielle Suche anfallenden logischen I/O-Kosten (C_{LScan} - *Gleichung 5.1*) entsprechen der Anzahl der belegten Datenblöcke (P_{USED}), die aus den entsprechenden Kataloginformationen (im eInformationsschema als Eigenschaft der Partitionen) abgelesen werden kann.

$$C_{LScan} = P_{USED}$$ Gleichung 5.1

Soll hier eine mögliche horizontale Partitionierung von Tabellen berücksichtigt werden, so ist bei der Errechnung der Kosten die Summe der belegten Blöcke aller Partitionen zu bilden, die für die Anfrage relevante Daten enthalten können. Dabei können u.U. Partitionen von vornherein von der Suche ausgeschlossen werden, weil aus der Selektionsbedingung und dem Partitionierungsschema erkennbar ist, dass sie keine relevanten Daten enthalten können.

In *Gleichung 5.1* wird der Einfluss der Datenbankpufferung nicht mit einbezogen. Von der Annahme ausgehend, dass der Aufwand für logische I/O-Operationen im Vergleich zum für physische I/O-Operationen anfallenden Aufwand vernachlässigt werden kann, ergibt sich *Gleichung 5.2*, in der der Pufferungsgrad (*Abschnitt 2.1*) der Blöcke des Segments (P_{BRS}) bei der Schätzung der Kosten (C_{PScan}) berücksichtigt wird.

$$C_{PScan} = P_{USED} \cdot (1 - P_{BRS})$$ Gleichung 5.2

Zur weiteren Beschleunigung der Zugriffe auf Sekundärspeichermedien beim sequenziellen Suchen nutzen Datenbank-Management-Systeme Prefetching-Mechanismen. Dabei werden mit einem Datenträgerzugriff mehrere Blöcke gelesen oder geschrieben. Bei der Nutzung solcher Mechanismen kann allerdings keine Rücksicht darauf genommen werden, ob sich einzelne benötigte Datenblöcke bereits im Puffer befinden. Die Anzahl Blöcke, die bei einem Zugriff auf einen Sekundärdatenträger gelesen werden, gibt die Prefetch-Rate (DB_{PRE}) an. Bei Berücksichtigung von Prefetching-Mechanismen ergibt sich für die Kostenschätzung *Gleichung 5.3*.

$$C_{P_{ScanPREF}} = \left\lceil \frac{P_{USED}}{DB_{PRE}} \right\rceil$$ Gleichung 5.3

In Abbildung 5.5 und der zugehörigen Kostenermittlung (*Gleichung 5.1*) werden eventuell vorhandene Auslagerungszeiger auf migrierte Tupel nicht verfolgt. Die entsprechenden Sätze werden beim Verarbeiten des Blocks gelesen, in den jeweils ausgelagert wurde. Das ist unproblematisch, da im relationalen Datenmodell die Einhaltung bestimmter Reihenfolgen von Tupeln nicht zwingend gewährleistet wird. Beobachtungen des Verhaltens von Systemen haben gezeigt, dass zumindest beim DBMS-Produkt Informix Auslagerungszeiger beim sequenziellen Suchen verfolgt werden. Damit werden die Tupel bei Vorhandensein eines Clustered Index in der Reihenfolge geliefert, in der sie beim letzten Ordnen nach dem Sortierkriterium des Indexschlüssels abgespeichert wurden. Das Verfolgen der Auslagerungszeiger führt durch die damit verbundenen „Sprünge“ (*Abbildung 5.6*) zu einer Erhöhung des I/O-Aufwands, der mit *Gleichung 5.4* abgeschätzt werden kann.

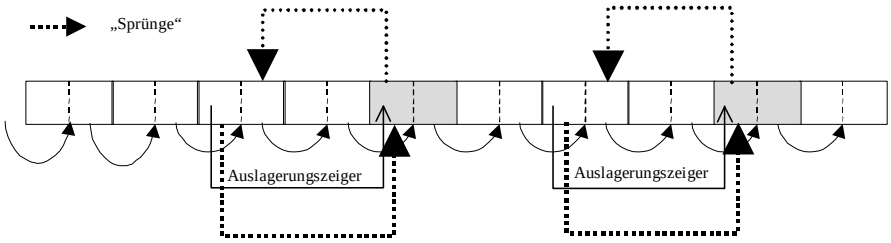


Abbildung 5.6: Sequenzielles Suchen mit Verfolgung von Auslagerungszeigern

Im Beispiel in Abbildung 5.6 werden in jedem Block zwei Sätze untergebracht (angedeutet durch die unterbrochenen Linien). Die grau hinterlegten Blöcke enthalten jeweils einen ausgelagerten Satz.

$$C_{L_{ScanOVFL}} = P_{USED} + 2 \cdot P_{OVFL}$$ Gleichung 5.4

Werden hier die Einflüsse der Datenbankpufferung berücksichtigt, ergibt sich *Gleichung 5.5*.

$$C_{P_{ScanOVFL}} = \left(P_{USED} + P_{OVFL}\right) \cdot \left(1 - P_{BRS}\right)$$ Gleichung 5.5

Hier fließt die Annahme ein, dass der den Auslagerungszeiger enthaltende Datenblock nach dem Lesen des eigentlichen Datensatzes noch im Puffer gehalten wird. Damit fällt für den „Rücksprung“ lediglich ein *logischer* Blockzugriff an.

Auch Segmente, die „reine“ Indexe oder indexorganisierte Tabellen enthalten, können sequenziell durchsucht werden. Derartige Operationen sollen in Anlehnung an den bei Oracle verwendeten Begriff als *Index Fast Full Scan* bezeichnet werden. Index Fast Full Scans können zum vollständigen Durchsuchen aller Blöcke auf der Blattebene einer indexorganisierten Tabelle angewendet werden oder wenn zur Ermittlung des Ergebnisses einer (Teil-)Abfrage lediglich die Werte des jeweiligen Indexschlüssels ausreichend sind (z.B. bei der Anwendung von Aggregatfunktionen). Beim Index Fast Full Scan werden alle Blöcke des Indexsegments gelesen (müssen gelesen werden), obwohl eigentlich nur die Blöcke, die die Blattknoten und die

Blöcke, die die Wegweiserinformationen zum ersten Blattknoten (also zum „linken Rand des Baums“) enthalten, benötigt werden. Wird lediglich gezielt auf die benötigten Blöcke zugegriffen, so müssen die Zugriffe in vielen kleinen Einheiten erfolgen. Durch die Anwendung von Prefetching-Mechanismen bei Index Fast Full Scans wird die Zahl der I/O-Operationen reduziert. Die dadurch entstehenden Geschwindigkeitsvorteile gleichen den durch das „Aussortieren“ der nicht benötigten Blöcke entstehenden Mehraufwand i.d.R. mehr als aus.

Auch beim sequenziellen Durchsuchen von Segmenten, in denen die Daten mehrerer Tabellen *tabellenübergreifend geclustert* abgespeichert sind, muss auf alle belegten Blöcke des Segments zugegriffen werden. Ein selektiver Zugriff lediglich auf die zu einer bestimmten Tabelle gehörenden Datensätze ist wegen der bewussten „Datenvermischung“ nicht möglich. Deshalb ist der Aufwand für das sequenzielle Durchsuchen einzelner Tabellen bei tabellenübergreifender Clusterung höher als bei der physisch getrennten Speicherung der einzelnen Tabellen.

Zur Abschätzung des anfallenden I/O-Aufwands für Index Fast Full Scans und für sequenzielles Durchsuchen von Clustern gelten *Gleichung 5.2* bzw. *Gleichung 5.3*.

5.4.1.2 Zugriffe über Indexe

Zunächst werden Zugriffe über Indexe betrachtet, die als B*-Bäume organisiert sind. Es wird angenommen, dass die Suchoperationen erfolgreich verlaufen, d.h. dass zu den Suchschlüsselwerten auch Daten gefunden werden. Bei erfolglosem Suchen entfällt jeweils der Aufwand für die Zugriffe auf die Datensätze. Zur Vereinfachung werden in diesem Abschnitt nur eindeutige (unique) Indexe berücksichtigt. Dies wird auch durch das „*U*“ am Ende der Bezeichnung der Kostenwerte ausgedrückt. Erweiterte Kostenfunktionen, die auch für nicht eindeutige Indexe anwendbar sind, können *Anhang C* entnommen werden.

Für den Zugriff auf einen in einer als Heap organisierten Tabelle gespeicherten Datensatz über einen als B*-Baum organisierten Index (*Index Lookup*) kann die anfallende Anzahl logischer Blockzugriffe mit der naheliegenden, auch in der Literatur (z.B. in [EN02, Mak03]) angegebenen *Gleichung 5.6*

$$C_{LILookupU} = \underbrace{I_{LEV}}_{Index} + \underbrace{1}_{Daten} \qquad \text{Gleichung 5.6}$$

abgeschätzt werden, wenn aus den Statistikdaten die Anzahl Ebenen des Index (I_{LEV}) ermittelt werden kann. Um die Auswirkungen migrierter Tupel (Satzauslagerungen) zu berücksichtigen, muss die Gleichung um die Gesamtzahl der in der Partition gespeicherten Sätze (P_{ANZT}) und die Anzahl ausgelagerter Sätze (P_{OVFL}) erweitert werden (*Gleichung 5.7*).

$$C_{LILookupU} = \underbrace{I_{LEV}}_{Index} + \underbrace{1 + \frac{P_{OVFL}}{P_{ANZT}}}_{Datenteil} \qquad \text{Gleichung 5.7}$$

Die Erweiterung von Gleichung 5.7 um die Pufferungsgrade der Datenblöcke und der einzelnen Indexebenen führt zu *Gleichung 5.8*, mit der die Anzahl notwendiger physischer Blockzugriffe ($C_{PILookupU}$) abgeschätzt werden kann.

$$C_{PILookupU} = \underbrace{\sum_{i=1}^{I_{LEV}} \left(1 - I_{BRi}\right)}_{Indexteil} + \underbrace{\left(1 + \frac{P_{OVFL}}{P_{ANZT}}\right) \cdot \left(1 - P_{BR}\right)}_{Datenteil}$$

Gleichung 5.8

Bei der Berechnung des Aufwands zum Durchsuchen des Indexteils wird für jede Ebene (i) des Indexbaums, beginnend bei der Wurzel, ggf. ein eigener Pufferungsgrad berücksichtigt (I_{BRi}), da bei Operationen über Indexen physische I/O-Vorgänge meist nur für Blöcke auf der Blattebene anfallen. Die Blöcke der inneren (höheren) Ebenen werden mit hoher Wahrscheinlichkeit im Datenbankpuffer vorgehalten. Ist eine Ermittlung von ebenenbezogenen Pufferungsgraden für Indexe in einer konkreten Systemumgebung nicht möglich, so können hier zumindest auch entsprechende Annahmen (wie bspw. in [KE04]) in die Berechnungen einfließen. Der Pufferungsgrad der Datenblöcke (P_{BR}) kann von dem in *Abschnitt 5.4.1.1* verwendeten Pufferungsgrad abweichen, wenn das konkrete DBMS unterschiedliche Ersetzungsstrategien für Datenzugriffe über Indexe und sequenzielles Suchen verwendet.

Die Verhaltensweise von Bereichsanfragen, die über Indexe abgewickelt werden (*Index Range Scan*) zeigt *Abbildung 3.10.* Die angegebenen Gleichungen gelten für eindeutige und nicht eindeutige Indexe. Die Anzahl notwendiger Blockzugriffe (*Gleichung 5.9*) ergibt sich hier aus der Summe der Anzahl Indexebenen (ohne Blattebene), der Anzahl Blätter des Indexbaums, die entlang der Verkettung durchlaufen werden müssen (I_{LEAF} bzw. ein Teil davon) und der Anzahl Tupel (P_{ANZT}) im Suchbereich, da für jeden Tupelzugriff ein Zugriff auf den jeweiligen Datenblock erfolgen muss.

$$C_{LIScan} = \underbrace{\left(I_{LEV} - 1\right)}_{innere\ Ebenen} + \underbrace{\max\left(S \cdot I_{LEAF}, 1\right)}_{Blattebene} + \underbrace{P_{ANZT} \cdot S}_{Datenteil}$$

Gleichung 5.9

S (Selektivität) gibt den Anteil des Suchbereichs an der Gesamtdatenmenge an, über den der Index Scan ausgeführt wird. Durch die Berücksichtigung der durch ausgelagerte Sätze zusätzlich anfallenden Blockzugriffe (z.B. Zugriff Nr. 4 in *Abbildung 3.10*) ergibt sich *Gleichung 5.10*.

$$C_{LIScan} = \underbrace{\left(I_{LEV} - 1\right)}_{innere\ Ebenen} + \underbrace{\max\left(S \cdot I_{LEAF}, 1\right)}_{Blattebene} + \underbrace{\left(P_{ANZT} + P_{OVFL}\right) \cdot S}_{Datenteil}$$

Gleichung 5.10

Eine Erweiterung um Pufferungsgrade führt dann zu *Gleichung 5.11*. Dabei entspricht I_{BRh} dem Pufferungsgrad der Blöcke auf der Blattebene.

$$C_{PIScan} = \underbrace{\sum_{i=1}^{I_{LEV}-1} \left(1 - I_{BRi}\right)}_{innere\ Ebenen} + \underbrace{\max\left(S \cdot I_{LEAF} \cdot \left(1 - I_{BRh}\right), \left(1 - I_{BRh}\right)\right)}_{Blattebene} + \underbrace{\left(P_{ANZT} + P_{OVFL}\right) \cdot \left(1 - P_{BR}\right) \cdot S}_{Datenteil}$$

Gleichung 5.11

In neueren Versionen des DBMS-Produkts Oracle wird eine verbesserte Implementierung des Index Scan eingesetzt. Bei dieser Variante wird versucht, die Zahl der Zugriffe auf Datenblöcke zu reduzieren. Befinden sich mehrere in der Sortierreihenfolge des Index aufeinanderfolgende Sätze in einem Datenblock, so werden diese mit einem Zugriff gelesen[8]. Je besser also die Daten nach dem Ordnungskriterium des Index sortiert sind, desto weniger Zugriffe auf Datenblöcke sind notwendig.

Zur Kostenschätzung werden in diesem Fall Informationen über den Grad der Übereinstimmung der Ordnung eines Index und der Sätze, auf die die Index-Einträge verweisen benötigt. Diese Informationen enthält der sog. *Clustering Factor* (I_{CLUST}). Im eInformationsschema wird der Clustering Factor in einer normalisierten Form (als Prozentwert) geführt. Dabei stehen 100% für eine Übereinstimmung der Sortierung. DBMS (z.B. Oracle oder Informix) speichern oft keinen solchen normalisierten Wert, sondern einen Wert, der zwischen der Anzahl Datenblöcke und der Anzahl Tupel einer Tabelle liegt. Damit liegt die Vermutung nahe[9], dass der Clustering Factor dort angibt, wie oft bei einem Index Scan über die gesamte Tabelle der Datenblock wechselt, in der der nächste zu lesende Satz gespeichert ist (also zwischen „fast gar nicht springen“ und „extremem Springen“ liegt). Dieser Wert wird auch für die Kostenermittlungen benötigt. Die Umrechnung des normalisierten Werts des Clustering Factors in die in *Gleichung 5.12* benötigte Anzahl Blockwechsel (dort über die Größe N_{BW} angegeben) kann mit *Gleichung C.1* erfolgen.

$$C_{PIScan} = \underbrace{\sum_{i=1}^{I_{LEV}-1}\left(1-I_{BRi}\right)}_{\text{innere Ebenen}} + \underbrace{\max\left(S\cdot I_{LEAF}\cdot\left(1-I_{BRh}\right),\left(1-I_{BRh}\right)\right)}_{\text{Blattebene}} + \underbrace{\left(N_{BW}+P_{OVFL}\right)\cdot\left(1-P_{BR}\right)\cdot S}_{\text{Datenteil}}$$

Gleichung 5.12

Bei Zugriffen auf Daten in *indexorganisierten Tabellen* muss unterschieden werden, ob der Zugriff über den Hauptindex, über den die IOT organisiert ist, oder über einen Sekundärindex erfolgt. Wird über den Hauptindex zugegriffen, so muss der Indexbaum von der Wurzel beginnend bis zur Blattebene durchsucht werden. Hinzu kommt noch der Aufwand für Zugriffe auf eventuell vorhandene Überlaufbereiche. Bei Zugriffen über *Sekundärindexe* ist der anfallende I/O-Aufwand vom für den Sekundärindex verwendeten Adressierungskonzept abhängig. Wegen der relativ hohen Wahrscheinlichkeit, dass sich der physische Speicherort von in indexorganisierten Tabellen gespeicherten Daten ändert (nämlich wenn sich der Wert im Indexattribut substanziell ändert), werden dort keine Tupelidentifikatoren verwendet. In Sekundärindexen werden i.d.R. logische Referenzen gespeichert (z.B. der zum entsprechenden Tupel gehörende Schlüsselwert im Hauptindex). Nachdem der Sekundärindex durchsucht wurde, wird die Suche unter Nutzung des dort

[8] Befindet sich der die benötigten Sätze enthaltende Block bereits im Puffer, so fällt hier auch nur ein logischer Blockzugriff an.

[9] Diese Vermutung wird auch durch die Ausführungen in [SM95] zur Anfrageoptimierung in relationalen DBMS gestützt.

gefundenen Schlüsselwerts im Hauptindex fortgesetzt. Für einen Datenzugriff müssen hier zwei Indexe durchsucht werden. Zur Reduzierung des Aufwands speichert Oracle in zu indexorganisierenden Tabellen gehörenden Sekundärindexen sog. Universal ROWIDs (*UROWID*). Diese basieren auf den Schlüsselwerten des Hauptindex und enthalten zusätzlich einen Verweis auf den Block, in dem der zu referenzierende Datensatz beim Anlegen des Sekundärindex gespeichert war. Der Oracle-Statistikwert `PCT_DIRECT_ACCESS` (I_{DIR_ACC} im eInformationsschema) gibt Auskunft über den prozentualen Anteil gültiger Blockverweise. Weitere Erläuterungen und Gleichungen zur Abschätzung der anfallenden I/O-Kosten sind in *Anhang C* aufgeführt.

Sollen alle in einem *Index Cluster* zu einem Cluster-Schlüsselwert gehörenden Tupel gelesen werden, so fallen bei einem Index Cluster die Zugriffe für das Durchmustern des Indexbaums, ein Zugriff auf den Datenblock im Primärbereich und evtl. die Zugriffe für das Durchsuchen der Überlaufbereiche an. Der I/O-Aufwand kann mit der *Gleichung 5.13* abgeschätzt werden.

$$C_{LICLookup} = \underbrace{I_{LEV}}_{Indexteil} + 1 + \underbrace{\frac{I_{CHAINS}}{I_{UNQ}}}_{Überlaufbereich} \qquad \text{Gleichung 5.13}$$

Die Anzahl der Cluster-Schlüsselwerte (I_{UNQ}) und die Gesamtzahl der zum Cluster gehörenden Überlaufblöcke (I_{CHAINS}) können aus den Statistikdaten des Cluster-Index abgelesen werden. Die Berücksichtigung von Pufferungsgraden führt zu *Gleichung 5.14*.

$$C_{PICLookup} = \underbrace{\sum_{i=1}^{I_{LEV}} \left(1 - I_{BR_i}\right)}_{Indexteil} + \underbrace{\left(1 + \frac{I_{CHAINS}}{I_{UNQ}}\right) \cdot \left(1 - P_{BR}\right)}_{Datenteil} \qquad \text{Gleichung 5.14}$$

Wird für den Cluster-Index ein *Hash-Verfahren* verwendet, so entfallen die in Gleichung 5.13 und Gleichung 5.14 enthaltenen Zugriffe zum Durchsuchen des Indexbaums. Unter Berücksichtigung der Einflüsse der Datenbankpufferung ergibt sich *Gleichung 5.15* zur Berechnung des anfallenden I/O-Aufwands.

$$C_{PHCLookup} = \left(1 + \underbrace{\frac{I_{CHAINS}}{I_{HKEYS}}}_{Überlaufbereich}\right) \cdot \left(1 - P_{BR}\right) \qquad \text{Gleichung 5.15}$$

Auch wenn Daten tabellenübergreifend geclustert abgespeichert werden, können die einzelnen Tabellen noch weiter „normal“ indexiert werden. Oracle arbeitet hier mit physischen Referenzen (ROWID). Deshalb verhalten sich Zugriffsoperationen über *Sekundärindexe* wie Operationen für „normale“ als Heap organisierte Tabellen, und es gelten die gleichen Kostenfunktionen.

5.4.2 Änderungsoperationen

Auch bei Änderungsoperationen fallen unterschiedliche Kosten an, wenn sie auf verschiedene Speicherungsstrukturen angewendet werden. Das Verhalten von

Einfüge-, Lösch- und Änderungsoperationen wird hier am Beispiel von als Heap organisierten Tabellen und den zugehörigen B^*-Baum-Indexen beschrieben. Weiterhin werden Gleichungen zur Abschätzung der anfallenden I/O-Kosten entwickelt. Als Maßeinheit wird wieder die Anzahl notwendiger Blockzugriffe verwendet.

Schreiboperationen werden von DBMS-Produkten i.d.R. asynchron ausgeführt. Ein geänderter Block wird dabei nicht sofort nach der Änderung auf den entsprechenden Datenträger geschrieben, sondern verbleibt zunächst im Puffer und wird lediglich zum Schreiben „vorgemerkt". Dazu kann er bspw. in eine sog. *Page Cleaner List* eingetragen werden. Das eigentliche Schreiben der geänderten Blöcke wird dann von separaten Prozessen bzw. Threads (sog. *Page Cleaner Agents* [IBM04]) durchgeführt. Solche Schreiboperationen werden z.B. initiiert, wenn die Anzahl Einträge in der Page Cleaner List einen vorgegebenen Schwellwert übersteigt oder ein bestimmtes Zeitintervall seit der letzten Schreiboperation vergangen ist. Für einzelne Änderungsoperationen wird in den nachfolgenden Betrachtungen angenommen, dass eine Änderungsoperation auch jeweils zum Schreiben der von der Operation geänderten Blöcke führt. Damit ergibt sich durch die asynchrone Ausführung von Schreiboperationen hier zunächst keine Reduzierung des anfallenden Aufwands.

Bei häufigen Änderungen an einzelnen Blöcken, wie sie beispielsweise im Rahmen von Datenbankreorganisationen auftreten, ist aber zu erwarten, dass das Schreiben der jeweiligen Blöcke erst nach mehreren Änderungen erfolgt und sich damit eine Aufwandsreduzierung ergibt. Dieser Fall wird in diesem Abschnitt allerdings zunächst nicht berücksichtigt. Für die mit der Durchführung von Reorganisationsmaßnahmen verbundenen (Massen-)Änderungsoperationen werden in *Kapitel 7* gesonderte Annahmen getroffen.

Im Rahmen von Änderungsoperationen kann auch der Fall auftreten, dass neue Blöcke reserviert werden müssen. Dazu ist ein Zugriff auf die Freispeicherverwaltungsinformationen des jeweiligen Segments notwendig, die geändert und in geänderter Form gespeichert werden müssen. Es wird angenommen, dass die Freispeicherverwaltungsinformationen mit hoher Wahrscheinlichkeit im Puffer gehalten und nach Änderungen im Rahmen asynchron ablaufender Schreiboperationen auf die jeweiligen Datenträger geschrieben werden. Zum Reservieren neuer Blöcke wird in den folgenden Kostenfunktionen jeweils (mit der Wahrscheinlichkeit der Auftretens gewichtet) ein Blockzugriff eingerechnet.

Zur Vereinfachung wird hier weiterhin davon ausgegangen, dass die Satzlänge die Größe des in einem Datenblock maximal zur Datenspeicherung zur Verfügung stehenden Platzes nicht übersteigt (einige DBMS-Produkte, DB2 etwa, fordern dies ohnehin). Der für die Transaktionsprotokollierung (Transaktions-Logging) anfallende Aufwand fließt zunächst nur über Faktoren ($C_{Log...}$) in die Berechnungen ein. In *Abschnitt 5.4.5* wird dieses Thema noch einmal aufgegriffen.

5.4.2.1 Einfügen

Bei der Ausführung von Einfügeoperationen erfolgt das Einfügen von Datensätzen entweder im letzten Block des Datenbereichs oder, wenn die Einfügeoperationen entsprechende Informationen der Freispeicherverwaltung berücksichtigen, in einem Block, der einen hinreichend hohen Anteil freien Speicherplatzes aufweist. Für die Kostenermittlung wird im Folgenden angenommen, dass das Einfügen immer im letzten Datenblock vorgenommen wird, dessen Adresse über die High Water Mark bekannt ist. Dabei ist es allerdings möglich, dass der neue Satz nicht mehr in jenem Block untergebracht werden kann, weil der zur Verfügung stehende Speicherplatz erschöpft ist. Dann muss ein neuer Block reserviert werden. Die Wahrscheinlichkeit, mit der dieser Fall eintritt, kann über der Anzahl Sätze bestimmt werden, die im Mittel in einem Datenblock untergebracht werden können (*TP - Gleichung B.2*). Berechnungsschemata für Speicherplatzabschätzungen sind allgemein bekannt. Die bei den jeweiligen Produkten zu berücksichtigenden Besonderheiten sind meist in der Systemliteratur angegeben. *Anhang B* enthält die Beschreibung von Vorgehensweisen bei Speicherbedarfsabschätzungen für Heap-Tabellen und zugehörige B^*-Baum-Indexe, die an das eInformationsschema angepasst wurden. Dort finden sich auch Berechnungsvorschriften für in diesem Abschnitt verwendete Größen (z.B. *TP, EP* usw.). Beschreibungen der Vorgehensweise und SQL-Skripte zur Speicherplatzabschätzung bei Hash- und Index-Clustern bei Oracle sind bspw. auch in [Sin00] bzw. in *Anhang C* der vorliegenden Arbeit zu finden. Der Aufwand für das Einfügen eines Datensatzes kann mit *Gleichung 5.16* geschätzt werden.

$$C_{InsertDaten} = \underbrace{(1 - P_{BR})}_{\text{Lesen letzter Block}} + \underbrace{\frac{1}{TP}}_{\text{Reservieren neuer Block}} + \underbrace{1}_{\text{Schreibaufwand}} \qquad \text{Gleichung 5.16}$$

Der Aufwand für das Lesen des letzten Blocks ist abhängig vom Grad der Pufferung der Blöcke des Datensegments. Wenn häufig eingefügt wird, befindet sich der Block mit hoher Wahrscheinlichkeit im Puffer. Mit der Wahrscheinlichkeit, dass ein neuer Block belegt werden muss, wird ein weiterer Zugriff zum Reservieren des neuen Blocks ausgeführt. Hinzu kommt der Aufwand für das Schreiben des geänderten Datenblocks. Werden Informationen über die Belegung von Datenblöcken separat (bspw. in Bitmap-Listen) verwaltet, so kann ohne vorherigen Zugriff entschieden werden, ob das Einfügen im bisherigen letzten Block oder in einem neuen Block erfolgen muss.

Neben dem Aufwand für das Einfügen des eigentlichen Datensatzes fällt für die zur Tabelle gehörenden Indexe Aufwand für das Einfügen bzw. Erweitern von Indexeinträgen an. Bei eindeutigen Indexen muss dazu ein kompletter Indexeintrag eingefügt werden. Im einfachsten Fall muss der Baum durchsucht und der neue Eintrag in den entsprechenden Blattknoten eingefügt werden. Anschließend muss der geänderte Blattknoten geschrieben werden.

Allerdings kann beim Einfügen auch ein Aufspalten des jeweiligen Blatts erforderlich werden. Der dabei anfallende zusätzliche Aufwand muss für die Aufwandsschätzungen gleichmäßig auf alle Einfügeoperationen verteilt werden. Wird zunächst angenommen, dass nur Einfügeoperationen auftreten, so fällt der

Zusatzaufwand bei jeder den (bspw. über Parameter wie PCTFREE) vorgegebenen Füllungsgrad der Blattknoten um eins überschreitenden Einfügeoperation an. Die Wahrscheinlichkeit, mit der Knoten aufgeteilt werden müssen, entspricht damit dem reziproken Wert der aus dem vorgegebenen Füllungsgrad resultierenden Kapazität des Knotens plus dem Eintrag, der den Knotenüberlauf verursacht. Die beim Indexaufbau nutzbare Kapazität eines Blattknotens kann über die Länge der Indexeinträge (*LE - Gleichung B.9*) und die Länge des in den Knoten zur Speicherung von Indexeinträgen zur Verfügung stehenden Bereichs bestimmt werden. Die Länge dieses Bereichs ergibt sich aus der Blockgröße (DB_{PS}), abzüglich des Speichers, der für Verwaltungsinformationen des Blocks (DB_{VIP})und des Speichers (DB_{LBA}) für die Verkettung der Blattknoten benötigt wird. Bei nicht eindeutigen Indexen können einerseits neue Einträge hinzugefügt werden, andererseits besteht aber auch die Möglichkeit, dass zu einem bereits existierenden Indexeintrag lediglich der Verweis auf den neuen Datensatz hinzugefügt werden muss. Beim Einfügen eines neuen Datensatzes wird also nicht in jedem Fall ein neuer Indexeintrag erzeugt. Bei der Berechnung der Länge von Indexeinträgen mit Gleichung B.9 wird berücksichtigt, dass je Schlüsselwert mehrere Verweise auf Datensätze gespeichert werden können. Dazu wird das als nahezu konstant bleibend angenommene Verhältnis der Anzahl eindeutiger Schlüsselwerte (I_{UNQ}) zur Gesamtzahl der Sätze der Tabelle (P_{ANZT}) mit in die Berechnungen einbezogen. Bei der Berechnung der Wahrscheinlichkeit, dass ein Blattknoten aufgespalten werden muss, muss dieses Verhältnis deshalb auch berücksichtigt werden.

Werden evtl. stattfindende Löschoperationen (durch die Speicherplatz in Indexknoten frei wird) zunächst vernachlässigt, so kann der Aufwandsanteil für das Aufspalten von Blattknoten mit *Gleichung 5.17* geschätzt werden.

$$A_{BSplit} = \frac{I_{UNQ}}{P_{ANZT}} \cdot \frac{LE}{\underbrace{DB_{PS} - DB_{VIP} - \underbrace{2 \cdot DB_{LBA}}_{Verkettung}}_{Bereich\ für\ Einträge} + \underbrace{\left(LE \cdot \frac{I_{UNQ}}{P_{ANZT}} \right)}_{Knotenüberlauf}} \qquad \text{Gleichung 5.17}$$

Das Aufspalten von Blattknoten kann auch Splitting-Operationen in den inneren Ebenen des Index nach sich ziehen. Da sich der Aufbau der Blattknoten bei B*-Bäumen vom Aufbau der Knoten der inneren Ebenen unterscheidet, muss der durch das Aufspalten von Knoten der inneren Ebenen einzurechnende Aufwandsanteil anders geschätzt werden (*Gleichung 5.18*). Dazu wird die Zahl der Einträge benötigt, die in einem Knoten der inneren Ebenen untergebracht werden können. Hierzu muss die Schlüssellänge (*LK* -Gleichung B.8), die Länge der für einen Indexeintrag evtl. benötigten Verwaltungsinformationen (DB_{LVK}) und die Länge des Verweises auf den jeweiligen Sohnknoten (DB_{LBA}) berücksichtigt werden. Die Zahl der Söhne eines Knotens der inneren Ebenen ist bei B-Bäumen üblicherweise um eins höher als die Zahl der enthaltenen Einträge. Der Platz für den zusätzlichen Verweis wird im Nenner bei der Berechnung des für Einträge verfügbaren Speichers berücksichtigt.

$$A_{ISplit} = \frac{\overbrace{LK + DB_{LVK} + DB_{LBA}}^{Länge\,eines\,Eintrags}}{\underbrace{DB_{PS} - DB_{VIP} - DB_{LBA}}_{Bereich\ für\ Einträge} + \underbrace{LK + DB_{LVK} + DB_{LBA}}_{Eintrag\ für\ Knotenüberlauf}}$$

Gleichung 5.18

Muss ein Knoten geteilt werden, so muss ein neuer Block reserviert werden. Eine Ausnahme bildet hier das Aufspalten des Wurzelknotens. In diesem Fall müssen zwei neue Blöcke reserviert werden. Nachdem die entsprechenden Einträge umverteilt wurden, müssen der ursprüngliche Knoten, der neue Knoten und deren Vaterknoten (von dem angenommen wird, dass er sich noch im Puffer befindet) geschrieben werden. Somit fallen drei (beim Aufspalten der Wurzel vier) zusätzliche Blockzugriffe an. Der Aufwand für das Einfügen von Indexeinträgen kann mit *Gleichung 5.19* abgeschätzt werden.

$$C_{InsertIndex} = \underbrace{\sum_{i=1}^{I_{LEV}} \left(1 - I_{BR_i}\right)}_{Durchsuchen} + \underbrace{1}_{Schreiben\,Blatt} + \underbrace{3 \cdot \left(A_{BSplit} \cdot \sum_{i=0}^{I_{LEV}-1} A_{ISplit}^{\,i} \right)}_{Aufwand\ durch\ Blocksplitting} + \underbrace{A_{BSplit} \cdot A_{ISplit}^{\,I_{LEV}-1}}_{Aufwand\ neue\ Wurzel}$$

Gleichung 5.19

Verfügbare DBMS-Produkte verwenden oftmals Implementierungen von Löschoperationen für B*-Bäume, bei denen gelöschte Einträge lediglich als ungültig gekennzeichnet werden. Der vorher belegte Speicherplatz kann dann für andere oder neue Einträge des Index genutzt werden. Verschmelzungsoperationen von Indexknoten finden meist nicht statt, da i.d.R. davon ausgegangen wird, dass die Zahl der Einfügeoperationen die Zahl der Löschoperationen übersteigt und somit die durch Löschoperationen entstehenden Freispeicherlücken wieder gefüllt werden. Diese Annahme fließt auch in *Gleichung 5.20* ein. Durch das zunächst stattfindende Auffüllen der entstandenen Freispeicherlücken müssen seltener Knoten geteilt werden. Um dies zu berücksichtigen, müssten die Anzahl Einfügeoperationen (E) und die Anzahl Löschoperationen (L) in die Aufwandsschätzung in Gleichung 5.20 einbezogen werden.

$$C_{InsertIndex} = \underbrace{\sum_{i=1}^{I_{LEV}} \left(1 - I_{BR_i}\right)}_{Durchsuchen} + \underbrace{1}_{Schreiben\,Blatt} + \left(\underbrace{4 \cdot \left(A_{BSplit} \cdot \sum_{i=0}^{I_{LEV}-1} A_{ISplit}^{\,i} \right)}_{Aufwand\ durch\ Blocksplitting} + \underbrace{A_{BSplit} \cdot A_{ISplit}^{\,I_{LEV}-1}}_{Aufwand\ neue\ Wurzel} \right) \cdot \left(\frac{E - L}{E + L} \right)$$

Gleichung 5.20

Zur Berechnung des gesamten bei einer Einfügeoperation anfallenden I/O-Aufwands müssen die Kosten zum Einfügen des Datensatzes und zum Aktualisieren aller Indexe summiert werden (*Gleichung 5.21*).

$$C_{Insert} = C_{InsertDaten} + LOG_{InsertDaten} + \sum_{\forall Indexe} \left(C_{InsertIndex} + C_{Log_{InsertIndex}} \right)$$

Gleichung 5.21

Der für das Transaktions-Logging anfallende Aufwand wird mittels der Größen $LOG_{InsertDaten}$ und $C_{LogInsertIndex}$ (siehe *Abschnitt 5.4.5 - Tabelle 5.2 und Gleichung 5.34*) berücksichtigt.

5.4.2.2 Löschen

Die hier angestellten Betrachtungen zur Abschätzung des bei Löschoperationen anfallenden Aufwands berücksichtigen das Löschen von Sätzen aus jeweils einer Tabelle. Der Fall von sich durch das Vorhandensein von Fremdschlüssel-Constraints eventuell intern ergebenden weiteren Löschoperationen (z.B. im Zusammenhang mit Klauseln wie `ON DELETE CASCADE`) wird in den Kostenfunktionen nicht berücksichtigt.

Beim Löschen von Daten werden die entsprechenden Datensätze als gelöscht gekennzeichnet (z.B. werden die zugehörigen Slot-Einträge als ungültig gekennzeichnet). Bevor dies jedoch geschehen kann, muss nach den zu löschenden Daten gesucht werden. Dazu werden die normalen Zugriffsoperationen genutzt. Die Auswahl einer bestimmten Zugriffsoperation erfolgt durch den Anfrageoptimierer. Eine Freigabe von durch „normale“ Löschoperationen vollständig geleerten Blöcken erfolgt (analog zu gängigen Implementierungen von DBMS) nicht.

Beim Löschen des eigentlichen Datensatzes muss auf den Block zugegriffen werden, in dem sich der Satz befindet. Der Aufwand ist vom Pufferungsgrad der Datenblöcke (P_{BR}) abhängig. Nachdem der Satz als gelöscht gekennzeichnet wurde, muss der geänderte Block geschrieben werden. Dabei wird hier wieder der Fall vernachlässigt, dass (bspw. bei Massenlöschungen) mehrere Löschoperationen erfolgen, bevor der Block (asynchron) gespeichert wird. Beim Löschen ausgelagerter Sätze muss neben dem eigentlichen Datensatz noch der Auslagerungszeiger gelöscht werden. Dieser Aufwand wird mit dem Anteil ausgelagerter Sätze an der Gesamtzahl der Sätze gewichtet. Mit *Gleichung 5.22* kann der Aufwand zum Löschen eines Datensatzes abgeschätzt werden.

$$C_{DeleteDaten} = \underbrace{(1 - P_{BR}) \cdot \left(1 + \frac{P_{OVFL}}{P_{ANZT}}\right)}_{Leseaufwand} + \underbrace{\left(1 + \frac{P_{OVFL}}{P_{ANZT}}\right)}_{Schreibaufwand} \qquad \text{Gleichung 5.22}$$

Beim Löschen von Datensätzen müssen auch evtl. vorhandene Indexe gepflegt werden. Dabei fällt zunächst der Aufwand für den Zugriff auf die jeweiligen Blattknoten an. Bei eindeutigen Indexen wird beim Löschen eines Satzes jeweils ein kompletter Indexeintrag gelöscht. Bei nicht eindeutigen Indexen kann auch nur das Entfernen des entsprechenden Verweises auf den zu löschenden Datensatz aus der Verweisliste des betroffenen Indexeintrags nötig sein. Die Tatsache, dass sich die zu einem Schlüsselwert gehörende Liste mit Verweisen auf Datensätze über die Grenzen von Blattknoten hinweg erstrecken kann, wird in *Gleichung 5.23* beim Suchaufwand in der Blattebene berücksichtigt. Nach der Aktualisierung muss der geänderte Block gespeichert werden.

$$C_{DeleteIndex} = \underbrace{\underbrace{\sum_{i=1}^{I_{LEV}-1}\left(1-I_{BR_i}\right)}_{innere\ Ebenen} + \underbrace{\max\left(\frac{I_{LEAF}}{I_{UNQ}},1\right)\cdot I_{BR_H}}_{Blattebene}}_{Suchaufwand} + \underbrace{1}_{Schreibaufwand}$$

Gleichung 5.23

Der I/O-Aufwand für das Löschen von Datensätzen (*Gleichung 5.24*) setzt sich aus dem eigentlichen Löschaufwand, dem Aufwand zum Schreiben der Log-Einträge und dem Suchaufwand zum Auffinden der zu löschenden Sätze zusammen. *S* steht dabei für die Selektivität der Suchoperation (vgl. auch *Abschnitt 5.4.3*), über die die Anzahl der zu löschenden Sätze geschätzt wird. Die Ermittlung der anfallenden Suchkosten kann mit den in *Abschnitt 5.4.1* und *Anhang C* angegebenen Kostenfunktionen erfolgen.

Eine Aufwandsreduzierung, die sich ergeben kann, wenn die anfängliche Suche der zu löschenden Sätze über einen zu aktualisierenden Index erfolgt, wird dabei vernachlässigt. Davon ausgehend, dass das Durchsuchen von Indexen größtenteils im Puffer erfolgt, ist die durch die Vereinfachung verursachte Abweichung als gering anzusehen.

$$C_{Delete} = C_{Suche} + \underbrace{S\cdot P_{ANZT}}_{Anz.\ Löschungen}\cdot\left(C_{DeleteDaten} + LOG_{DeleteDaten} + \sum_{\forall Indexe}\left(C_{DeleteIndex} + C_{LogDeleteIndex}\right)\right)$$

Gleichung 5.24

5.4.2.3 Ändern

Auch beim Ändern von Datensätzen müssen diese zunächst gesucht werden. Dazu werden ebenfalls die normalen Zugriffsoperationen genutzt, die jeweils durch den Anfrageoptimierer ausgewählt werden.

Der Block, der den zu ändernden Datensatz enthält muss gelesen und nach der Änderung geschrieben werden. Bei Massenänderungen eventuell mögliche Verringerungen des tatsächlichen Schreibaufwands werden auch hier nicht berücksichtigt. Wird ein Datensatz durch die Änderung verlängert, so kann der Fall eintreten, dass er ausgelagert werden muss. Dann muss ein zusätzlicher Block reserviert und geschrieben werden. Die Wahrscheinlichkeit dafür wird über das Verhältnis von vorhandenen migrierten Tupeln zur Gesamtzahl der Tupel geschätzt. Mit *Gleichung 5.25* kann der Aufwand zum Ändern eines Datensatzes bestimmt werden.

$$C_{UpadteDaten} = \underbrace{\left(1-P_{BR}\right)\cdot\left(1+\frac{P_{OVFL}}{P_{ANZT}}\right)}_{Leseaufwand} + \underbrace{1}_{Schreibaufwand} + \underbrace{2\cdot\frac{P_{OVFL}}{P_{ANZT}}}_{Aufwand\ f.\ Auslagerung}$$

Gleichung 5.25

Wird der Wert eines indexierten Attributs geändert, so müssen die das entsprechende Attribut enthaltenden Indexe gepflegt werden. Dies wird durch das Löschen und Einfügen von Indexeinträgen bzw. das Aktualisieren von Verweislisten realisiert. In

Gleichung 5.26 wird zur Ermittlung der Kosten für die Pflege eines Index auf die mit Gleichung 5.19 und Gleichung 5.23 möglichen Kostenschätzungen zurückgegriffen.

$$C_{Update_{Index}} = C_{Delete_{Index}} + C_{Insert_{Index}}$$

Gleichung 5.26

Auch bei Änderungsoperationen (*Gleichung 5.27*) wird, neben dem Aufwand zum Ändern der Datensätze und zum Aktualisieren der Indexe, der Aufwand zum Schreiben von Log-Einträgen und der Suchaufwand zum Auffinden der zu ändernden Sätze berücksichtigt. Eine Indexaktualisierung muss im Unterschied zu Einfüge- und Löschoperationen hier allerdings nicht für alle Indexe erfolgen, sondern nur für die Indexe, die Attribute enthalten, deren Werte geändert wurden. Muss ein Datensatz nach der Änderung in einen neuen Datenblock ausgelagert werden, so fällt auch ein erhöhter Log-Aufwand an, da im alten Datenblock der bisherige Satz in einen Auslagerungszeiger umgewandelt wird und der geänderte Satz in den neuen Block eingefügt wird. Für den damit verbundenen Mehraufwand steht die Größe LOG_{OVFL} (vgl. auch *Abschnitt 5.4.5*).

$$C_{Update} = C_{Suche} + \underbrace{S \cdot P_{ANZT}}_{Anz.\,Änderungen} \cdot$$

$$\cdot \left(C_{Update_{Daten}} + LOG_{Update\,Daten} + \underbrace{\frac{P_{OVFL}}{P_{ANZT}} \cdot LOG_{OVFL}}_{Log-Aufwand\ f.\ Auslagerung} + \sum_{\forall\, betroffenen\, Indexe} \left(C_{Update_{Index}} + C_{Log_{UpdateIndex}} \right) \right)$$

Gleichung 5.27

5.4.3 Join-Operationen

In den Beispiel-Workloads in *Abschnitt 6.3.2.6* werden auch Join-Operationen ausgeführt. Deshalb werden hier Kostenabschätzungen für Nested-Loop-Joins und Hash-Joins kurz betrachtet. In [Gan05] wurde das Thema sowie das Thema der Bestimmung der Selektivität von Anfragen eingehender behandelt. Join-Operationen selbst führen keine Datenzugriffe durch. Die Ausführung einer Join-Operation bewirkt implizit die Ausführung von Zugriffsoperationen. Das heißt, dass in Speicherungsstrukturen vorhandene Degenerierungen nicht bei der Ermittlung der Kosten für Join-Operationen berücksichtigt werden müssen. Ihre Berücksichtigung erfolgt bei der Schätzung der Kosten für die Zugriffsoperationen. Die vom Optimierer jeweils zur Ausführung einer Join-Operation ausgewählte Implementierungsvariante hat lediglich Einfluss darauf, wie oft die jeweiligen Zugriffsoperationen, die i.d.R. auch in den Ausführungsplänen ausgewiesen sind (*Abbildung 5.7*), ausgeführt werden. Deshalb kann hier auch auf die bekannten Verfahren zur näherungsweisen Berechnung der bei Join-Operationen anfallenden Kosten zurückgegriffen werden.

```
SQLWKS> explain plan for
     2> SELECT d.name,d.ort,v.ziel_rufnr,v.zeit,v.verb_dauer FROM dienstkunden d,
     3> verbindungen v WHERE d.dk_nr=100100 AND d.dk_nr=v.gew_nr;
Anweisung verarbeitet
SQLWKS> SELECT id, operation, options, object_name, cost FROM plan_table;
ID         OPERATION                 OPTIONS           OBJECT_NAME             COST
---------- ------------------------- ----------------- -------------------- ----------
         0 SELECT STATEMENT                                                      3771
         1 NESTED LOOPS                                                          3771
         2 TABLE ACCESS              FULL              DIENSTKUNDEN            2882
         3 TABLE ACCESS              FULL              VERBINDUNGEN             889
```

Abbildung 5.7: Auszug aus einem Ausführungsplan (Oracle)

Eine einfache Implementierung stellt der Nested-Loop-Join dar, der durch zwei verschachtelte Schleifen realisiert wird (*Abbildung 5.8*). In der äußeren Schleife werden alle Tupel der linken Tabelle (L) und in der inneren Schleife jeweils alle Tupel der rechten Tabelle (R) durchlaufen. Damit wird jedes Tupel der linken Tabelle mit jedem Tupel der rechten Tabelle gemäß der Join-Bedingung verglichen. Handelt es sich um einen Equi-Join (wie in Abbildung 5.8), so wird auf Gleichheit der Werte der Join-Attribute geprüft.

```
resulttable := {};

for each tuple r ∈ L do                    -- äußere Schleife
     for each tuple s ∈ R do               -- innere Schleife
          if l.A = r.A then                -- Gleichheit wegen Equi-Join
               resulttable := resulttable ∪ {(l,r)};
     end; (for)
end; (for)
```

Abbildung 5.8: Pseudocode-Darstellung zur Implementierung eines Nested-Loop-Join [Gan05]

Bei der Ausführung eines Nested-Loop-Join fallen Kosten für eine Zugriffsoperation auf die linke Tabelle an. Auf die rechte Tabelle muss mehrfach zugegriffen werden. Die Zahl der Wiederholungen entspricht der Anzahl der Treffer, die der Zugriff auf die linke Tabelle liefert. Um diesen Wert abschätzen zu können, muss die Selektivität[10] (S) des Zugriffs auf die linke Tabelle bekannt sein.

Zur Ermittlung der Selektivität müssen statistische Daten über die in der Zugriffsoperation verwendeten Suchschlüssel vorhanden sein. Erfolgt der Zugriff über einen Index, so kann die Selektivität eines einfachen Index Lookup aus der Anzahl eindeutiger Schlüsselwerte des Index (I_{UNQ}) und der Anzahl der in der indexierten Tabelle enthaltenen Tupel (P_{ANZT}) ermittelt werden (*Gleichung 5.28*).

$$S = \frac{I_{UNQ}}{P_{ANZT}}$$ Gleichung 5.28

[10] Als Selektivität wird hier das Verhältnis der Anzahl Treffer, die in einer Tabelle gefunden werden, zur Gesamtzahl der Tupel der Tabelle angesehen. Das heißt, je größer der Zahlenwert für die Selektivität, desto geringer die Auswahlschärfe der Selektionsbedingung.

Für eine Bereichsanfrage kann die Selektivität im einfachsten Fall mit *Gleichung 5.29* näherungsweise geschätzt werden.

$$S = \frac{1 + SK_{MAX} - SK_{MIN}}{1 + I_{MAX} - I_{MIN}}$$ Gleichung 5.29

Dazu müssen die folgenden Größen bekannt sein:

- SK_{MAX} entspricht der oberen Intervallgrenze in der Selektionsbedingung.
- SK_{MIN} entspricht der unteren Intervallgrenze in der Selektionsbedingung.
- I_{MAX} entspricht dem größten Schlüsselwert des Index.
- I_{MIN} entspricht dem kleinsten Schlüsselwert des Index.

Voraussetzung für diese beiden sehr einfachen Methoden ist in etwa eine Gleichverteilung und die Unabhängigkeit der Schlüsselwerte, die oftmals allerdings nicht vorausgesetzt werden kann. DBMS-Produkte benutzen ansonsten intern oft Histogramme, um genauere Informationen über Werteverteilungen festzuhalten. Dazu wird der Wertebereich eines Schlüssels in Intervalle aufgeteilt, für die die Häufigkeit des Auftretens der Attributwerte festgehalten wird. Innerhalb der Intervalle wird von einer Gleichverteilung der Schlüsselwerte ausgegangen. *Abbildung 5.9* zeigt ein Beispiel für ein Histogramm, aus dem Informationen über die Altersstruktur der Kunden eines Versandhandelsunternehmens abgelesen werden können.

Sollen bei einer Anfrage nur die Daten der Kunden berücksichtigt werden, die 50 Jahre und älter sind, so müssen die Zahlen der drei rechten Säulen summiert werden. Das mittlere Intervall muss aufgeteilt werden. Insgesamt erstreckt sich dass Intervall über zehn Schlüsselwerte, auf die jeweils 780 Kunden entfallen. In die Zielgruppe fallen fünf Schlüsselwerte. Damit kann die Zahl der Treffer der Anfrage auf insgesamt 22200 Treffer geschätzt werden.

Die Selektivität ergibt sich wieder aus dem Verhältnis der Anzahl der Treffer zur Gesamtzahl der Tupel der Tabelle (*Gleichung 5.30*).

$$S = \frac{Anzahl\ Treffer}{P_{ANZT}}$$ Gleichung 5.30

Sind keine Informationen über die Anzahl Schlüsselwerte oder deren Verteilung bekannt, so wird mit festen Annahmen gearbeitet. Dann wird z.B. angenommen, dass ein bestimmter Suchschlüsselwert bei 10% aller Tupel einer Tabelle vorkommt [HR01].

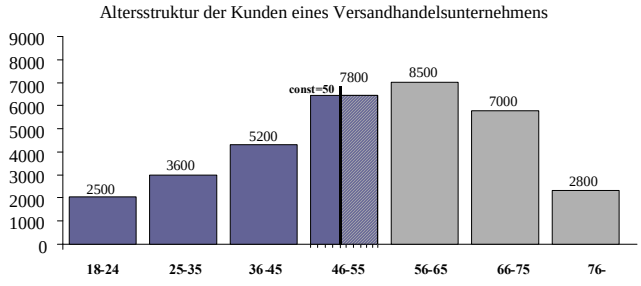


Abbildung 5.9: Beispiel für ein Histogramm

Das Thema der Bestimmung der Selektivität von Anfragen sollte hier nur kurz angerissen werden. Da die Selektivität von Anfragen unabhängig von in physischen Speicherungsstrukturen auftretenden Degenerierungen ist, kann auch hier auf die bekannten Vorgehensweisen zurückgegriffen werden.

Ist die Selektivität der Anfrage gegen die linke Tabelle bekannt, so können die für die Ausführung eines Nested-Loop-Join anfallenden I/O-Kosten mit *Gleichung 5.31* geschätzt werden.

$$C_{NestedLoop} = C_{Suche_{links}} + S \cdot P_{ANZT_{links}} \cdot C_{Suche_{rechts}} \qquad \text{Gleichung 5.31}$$

Die Kosten für das Durchsuchen der linken bzw. rechten Tabelle können analog zu *Abschnitt 5.4.1* geschätzt werden. Da für die Anzahl der gegen die rechte Tabelle gerichteten Anfragen die Anzahl Treffer benötigt wird, die die gegen die linke Tabelle gerichtete Anfrage liefert, wird die Selektivität mit der Zahl der Tupel der linken Tabelle multipliziert.

Die zweite Implementierungsvariante, die hier kurz betrachtet werden soll, ist der Hash-Join. Die von Oracle [Ora03c] verwendete Implementierungsvariante läuft in zwei Phasen ab. Dabei wird in eine sog. innere Tabelle und eine äußere Tabelle unterschieden. Die kleinere Tabelle wird (unter Berücksichtigung evtl. vorher auf die Tabellen angewendeter Selektionsoperationen) als innere Tabelle verwendet. Zunächst wird auf die innere Tabelle eine Suchoperation angewendet, bei der evtl. vorhandene Selektionsbedingungen berücksichtigt werden. Auf die Werte der Join-Attribute der von der Suchoperation gelieferten Treffer wird eine Hash-Funktion angewendet und die Daten in Buckets im Arbeitsspeicher untergebracht. In der sich an diese Partitionierungsphase [EN02] anschließenden Probing-Phase wird auf die äußere Tabelle zugegriffen. Dabei wird die Hash-Funktion ebenfalls für jeden Treffer auf die Werte der Join-Attribute angewendet. Damit ist bekannt, in welchem Bucket eventuell Tupel enthalten sein können, für die eine Verknüpfung durchzuführen ist. Unter der Voraussetzung, dass alle Buckets im Arbeitsspeicher gehalten werden können, ist diese Implementierung sehr effizient. Auf beide Tabellen muss jeweils nur eine Zugriffsoperation angewendet werden. Die Kosten können mit *Gleichung 5.32* geschätzt werden.

$$C_{HashJoin} = C_{Suche_{innen}} + C_{Suche_{außen}} \qquad \text{Gleichung 5.32}$$

Werden mehr als zwei Tabellen miteinander verknüpft, so werden mehrere Join-Operationen nacheinander ausgeführt. Die Ergebnistabelle der ersten Join-Operation ist Ausgangspunkt für die zweite Verknüpfung usw. Für Kostenermittlungen ist es dann auch nötig, die Größe der Zwischenergebnisse zu schätzen sowie eventuelle Schreib- und Lesekosten (auf temporäre Bereiche) für deren Erzeugung und Weiterverarbeitung zu berücksichtigen. In [Gan05] werden auch dafür Ansätze aufgezeigt, die hier aber nicht näher betrachtet werden sollen.

5.4.4 Ermittlung datenbankobjektspezifischer Pufferungsgrade

Viele der üblichen Kostenmodelle berücksichtigen Pufferungsgrade (Buffer Hit Ratio) von Daten- und Indexblöcken entweder gar nicht oder mittels grober

Annahmen. Dies ist oft darauf zurückzuführen, dass insbesondere detaillierte datenbank*objekt*- und *operations*bezogene Statistiken (etwa tabellen- oder indexbezogen) über Pufferungsgrade von DBMS-Produkten derzeit i.d.R. nicht direkt bzw. nicht nach „außen sichtbar“ angeboten werden, weil sie sehr feingranular sind und übergreifend über mehrere Schichten der DBMS-Architektur gesammelt werden müssten. Häufig bieten DBMS-Produkte nur Statistiken über alle systemweit angefallenen physischen und logischen Blockzugriffe an. Die in der vorliegenden Arbeit vorgestellten Kostenfunktionen berücksichtigen aber teilweise detaillierte Pufferungsgrade. Stellt das jeweilige Basissystem keine derartigen Informationen bereit, so müssen angenommene Werte eingesetzt werden. Wird für den Pufferungsgrad jeweils der Wert 0 angenommen, so liefen die Kostenfunktionen die Werte für den anfallenden logischen I/O-Aufwand.

Zur Überprüfung der Kostenfunktionen wurde im Rahmen der in [Kli05] angestellten Untersuchungen auch der Prototyp einer Komponente zur Ermittlung datenbankobjektspezifischer Pufferungsgrade implementiert. Als Basis für Implementierung und Test wurde das DBMS-Produkt Oracle in den Versionen 8i bis 10g verwendet. Die Komponente ermittelt den Pufferungsgrad (z.B. P_{BR} für eine Daten enthaltende Partition - *Gleichung 5.33*) in Form des Verhältnisses aus der Anzahl der von einem Datenbankobjekt belegten Blöcke (P_{USED}) und der Anzahl Blöcke des Datenbankobjekts, die im Puffer gefunden werden (NP_B). Eine Gleichverteilung anfallender Zugriffe angenommen, stellt der Pufferungsgrad hier die Wahrscheinlichkeit dar, mit der sich ein benötigter Block im Puffer befindet.

$$P_{BR} = \frac{NP_B}{P_{USED}} \qquad \text{Gleichung 5.33}$$

Neben den statistischen Daten aus dem eInformationsschema, wird die von Oracle in relationaler Form zur Verfügung gestellte Schnittstelle zum Datenbankpuffer genutzt. Aus der Systemtabelle `x$bh` bzw. der Sicht `v$bh` kann für alle im Puffer befindlichen Daten- bzw. Indexblöcke neben der Blocknummer und Statusinformationen auch der Identifikator des Datenbankobjekts abgelesen werden, zu dem der Block gehört. Damit kann relativ einfach die Zahl der im Puffer befindlichen Blöcke eines Datenbankobjekts bestimmt werden. Für Datenbereiche ist diese Information ausreichend. Aufwendiger ist die Ermittlung detaillierter Pufferungsgrade für Indexbäume. Für eine möglichst genaue Bestimmung der beim Durchmustern eines Indexbaums zu erwartenden Anzahl physischer Blockzugriffe, muss der Pufferungsgrad der einzelnen Indexebenen bekannt sein. Aus den statistischen Daten, die zu Indexbäumen verfügbar sind (Anzahl Blätter und Anzahl Ebenen) lässt sich die Anzahl der auf den einzelnen Indexebenen befindlichen Blöcke (zumindest wenn der Index über mehr als drei Ebenen verfügt) nur ungenau berechnen [Ham83, Küs83].

Daher werden zunächst Informationen über den Indexbaum gesammelt, indem alle belegten Blöcke des Indexsegments gelesen werden. Aus den Verwaltungsinformationen der Blöcke kann auch die Ebene abgelesen werden, auf der sich der Block im Indexbaum befindet.

Diese Vorgehensweise ist aufwendig. Die Komponente zur Ermittlung detaillierter Pufferungsgrade wurde von uns allerdings auch vorrangig mit dem Ziel der Überprüfung der Kostenfunktionen implementiert. Weiterhin dient das Scannen der Indexblöcke lediglich dazu, notwendige statistischen Basisinformationen zur Verfügung zu stellen, welche zur Ermittlung aktueller Pufferungsgrade verwendet werden können, bis sich die konkrete Struktur eines Indexbaums (z.B. durch das Einfügen größerer Datenmengen) signifikant geändert hat. Die Berechnung des aktuellen Pufferungsgrads einer Indexebene erfolgt (analog zu Gleichung 5.33) aus dem Verhältnis der Anzahl der im Puffer befindlichen Blöcke einer Indexebene zur Gesamtzahl der Blöcke auf der entsprechenden Ebene.

Der mit der Vorgehensweise verbundene relativ hohe Aufwand zur Bestimmung detaillierter Pufferungsgrade, lässt die konkrete Implementierung nicht für den permanenten Einsatz im laufenden Datenbankbetrieb geeignet erscheinen. Allerdings kann sie beispielsweise die Ermittlung von Werten unterstützen, die als Annahmen in die Berechnungen einfließen.

Abhängig von den konkreten Implementierungen bei Datenbank-Management-Systemen ist es möglich, dass bei der Ausführung sequenzieller Suchen andere Seitenersetzungsstrategien angewendet werden als bei Zugriffen über Indexe. Damit soll verhindert werden, dass häufig benötigte Blöcke durch (evtl. seltenes) sequenzielles Durchsuchen großer Tabellen aus dem Puffer verdrängt und für nachfolgende Operationen erst wieder mit hohem Aufwand in den Puffer gebracht werden müssen. In solchen Fällen müssten verschiedene Pufferungsgrade für Datenblöcke festgehalten werden, was im eInformationsschema auch so möglich ist.

5.4.5 Schätzung des Aufwands für Transaktionsprotokollierung

Um die von Transaktionen geforderte ACID-Eigenschaft sicherzustellen werden u.a. Änderungen an Datenbankinhalten im Transaktionsprotokoll (Log) vermerkt. Im Rahmen der in *Abschnitt 5.4.2* beschriebenen Änderungsoperationen und besonders auch im Rahmen von In-Place-Reorganisationen fallen I/O-Kosten für das Schreiben von Log-Einträgen an. In der vorliegenden Arbeit werden allerdings nur Log-Einträge berücksichtigt, die Änderungen an Daten- und Indexbereichen widerspiegeln. Einträge, die bspw. im Rahmen von Checkpoints erzeugt werden, bleiben unberücksichtigt. Der konkrete Aufbau der Log-Einträge unterscheidet sich bei einzelnen Datenbank-Management-Systemen. Unter Berücksichtigung verschiedener Quellen [Gol98, EN02, HR01, SHS05] bestehen Log-Einträge meist aus einem Teil mit fester Länge und einem Teil mit variabler Länge. Der feste Teil enthält Informationen wie bspw. die Nummer des Log-Eintrags (LSN bzw. LRSN), die Nummer der zugehörigen Transaktion, den Typ des Log-Eintrags, die Nummer des Blocks, an dem die Änderung vorgenommen wurde, die Identifikation des betroffenen Datensatzes (bspw. dessen RID) oder Indexeintrags und evtl. die Nummer des vorherigen Log-Eintrags der selben Transaktion. Für die jeweilige Länge dieses festen Teils stehen im eInformationsschema die Größen DB_{LLOGD} für Einträge, die Veränderungen an Datensätzen repräsentieren und DB_{LLOGI} für Einträge, die

Veränderungen an Indexeinträgen widerspiegeln. Für Indexe wird angenommen, dass nur die Veränderungen an den Einträgen in der Blattebene protokolliert (geloggt) werden. Veränderungen an Blöcken der inneren Ebenen, die bspw. durch Splitting-Operationen nach dem Einfügen oder Verlängern von Indexeinträgen auf der Blattebene auftreten können, werden nicht im Log aufgezeichnet. Der variabel lange Teil kann, abhängig vom Typ des Log-Eintrags, eine Redo-Information, die im Falle eines Recovery benötigt wird und eine Undo-Information zum Zurücksetzen der von einer Transaktion vorgenommenen Änderungen im Falle des Transaktionsabbruchs enthalten. Im Rahmen der vorliegenden Arbeit wird angenommen, dass die Undo- und Redo-Information jeweils den gesamten geänderten Datensatz bzw. Indexeintrag enthält. Daher richtet sich die Länge dieses Teils auch nach der jeweiligen Länge von Datensätzen (LE) bzw. Indexeinträgen (LT). Wenn beim Ändern eines Datensatzes ein Überlaufsatz entsteht, so müssen zwei Log-Sätze geschrieben werden. Der Log-Satz für den Block, in dem der Satz bisher gespeichert war enthält als Undo-Image den Datensatz im alten Zustand und den Auslagerungszeiger als Redo-Image. Der zweite Logsatz steht für die Einfügeoperation im neuen Block. Zusätzlich zum Schreibaufwand für das Logging einer „normalen“ Änderung muss der feste Teil des neuen Log-Eintrags und das Image des Auslagerungszeigers geschrieben werden. Für diesen Zusatzaufwand steht die Größe LOG_{OVFL}.

Die Längen der Log-Einträge werden in Byte angegeben. Das Schreiben der Log-Einträge erfolgt gepuffert. Das heißt, dass die zu schreibenden Log-Einträge zunächst in einem Puffer von der Größe eines Blocks zwischengespeichert werden. Erst wenn der Block gefüllt ist, wird dieser auf ein entsprechendes Sekundärspeichermedium geschrieben. Hiermit ist zwar ein gewisses Risiko verbunden, das aber in Kauf genommen wird. Da in der vorliegenden Arbeit als Einheit für Aufwandsschätzungen Blockzugriffe verwendet werden, wird die Größe der Log-Einträge jeweils durch die Blockgröße geteilt.

Die Länge von Log-Einträgen, die Änderungen an Daten widerspiegeln, kann i.d.R. direkt angegeben werden (vgl. auch Tabelle 5.2).

Formelzeichen	Verwendung	Undo-Image	Redo-Image	Länge
$LOG_{InsertDaten}$	Einfügen eines Datensatzes		×	$(DB_{LLOGD} + LT)/DB_{PS}$
$LOG_{UpdateDaten}$	Ändern eines Datensatzes	×	×	$(DB_{LLOGD} + 2 \cdot LT)/DB_{PS}$
$LOG_{DeleteDaten}$	Löschen eines Datensatzes	×		$(DB_{LLOGD} + LT)/DB_{PS}$
LOG_{OVFL}	Mehraufwand neuer Überlaufsatz		×	$(DB_{LLOGD} + LP)/DB_{PS}$
$LOG_{InsertIndex}$	Einfügen eines Indexeintrags		×	$(DB_{LLOGI} + LE)/DB_{PS}$
$LOG_{UpdateIndex}$	Ändern eines Indexeintrags	×	×	$(DB_{LLOGI} + 2 \cdot LE)/DB_{PS}$
$LOG_{DeleteIndex}$	Löschen eines Indexeintrags	×		$(DB_{LLOGI} + LE)/DB_{PS}$

Tabelle 5.2: Aufbau von Log-Einträgen

Bei im Rahmen von Änderungsoperationen vorzunehmenden Aktualisierungen von Indexen müssen je Index u.U. mehrere Log-Einträge erzeugt werden. Mit den

folgenden Gleichungen kann die Summe der Längen der dabei zu erzeugenden Log-Einträge näherungsweise ermittelt werden.

Bei eindeutigen Indexen ist das Einfügen oder Löschen eines Datensatzes auch immer mit dem Einfügen oder Löschen eines Indexeintrags verbunden. Bei nicht eindeutigen Indexen kann das Einfügen oder Löschen eines Datensatzes auch nur Änderungen an den Verweislisten von vorhandenen Indexeinträgen bewirken. Dies muss bei der näherungsweisen Bestimmung der Kosten für das Schreiben von Log-Einträgen in *Gleichung 5.34* und *Gleichung 5.35* berücksichtigt werden. Die Wahrscheinlichkeit, dass ein neuer Indexeintrag eingefügt werden muss, wird über das Verhältnis eindeutiger Schlüsselwerte zur Gesamtzahl Tupel ermittelt. Mit Gleichung 5.34 kann der Aufwand zum Schreiben der im Rahmen der Pflege eines Index während einer Einfügeoperation anfallenden Log-Einträge geschätzt werden.

$$C_{Log_{InsertIndex}} = \underbrace{\frac{I_{UNQ}}{P_{ANZT}} \cdot LOG_{Insert_{Index}}}_{Einfügen\ Eintrag} + \underbrace{\left(1 - \frac{I_{UNQ}}{P_{ANZT}}\right) \cdot LOG_{Update_{Index}}}_{Einfügen\ Verweis} \qquad \text{Gleichung 5.34}$$

Zur Schätzung des Aufwands zum Schreiben von Log-Einträgen im Rahmen der mit einer Löschoperation verbundenen Pflege von Indexen kann Gleichung 5.35 verwendet werden.

$$C_{Log_{DeleteIndex}} = \underbrace{\frac{I_{UNQ}}{P_{ANZT}} \cdot LOG_{Delete_{Index}}}_{Löschen\ Eintrag} + \underbrace{\left(1 - \frac{I_{UNQ}}{P_{ANZT}}\right) \cdot LOG_{Update_{Index}}}_{Löschen\ Verweis} \qquad \text{Gleichung 5.35}$$

Müssen bei einer Änderung von Attributwerten Indexeinträge gepflegt werden, so sind folgende Fälle möglich:

- Ein Indexeintrag muss gelöscht und ein Indexeintrag eingefügt werden.
- Ein Indexeintrag muss gelöscht und ein Indexeintrag geändert werden.
- Ein Indexeintrag muss geändert und ein Indexeintrag eingefügt werden.
- Zwei Indexeinträge müssen geändert werden.

Die Kosten hierfür lassen sich unter Verwendung der Werte aus Gleichung 5.34 und Gleichung 5.35 ermitteln (*Gleichung 5.36*).

$$C_{LOG_{UpdateIndex}} = C_{LOG_{DeleteIndex}} + C_{LOG_{UpdateIndex}} \qquad \text{Gleichung 5.36}$$

6 Reorganisationsbedarfs- und -nutzenermittlung

In diesem Kapitel wird ein im Rahmen der vorliegenden Arbeit entwickeltes Verfahren zur Durchführung von Reorganisationsbedarfsanalysen und zur Abschätzung des Nutzens von Datenbankreorganisationen beschrieben. Das Verfahren basiert auf Informationen über die in einer konkreten Systemumgebung gegen eine Datenbank gerichtete typische Workload und statistischen Informationen über den Zustand der internen Strukturen zur Datenspeicherung [Dor05b]. Die Workload-Informationen werden durch eine Protokollierung der gegen die Datenbank gerichteten Anweisungen unter Einbeziehung des Anfrageoptimierers ermittelt.

Die Informationen über den Zustand der physischen Speicherungsstrukturen werden dem ebenfalls in dieser Arbeit entwickelten eInformationsschema entnommen. Mit Hilfe der in *Kapitel 5* und den entsprechenden Anhängen vorgestellten Kostenfunktionen werden die zur Abarbeitung der protokollierten Workload notwendigen I/O-Operationen vor und nach einer eventuellen Reorganisation abgeschätzt. Die Differenz der Kostenwerte stellt den Nutzen der Datenbankreorganisation dar.

In *Abschnitt 6.1* werden verschiedene Möglichkeiten zur Bestimmung von Reorganisationszeitpunkten betrachtet. Eine Überblicksdarstellung des Verfahrens zur Reorganisationsbedarfs- und -nutzenermittlung erfolgt in *Abschnitt 6.2*. Dabei wird u.a. auch kurz auf Analysen bezüglich der Größe Speicherplatz eingegangen. Der Schwerpunkt dieses Kapitels liegt aber auf Vorgehensweisen zur Quantifizierung des Nutzens von Datenbankreorganisationen bezüglich der Systemleistung. Die Ausführungen dazu und Erläuterungen zur prototypischen Implementierung eines Werkzeugs zur Bestimmung des Nutzens von Datenbankreorganisationen sowie die Ergebnisse durchgeführter Tests finden sich in *Abschnitt 6.3*.

6.1 Ansätze zur Bestimmung von Reorganisationszeitpunkten

Die Bestimmung der Zeitpunkte, zu denen Datenbankreorganisationen durchgeführt werden sollen, erfolgt in der Praxis auf unterschiedliche Art. Häufig spielen dabei Erfahrungswerte eine Rolle. Bei Anwendungssystemen mit relativ stabiler Workload existieren nach einer gewissen Laufzeit Erfahrungen. Diese können genutzt werden, um Reorganisationsintervalle und zu reorganisierende Datenbankbereiche zu bestimmen. So kann ein DBA, nachdem er die einem Anwendungssystem zugrunde liegende Datenbank mehrmals erfolgreich reorganisiert und damit i.d.R. eine Einsparung von Speicherplatz und eine Verbesserung der Systemleistung erreicht hat, in etwa einschätzen, in welchen Intervallen reorganisiert werden sollte und welcher Nutzen durch die Reorganisation erreicht werden kann. Ändern sich allerdings die Umgebungsbedingungen der Anwendung (Hard- und Softwarebasis bzw. insbesondere die Workload), so müssen erst wieder neue Erfahrungen gesammelt werden. Dieses Sammeln von Erfahrungen ist meist ein länger währender Prozess, der evtl. auch teilweise unnötige Reorganisationen mit einschließt. Wenn aber an ein Anwendungssystem hohe Verfügbarkeitsanforderungen gestellt werden oder wenn

sich die Auslastung des Systems nahe der Kapazitätsgrenze bewegt, sind solche Vorgehensweisen nach der *Trial And Error*-Methode meist nicht akzeptabel. Hier müssen unnötige Wartungsarbeiten vermieden werden.

Werkzeuge zur Überwachung der Systemleistung (z.B. `Statspack` bei Oracle [DW00], `onperf` bei Informix [IBM05a], der `snapshot monitor` von DB2 [IBM04] oder der `SQL Server Profiler` des Microsoft SQL Servers [Bau06]), mit deren Hilfe Hardware- und Konfigurationsprobleme erkannt werden können, sind für Reorganisationsbedarfsanalysen meist ungeeignet, da sie normalerweise keine Analyse des Zustands der physischen Speicherungsstrukturen oder eine Überwachung von Zustandsveränderungen vornehmen.

Zur Durchführung von Reorganisationsbedarfsanalysen können bspw. die vom jeweiligen DBMS geführten Statistikdaten zur Lokalisierung von Degenerierungen verwendet werden. Auch viele der verfügbaren Werkzeuge zur Unterstützung von Datenbankadministratoren greifen auf die im jeweiligen Datenbankkatalog bzw. dessen Erweiterungen hinterlegten Informationen zurück (vgl. auch *Abschnitt 4.4*). Errechnete Kennzahlen über den aktuellen physischen Zustand der Speicherungsstrukturen werden in aufbereiteter Form angezeigt. Bei Überschreitungen von Schwellwerten werden Reorganisationsempfehlungen abgegeben, weil dann mit einer gewissen Wahrscheinlichkeit negative Auswirkungen auf die Systemleistung zu erwarten sind. Eine Quantifizierung der Auswirkungen vorhandener Degenerierungen in der konkreten Systemumgebung (unter Beachtung der dort anfallenden Workload!) erfolgt nicht.

Soll die Entscheidung über die Durchführung einer Datenbankreorganisation abhängig von den konkret in der jeweiligen Anwendungsumgebung zu erwartenden (positiven) Auswirkungen auf die Systemleistung gefällt werden, ist das alleinige Lokalisieren von aktuell vorhandenen Degenerierungen nicht mehr ausreichend. Ihre Wirkungen auf die Systemleistung müssen *quantifiziert* werden. Dies ist insbesondere notwendig, wenn nicht auf Erfahrungswerte zurückgegriffen werden kann. Dazu müssen neben Art und Umfang vorhandener Degenerierungen auch Informationen über die gegen die betrachteten Datenbankobjekte gerichtete Workload bekannt sein. Wird auf eine Tabelle bspw. nur selten (und wenn überhaupt, dann unter Nutzung von Indexen) zugegriffen, so hat dort vorhandener eingestreuter Freiplatz kaum Auswirkungen auf die Systemleistung und eine Reorganisation ist höchstens zur Gewinnung von Speicherplatz sinnvoll.

Vorhandene Degenerierungen führen i.d.R. zu einer Erhöhung der Zahl der I/O-Operationen. Deshalb wird der mit einer Datenbankreorganisation erreichbare Nutzen hier als die zu erwartende prozentuale Einsparung an I/O-Kosten angegeben. Als Maß für diese Kosten wird die Anzahl anfallender Blockzugriffe (wie mit den Gleichungen aus *Kapitel 5* ermittelbar) verwendet. Diese stellen ein aus unserer Sicht geeignetes Maß für den anfallenden I/O-Aufwand dar.

Um den Einfluss vorhandener Degenerierungen auf den notwendigen I/O-Aufwand quantifizieren zu können, muss der auf der Satz- und Zugriffspfadschnittstelle durch sie entstehende Mehraufwand ermittelt werden. Der für die elementaren Zugriffsoperationen jeweils ermittelte Mehraufwand ist allerdings noch wenig

aussagekräftig. Hier wird nicht berücksichtigt, welchen Anteil die jeweilige Einzeloperation an der

- Gesamt-Workload,
- der auf das betrachtete Datenbankobjekt angewendeten Workload oder
- der betrachteten SQL-Anweisung

ausmacht. Diese Anteile müssen ermittelt und berücksichtigt/gewichtet werden. Dadurch erfolgt die Überwindung der konzeptuellen Kluft zwischen den grundlegenden Operationen auf der Satz- und Zugriffsschnittstelle einerseits und komplexen SQL-Anweisungen bzw. ganzen Anwendungen andererseits.

Ausgehend von der prognostizierten Einsparung von I/O-Aufwand muss der Datenbankadministrator letztendlich entscheiden, ob dieses Einsparungspotenzial den Aufwand einer Datenbankreorganisation rechtfertigt. Die Verwendung von Prozentwerten ermöglicht zunächst auch eine Unabhängigkeit von Einflussgrößen, wie bspw. der Leistungsfähigkeit der konkret verwendeten Hardware. In einer konkreten Anwendungsumgebung kann der Prozentwert dann, wenn noch weitere Größen (z.B. Operationslaufzeiten) bekannt sind, als Basis für entsprechende Umrechnungen (z.B. in Zeiteinheiten) dienen.

6.2 Reorganisationsbedarfs- und -nutzenermittlung

Die konkreten Abläufe bei Reorganisationsbedarfs- und -nutzenanalysen richten sich (wie auch aus *Abbildung 6.1* ersichtlich) größtenteils nach dem Ziel, das mit Datenbankreorganisation verfolgt wird.

Soll festgestellt werden, ob mit einer Reorganisation *Speicherplatz* wiedergewonnen werden kann (**I** – in Abbildung 6.1), so ist eine Analyse aufgrund der statischen Natur von Speicherplatz und des Vorhandenseins objektiver Maße i.d.R. relativ unproblematisch. Nachdem aktuelle Kennzahlen ermittelt wurden, die die Speicherungsstrukturen beschreiben, kann der Speicherplatz abgeschätzt werden, der für die Aufnahme der derzeit zu speichernden Datenmenge nach einem Neuaufbau der Speicherungsstrukturen benötigt wird (vgl. auch *Anhang B*). Durch den Vergleich der errechneten Werte (die günstigenfalls erreicht werden können) mit den aktuellen Statistikdaten über den reservierten Speicherplatz kann bestimmt werden, wie viel eingestreuter und eliminierbarer Freiplatz innerhalb des Datenbankobjekts vorliegt. Durch eine Reorganisation kann dieser Speicher wieder für Datenbankobjekte frei zur Verfügung gestellt werden. Dies stellt in diesem Fall den Nutzen der Reorganisation dar.

Eine Datenbankreorganisation *muss* immer dann durchgeführt werden, wenn bestimmte, durch die Implementierung des DBMS oder hardware- bzw. betriebssystemseitig vorgegebene Grenzen (*Schwellwerte* – **II** – in Abbildung 6.1), wie z.B. maximale Dateigrößen, maximale Anzahl verwaltbarer Speichereinheiten (z.B. Extents je Tabelle) usw., demnächst erreicht werden. In diesem Fall hat der DBA kaum Entscheidungsfreiheit. Er muss die entsprechenden Schwellwerte identifizieren und – möglichst rechtzeitig („proaktiv“) – geeignete Maßnahmen

durchführen, wie z.B. das Neuanlegen der Tabellen mit deutlich erhöhten Extent-Größen.

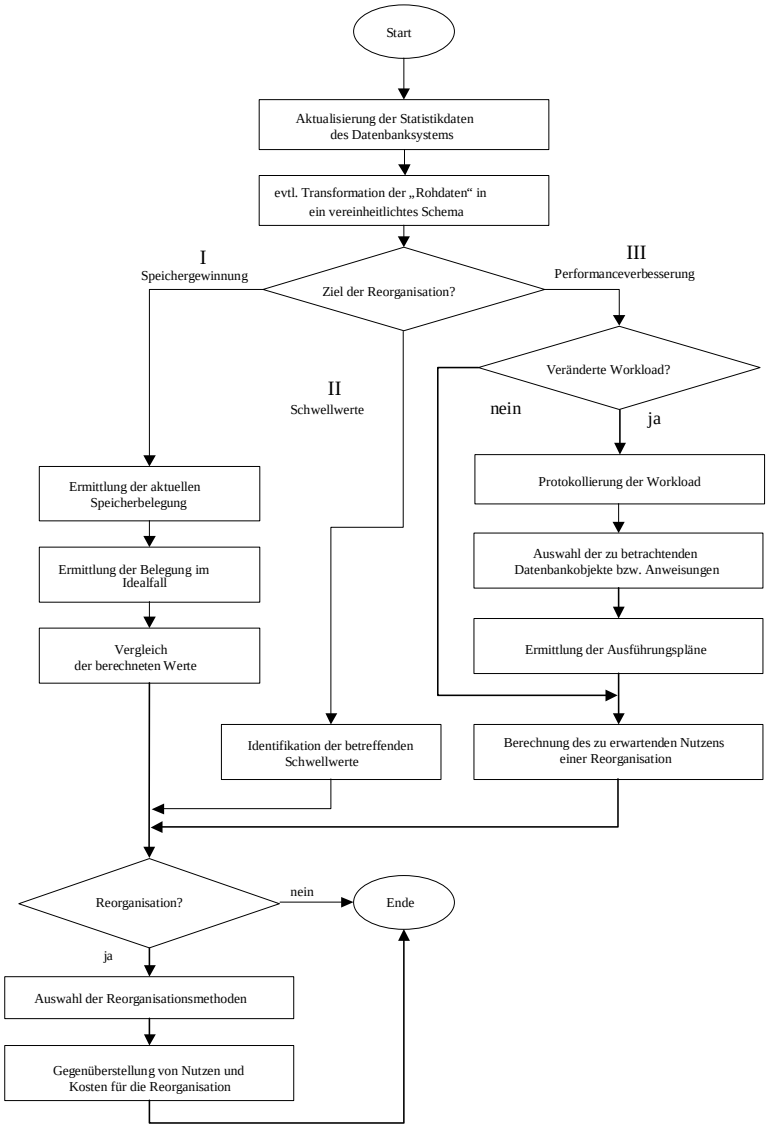


Abbildung 6.1: Ablauf von Reorganisationsbedarfs- und -nutzenanalysen

DBMS- und Betriebssystemhersteller haben in den vergangenen Jahren intensiv daran gearbeitet, solche Grenzen zu überwinden oder die Grenzen so weit zu verschieben, dass sie in der Praxis kaum erreicht werden können. Welche Grenzen existieren und wo die jeweilige Grenze genau liegt, ist systemabhängig und kann allgemein gültig kaum erfasst werden. Hier muss der DBA die in der Systemkonfiguration auftretenden Grenzen ermitteln. Zusätzlich sind geeignete Verfahren einzusetzen, Entwicklungen, die zum Erreichen solcher Grenzen führen können, rechtzeitig zu erkennen und zu überwachen. Einerseits muss eine rechtzeitige Reorganisation sichergestellt werden, andererseits sind die Verfügbarkeitsanforderungen der Anwender weitestgehend zu berücksichtigen.

Wesentlich aufwendiger ist die Durchführung einer Reorganisationsbedarfs- und -nutzenanalyse, wenn eine Einsparung von Verarbeitungskosten und damit eine Verbesserung des Antwortzeitverhaltens (also *Performanceverbesserungen* – **III –** in Abbildung 6.1) Ziele der Reorganisation sind. Auf diesem Szenario liegt der Schwerpunkt der folgenden Betrachtungen.

Die Verringerung der Verarbeitungskosten stellt dabei den Nutzen der Datenbankreorganisation dar. Hier ist ein schrittweises Vorgehen notwendig. Von den vorhandenen Degenerierungen kann nicht direkt auf deren Auswirkungen auf die Systemleistung auf der Anwendungsebene geschlossen werden.

Die von einer Anwendung oder von einzelnen SQL-Anweisungen erzeugte Workload besteht aus einer Abfolge von einzelnen Planoperatoren (z.B. Table Scans, Index Scans etc.), auf die vorhandene Degenerierungen verschiedene Auswirkungen haben. Deshalb reicht bei einer Analyse, bei der festgestellt werden soll, ob eine Performanceverbesserung erreicht werden kann, das alleinige Lokalisieren von Degenerierungen und die Ermittlung des Degenerierungsgrads zur Feststellung eines Reorganisationsbedarfs nicht aus. Es muss ermittelt werden, ob und in welchem Umfang in der gegen die betrachteten Datenbankobjekte gerichteten Workload Operationen enthalten sind, auf die die vorhandenen Degenerierungen negative Auswirkungen haben. Dazu werden zunächst die gegen die Datenbank gerichteten Anweisungen in einem *Anweisungsprotokoll* festgehalten. Für die festgehaltenen Anweisungen können in einer konkreten Systemumgebung mittels des Anfrageoptimierers die Folgen von Planoperatoren (also die *Ausführungspläne*) ermittelt werden, die dort zur Abarbeitung der Workload verwendet werden. Sollen nur bestimmte Datenbankobjekte oder Anweisungen in die Betrachtungen einbezogen werden, so werden nicht alle im Protokoll vorhandenen Anweisungen berücksichtigt. Das heißt, ein Teil der protokollierten Anweisungen wird explizit oder implizit ausgeschlossen.

Die Protokollierung der Anweisungen und die Ermittlung der Planoperatoren muss, im Unterschied zu den weiteren Schritten, nicht zwingend bei jeder Reorganisationsbedarfs- und -nutzenanalyse neu erfolgen, sondern immer nur dann, wenn sich die Workload signifikant ändert.

Anschließend kann der mit der Datenbankreorganisation erreichbare *Nutzen* bestimmt (vorhergesagt, abgeschätzt) werden. Dieser Nutzen wird als prozentuale Verringerung des I/O-Aufwands angegeben, der zur Ausführung der der Analyse zu Grunde liegenden Workload benötigt wird.

Nach der Auswahl geeigneter *Reorganisationsmethoden* für die in die Betrachtungen einbezogenen Datenbankobjekte, können die zur Reorganisationsdurchführung anfallenden *I/O-Kosten* abgeschätzt werden (vgl. *Kapitel 7*). Wenn bekannt ist, welcher Nutzen mit der Reorganisation einzelner Datenbankobjekte erreicht werden kann und welche Kosten dabei jeweils verursacht werden, kann eine Menge von zu reorganisierenden Datenbankobjekten so zusammengestellt werden, dass bei einer gegebenen oberen Grenze für die Reorganisationskosten der erreichbare *Nutzen maximiert* wird. Auf diese Optimierungsfragestellung wird in *Kapitel 8* näher eingegangen.

6.3 Quantifizierung des Nutzens bezüglich der Systemleistung

6.3.1 Überblick

Bei der hier beschriebenen Methode zur Quantifizierung des Nutzens von Datenbankreorganisationen wird workload-bezogen der nach einer Reorganisation anfallende (zu erwartende, reduzierte) I/O-Aufwand ermittelt und ins Verhältnis zum Aufwand vor der Reorganisation gesetzt. Die Nutzenermittlung erfolgt dabei in mehreren Schritten, die nachfolgend im Überblick kurz dargestellt werden (siehe auch Abbildung 6.1 und *Abbildung 6.2*). Detaillierte Betrachtungen zu wichtigen Teilschritten werden dann in *Abschnitt 6.3.2* angestellt.

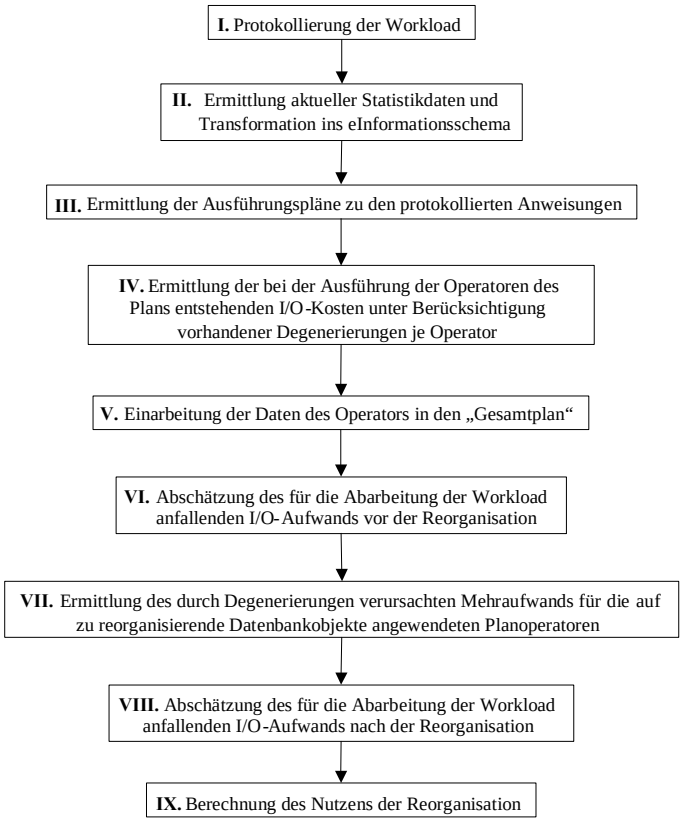


Abbildung 6.2: Ablauf der Ermittlung des Nutzens einer Datenbankreorganisation

So wie Reorganisationen von Datenbanken mehr oder weniger regelmäßig und rechtzeitig vor dem Auftreten von Problemen, die den laufenden Datenbankbetrieb gefährden könnten, erfolgen müssen, sollte auch die Bewertung des zu erwartenden Nutzens in mehr oder weniger regelmäßigen Abständen erfolgen. Werden aus den Nutzenanalysen gewonnene Informationen über einen Zeitraum hinweg gespeichert, können die Entwicklung vorhandener Degenerierungen und des von einer Datenbankreorganisation zu erwartenden Nutzens verfolgt [Wil04] und Trends abgeleitet werden. Abhängig von diesen Trends können Analyseintervalle weitgehend automatisch bestimmt und dem Administrator Vorschläge für durchzuführende Reorganisationen unterbreitet werden. Dieser Ansatz kann als Erweiterungsmöglichkeit von Werkzeugen zur teilweisen automatischen

Administration von Datenbanksystemen angesehen werden, die von DBMS-Herstellern sowie Drittanbietern entwickelt wurden und werden.

Die Grundidee der Methode besteht darin, ausgehend von einem (möglichst auf dem jeweils betrachteten Produktionssystem erstellten) Workload-Protokoll und statistischen Kenngrößen, die Informationen über den Zustand der physischen Speicherungsstrukturen auf dem Produktionssystem liefern, den Nutzen einer Datenbankreorganisation mit rechnerischen Verfahren quantifizieren zu können. Anstelle einer rechnerischen Ermittlung des Nutzens von Datenbankreorganisationen kann auch der Zustand physischer Speicherungsstrukturen nach einer Reorganisation über entsprechende Statistikdaten im Datenbankkatalog „simuliert“[11] werden. Bei diesem Verfahren können die Kosten vor und (anhand der simulierten Strukturen) nach der Reorganisation unter Nutzung des Anfrageoptimierers geschätzt werden.

Dabei werden auch durch die Reorganisation evtl. hervorgerufene Änderungen der Ausführungspläne berücksichtigt. Allerdings ist die Simulation, insbesondere bei Produktivsystemen, oft nur aufwendig zu realisieren, da sie keine negativen Auswirkungen (z.B. ungünstige Ausführungspläne für Anfragen) auf den parallel zur Analyse laufenden Produktivbetrieb haben darf. Bei der Verwendung von Testsystemen, die meist für prinzipielle Funktionstests gedacht sind, ist es über die reinen Statistikdaten hinaus aber i.d.R. nur schwer möglich, die wirklich gleiche Systemumgebung (gleiche Hardware – z.B. vorhandene Datenträger – oder gleiche Konfiguration des DBMS – z.B. Größe des Puffer-Pools – usw.) wie bei den eigentlich zu betrachtenden Produktivsystemen zu erzeugen oder zu simulieren. Allein durch unterschiedliche Konfigurationen von Test- und Produktivsystemen können die Anfrageoptimierer durchaus unterschiedliche Ausführungspläne liefern. Kostenwerte und relative Kostenanteile können sich unterscheiden. Damit lassen sich die auf einem Testsystem ermittelten Nutzenwerte evtl. nicht direkt auf die jeweiligen Produktivsysteme übertragen.

Erschwerend für eine möglichst produktneutrale Lösung kommt die fehlende Standardisierung der Struktur der Daten hinzu, die den physischen Zustand von Datenbankobjekten im Katalog widerspiegeln. Der Norm-Katalog deckt dies ja nicht ab. Außerdem wird der Normkatalog von den meisten DBMS-Produkten nicht vollständig implementiert. Es erscheint hier deshalb günstiger, zur Ermittlung der I/O-Kosten mit dem in Kapitel 5 vorgestellten eigenen Kostenmodell zu arbeiten. Der durch Degenerierungen verursachte Mehraufwand kann dann in den Kostenformeln isoliert und bei der Berechnung der nach einer Reorganisation zur Abarbeitung einer Workload anfallenden voraussichtlichen I/O-Kosten abgezogen werden.

Zur Workload-Ermittlung werden zunächst in einem als repräsentativ anzusehenden Zeitraum die gegen die Datenbank (oder Ausschnitte hiervon) gerichteten (SQL-)

[11] Bei einigen Werkzeugen, die Datenbankadministratoren bei der Indexierung von Datenbeständen unterstützen sollen (Index Advisor, Design Advisor [IBM04] usw.), wird die Nutzung von Workload-Beschreibungen und die Simulation von Indexen durch das Eintragen entsprechender Daten in den Datenbankkatalog bereits angewendet [CN97, CN98]. In diesem Zusammenhang wurden auch Methoden zur Begrenzung der Größe von Workload-Beschreibungen (Workload-Protokollen) entwickelt [CGN02].

Anweisungen und deren Ausführungshäufigkeiten protokolliert (**I** – in Abbildung 6.2).

Anschließend werden aus im Datenbankkatalog vorhandenen Statistikdaten Informationen über den aktuellen („degenerierten“) Zustand der physischen Speicherungsstrukturen ermittelt. Werden vom konkreten DBMS nicht alle Statistikdaten wegen des damit verbundenen Aufwands automatisch und zeitnah gepflegt, muss zuvor noch eine Aktualisierung der Statistikdaten explizit erfolgen (**II**). Für die Quantifizierung des Nutzens von Datenbankreorganisationen sind diese Daten weitgehend ausreichend. Lediglich wenn genaue Analysen für z.B. feine Granulate (Blöcke oder Extents) durchgeführt werden sollen, reichen die im Katalog geführten Informationen oft nicht aus, da für solch feine Granulate üblicherweise keine Statistiken ermittelt werden. Hier steht allerdings auch die Frage nach dem Nutzen solcher Analysen, da eine selektive, zielgerichtete Reorganisation so kleiner Einheiten mit den zur Verfügung stehenden Werkzeugen meist nicht möglich ist. Das momentan üblicherweise feinste Granulat für Datenbankreorganisationen stellen einzelne Partitionen [Now01a] von Tabellen oder Indexen dar.

Bei der DBMS-internen Abarbeitung der in der Workload enthaltenen Anweisungen werden mittels relativ einfacher Grundoperationen (Planoperatoren) die notwendigen I/O-Operationen ausgeführt. In Schritt **III** werden die Anweisungen des Workload-Protokolls an den Anfrageoptimierer zur Ermittlung der jeweiligen Ausführungspläne übergeben. Dessen Aufgabe ist es, wie üblich unter Berücksichtigung vorhandener Statistikdaten Folgen von Planoperatoren (Ausführungspläne) zu finden, mit denen gegebene SQL-Anweisungen *in der konkreten Systemumgebung* auf kostengünstige Weise realisiert werden können.

Anhand des jeweils vom Anfrageoptimierer gelieferten Ausführungsplans kann mit den vorgestellten Kostenfunktionen die Berechnung der durch die Anwendung der jeweiligen Planoperatoren verursachten I/O-Kosten erfolgen (**IV**). Weiterhin muss die Anzahl der Ausführungen des Planoperators (der Wiederholungsfaktor) berücksichtigt werden.

Im Rahmen der Ausführung verschiedener DML-Anweisungen kommen die unterschiedlichen Planoperatoren je Datenbankobjekt in mehreren Ausführungsplänen vor. Zur Begrenzung des Aufwands in den Folgeschritten werden Planoperatoren gleichen Typs, die auf die gleichen Datenbankobjekte angewendet werden, kostenmäßig zusammengefasst. Damit wird eine Liste („Gesamtplan“) aufgebaut (**V**), in der die einzelnen Operatoren, die auf die jeweiligen Datenbankobjekte angewendet werden, jeweils nur einmal vorkommen.

Durch Summierung der Kostenwerte für die im Gesamtplan vorkommenden Planoperatoren kann nun die Anzahl der *vor* einer eventuellen Reorganisation für die Abarbeitung der Workload anfallenden Blockzugriffe berechnet werden (**VI**).

Unter Nutzung der bereits ermittelten aktuellen Statistikdaten kann für die zur Reorganisation vorgesehenen Datenbankobjekte berechnet werden, wie hoch (prozentual) der durch vorhandene Degenerierungen verursachte Mehraufwand für jeden auf das jeweilige Datenbankobjekt angewendeten Planoperator in etwa ist

(**VII**). Zur Abschätzung der I/O-Kosten *nach* einer Reorganisation müssen vor der Summierung noch die Kostenwerte der Planoperatoren, die auf zu reorganisierende Datenbankobjekte angewendet werden, um den in Schritt VII errechneten Mehraufwand verringert werden (**VIII**). Dabei wird hier vereinfachend angenommen, dass die Ausführungspläne vor und nach der Reorganisation gleich sind. Ändert sich allerdings der Ausführungsplan nach der Reorganisation, so kann, eine korrekte Arbeitsweise des Anfrageoptimierers vorausgesetzt, davon ausgegangen werden, dass der Verarbeitungsaufwand noch weiter sinkt. Das heißt, dass das bei der hier vorgestellten Form einer Reorganisationsnutzenbestimmung ermittelte Einsparungspotenzial bei der Reorganisationsdurchführung u.U. übertroffen werden kann.

Dieser Wert wird dann in Schritt **IX** den Kosten *vor* der Reorganisation gegenübergestellt, um die relative Einsparung an Blockzugriffen (den Nutzen der Datenbankreorganisation) zu bestimmen.

6.3.2 Abschätzung des Nutzens von Datenbankreorganisationen im Detail

6.3.2.1 Mögliche Ansätze zur Gewinnung von Workload-Informationen

Eine zunächst einfach und gut geeignet erscheinende Möglichkeit zur Workload-Ermittlung ist die Verwendung des bei Softwareentwicklern und Benutzern *vorhandenen Wissens* über die Anwendung. Bei genauerer Betrachtung ergeben sich hier allerdings Probleme. Softwareentwickler und Benutzer kennen meist nur das Verhalten des Anwendungsteils, an deren Entwicklung sie beteiligt waren bzw. mit deren Nutzung sie befasst sind. Die Erstellung eines Gesamtbildes ist hier mit vertretbarem Aufwand oft nicht realisierbar. Gleiches gilt für den für die Quantifizierung notwendigen Detaillierungsgrad.

Methoden zur Ermittlung von Informationen über die gegen eine Datenbank gerichtete Workload sollten somit bei der zentralen Instanz zur Verwaltung und Verarbeitung der Daten (also dem DBMS) ansetzen. Hier fallen die benötigten Informationen an. Sie müssen nur zugänglich gemacht werden.

DBMS führen typischerweise *Statistiken über erfolgte Blockzugriffe.* Diese werden aber aus Aufwandsgründen meist systemweit oder bezogen auf grobe Granulate (z.B. einzelne Table Spaces oder Files) gesammelt. Für eine auf Datenbankobjekte bezogene Workload-Ermittlung, wie sie in der vorliegenden Arbeit angestrebt wird, sind sie daher eher ungeeignet.

Als weitere mögliche Informationsquelle kommen zunächst auch die *Logs* in Betracht. Für die Quantifizierung des Nutzens von Datenbankreorganisationen sind diese Logs allerdings ungeeignet, da dort typischerweise Retrieval-Operationen (Suchoperationen), die ja einen erheblichen Teil der Workload ausmachen und besonders performancerelevant sind, nicht mit geloggt werden.

Erfolgversprechende Informationsquellen stellen die *an der Anfrageverarbeitung beteiligten Komponenten*, wie die Anfrageanalysekomponente/-compiler (Query Processor) und der Anfrageoptimierer, dar. Da alle Anfragen, die gegen die

Datenbank gerichtet werden, zunächst den Query Processor passieren müssen, können hier Informationen über die einzelnen Anweisungen und über die Häufigkeit ihres Auftretens gesammelt werden. Dazu muss dieser allerdings über geeignete Schnittstellen nach außen verfügen oder es muss eine entsprechende Komponente zur Protokollierung der Anweisungen samt Häufigkeit der Ausführung vorgeschaltet werden. Ausgehend von den vom Query Processor angenommenen Anweisungen ermittelt der Anfrageoptimierer einen Plan für deren kostengünstige Ausführung [Bel96] mittels einzelner Planoperatoren. Auf der Ebene dieser Planoperatoren ist es möglich, die zur Ausführung jeweils anfallenden Kosten abzuschätzen. Die Kostenschätzungen im Rahmen der Anfrageoptimierung werden ebenso durchgeführt.

Natürlich wäre auch eine *Protokollierung der Zugriffe auf der Satz- bzw. Zugriffspfadschnittstelle* des DBMS zur Profilerstellung gut geeignet. Hier sind allerdings, im Unterschied zur Protokollierung der DML-Anweisungen, die ohne größere Probleme auch aufgesetzt realisiert werden kann, Eingriffe in tiefere Ebenen des DBMS nötig, wenn dieses nicht von vornherein über entsprechende Schnittstellen verfügt und diese Schnittstellen extern dokumentiert öffnet. Allerdings würde die Nutzung solcher Schnittstellen auch wieder zu größeren Systemabhängigkeiten als gewünscht führen.

6.3.2.2 Aufbereitung des Anweisungsprotokolls

Aufgrund der meist großen Menge zu bewegender Daten und der auftretenden Behinderungen des Datenbankbetriebs werden Reorganisationen i.d.R. nicht für alle in einer Datenbank gespeicherten Objekte zeitgleich oder in gleichen Intervallen ausgeführt. Üblicherweise wird eine Auswahl bestimmter Datenbankobjekte (*Reorganisationskandidaten*) getroffen. Diese Auswahl kann z.B. zunächst aufgrund von Informationen über den Degenerierungsgrad der Datenbankobjekte oder aufgrund von Aussagen der Benutzer zum Antwortzeitverhalten (bzw. allgemein anhand bestimmter *Workload-Teile*) erfolgen.

Die Auswahl der Datenbankobjekte, die als Reorganisationskandidaten in Betracht kommen, kann *objekt-* oder *anweisungsbezogen* erfolgen. Einen Überblick über die Möglichkeiten zeigt *Abbildung 6.3.*

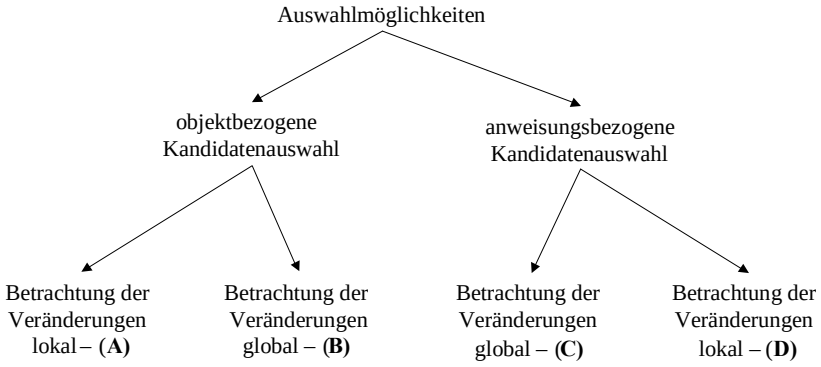


Abbildung 6.3: Auswahl von Reorganisationskandidaten und zu berücksichtigenden Anweisungen

Bei den lokalen Betrachtungen nach einer objektbezogenen Kandidatenauswahl (**A**) werden nur die Planoperatoren einbezogen, die auf die Reorganisationskandidaten angewendet werden, weil z.B. nur die Auswirkungen auf die gegen die Reorganisationskandidaten angewendeten Operationen von Interesse bei der Analyse sind.

Nach einer anweisungsbezogenen Kandidatenauswahl kann lokal (**D**) z.B. betrachtet werden, wie sich die Reorganisation eines oder mehrerer von der betrachteten Anweisung benötigten Datenbankobjekte auf den zur Abarbeitung *dieser* Anweisung notwendigen I/O-Aufwand auswirkt.

Mit den lokalen Betrachtungen kann für Einzelprobleme, wie z.B. unbefriedigendes Leistungsverhalten einer einzelnen Anweisung oder bestimmter Anweisungen, gezielt geprüft werden, ob und inwieweit durch eine Reorganisation der von der Anweisung betroffenen Datenbankobjekte eine Performance-Verbesserung erwartet werden kann. Allerdings werden die Auswirkungen auf die Gesamt-Workload nicht berücksichtigt. So kann z.B. das Beseitigen von in den Datenblöcken einer Tabelle vorhandenem Freiplatz für eine gerade betrachtete sequenzielle Suche über der Tabelle (Table Scan) Leistungsverbesserungen erwarten lassen. Negative Auswirkungen, die die konsequente Beseitigung des Freiplatzes aber auf Insert- und Update-Operationen haben kann, bleiben hier unberücksichtigt.

Die globalen Betrachtungen hingegen zielen auf die durch eine Reorganisation hervorgerufenen Veränderungen des I/O-Aufwands bezüglich der Abarbeitung der gesamten protokollierten Workload ab. Hier kann die Auswahl der Reorganisationskandidaten objekt- (**B**) oder anweisungsbezogen (**C**) erfolgen, was den eigentlichen Unterschied ausmacht, die übrigen Vorgehensweisen bleiben aber jeweils gleich.

6.3.2.3 Mehraufwandsabschätzungen

Über die in *Kapitel 5* und den zugehörigen Anhängen vorgestellten Kostenfunktionen können die vor einer Datenbankreorganisation anfallenden Kosten zur Workload-Abarbeitung berechnet (vorhergesagt) werden. Um die Kosten nach der Reorganisation abschätzen zu können, wird nun für die zu reorganisierenden Datenbankobjekte der durch vorhandene Degenerierungen verursachte Mehraufwand bestimmt (**VI.** in Abbildung 6.2).

Abbildung 6.4 zeigt noch einen Überblick über häufig auftretende Degenerierungen, die vorrangig danach unterteilt wurden, ob sie in Bereichen zur Speicherung der Primärdaten oder in Bereichen zur Speicherung von Zugriffspfaden (Indexen) auftreten. Das soll aber nicht davon ablenken, dass in Datenbereichen auftretende Degenerierungen (z.B. migrierte Tupel oder nicht eingehaltene interne Sortierreihenfolgen) hauptsächlich bei Zugriffen über Indexe Erhöhungen des Verarbeitungsaufwands verursachen. Neben unzureichender Auslastung des Speicherplatzes in Indexbereichen, die auch dazu führen kann, dass sich die Höhe des Indexbaums und damit der Aufwand für Suchen über den jeweiligen Index durch Beseitigung der Freispeicherfragmente verringern ließe, werden die Auswirkungen

von ungültigen physischen Verweisen (Blockadressen) in zu indexorganisierten Tabellen gehörenden Sekundärindexen betrachtet (vgl. *Anhang D*). Auch in Datenbereichen tritt eingestreuter Freiplatz häufig auf und führt zu einer Erhöhung des Verarbeitungsaufwands. Gleiches gilt bei der Verfolgung von Auslagerungszeigern zum Zugriff auf migrierte Tupel sowie das Durchsuchen von Überlaufbereichen, die durch Bucket-Überläufe in Hash-Tabellen oder Clustern entstanden sind. Können die einzelnen Extents eines Segments nicht physisch fortlaufend reserviert werden, kommt es zu einer Erhöhung der Zahl notwendiger Positionierungsoperationen der Lese-/Schreibköpfe und damit zu einer Verlängerung der Antwortzeit von Anfragen. Besonders für Bereichssuchen über Indexe ist es vorteilhaft, wenn die Daten intern nach dem Ordnungskriterium des jeweiligen Indexschlüssels sortiert gespeichert werden. Dadurch wird der Aufwand für das „Springen“ zwischen den Datenblöcken reduziert. Detailliert wurden die einzelnen Degenerierungen bereits in *Kapitel 3* betrachtet.

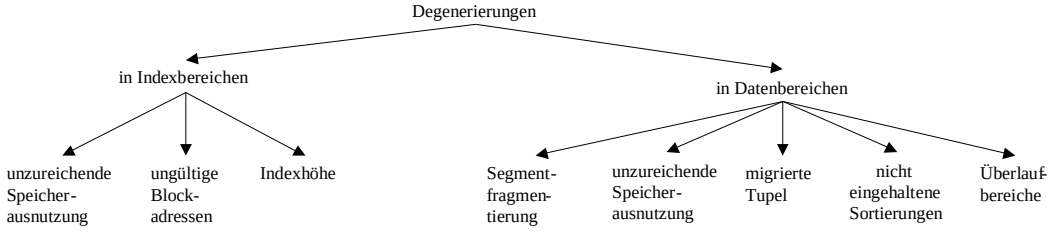


Abbildung 6.4: Häufig auftretende Degenerierungen

In diesem Abschnitt werden für einige ausgewählte Degenerierungen und Planoperatoren Kostenfunktionen vorgestellt, über die die Auswirkungen vorhandener Degenerierungen auf die Verarbeitungskosten bestimmt werden können. Dabei wurden vorrangig Degenerierungen und Planoperatoren ausgewählt, die auch in *Abschnitt 5.4.1* berücksichtigt wurden und die teilweise im Rahmen des in *Abschnitt 6.3.2.6* beschriebenen Beispiels von Bedeutung sind. Weitere Kostenfunktionen zur Abschätzung des von vorhandenen Degenerierungen verursachten Mehraufwands bei der Abarbeitung von Zugriffsoperationen finden sich in *Anhang D*.

Werden die aktuellen Kostenschätzungen für die gegen zu reorganisierende Datenbankobjekte gerichteten Planoperatoren um diesen Mehraufwand verringert, so ergeben sich die nach einer Reorganisation zu erwartenden reduzierten Kosten. Die Veränderung des I/O-Aufwands ist abhängig von den in den physischen Speicherungsstrukturen vorhandenen Degenerierungen und den Planoperatoren, die auf diese Strukturen angewendet werden. In Datenblöcken vorhandener Freispeicher hat z.B. vor allem bei der Anwendung sequenzieller Suchoperationen eine Erhöhung der Anzahl notwendiger Blockzugriffe zur Folge. Die Zahl der Blockzugriffe bei Suchoperationen über Indexe bleibt weitgehend unverändert. Zur Ermittlung des durch vorhandene Degenerierungen zu erwartenden Mehraufwands ist es also notwendig, Planoperatoren und verschiedene Typen und Umfänge von Degenerierungen in ihrem Zusammenhang zu betrachten. Migrierte Tupel (Satzauslagerungen) sowie evtl. vorhandene Überlaufblöcke haben überwiegend Auswirkungen auf Zugriffsoperationen über Indexe (z.B. Index Lookup bzw. Index

Scan). Ziel ist es zunächst, den bei der Anwendung eines in der Workload enthaltenen Planoperators anfallenden prozentualen Mehraufwand an Blockzugriffen „lokal“ zu bestimmen, unabhängig vom Anteil, den der Planoperator am für die gesamte Workload anfallenden I/O-Aufwand ausmacht.

Zur Überprüfung, ob die angegebenen Kostenfunktionen auf reale DBMS anwendbar sind, wurde jeweils der theoretisch errechnete Mehraufwand mit den Ergebnissen von Messreihen verglichen, die unter Nutzung von Oracle durchgeführt wurden. Dazu wurden an jedem Messpunkt statistische Daten über den Zustand der Speicherungsstrukturen (Tupelzahl, Anzahl belegter Blöcke, Indexhöhe usw.) ermittelt. Aus diesen Daten wurde ein Wert für den theoretisch entstehenden I/O-Mehraufwand (in %) berechnet. Dieser wurde mit einem gemessenen Wert verglichen. Alle Messreihen liefen im Prinzip nach dem folgenden gleichen Schema ab:

- Zunächst wurden entsprechende Datenbankobjekte (Tabelle, Indexe, Cluster etc.) neu erstellt und weitestgehend degenerierungsfrei mit künstlich erzeugten Daten gefüllt. Anschließend wurden die für die jeweilige Betrachtung notwendigen Statistikdaten (sozusagen als Basiswerte) ermittelt und festgehalten.
- Danach wurden die jeweils zu betrachtenden Suchoperationen ausgeführt und der während der Ausführung angefallene I/O-Aufwand ermittelt. Auch dieser ermittelte I/O-Aufwand stellt einen Basiswert dar. Er steht für den Aufwand, der bei der Anwendung der entsprechenden Operation auf die betrachteten Datenbankobjekte anfällt, wenn sie frei von Degenerierungen sind.
- Anschließend wurde durch die Ausführung von bestimmten (zielgerichteten) Änderungsoperationen ein Datenbankbetrieb simuliert, bei dem die entsprechenden aktuell betrachteten Degenerierungen entstehen. Diese Simulation wurde zyklisch wiederholt, um den Degenerierungsgrad schrittweise zu erhöhen.
- Nach jedem Simulationszyklus wurden wieder die entsprechenden Statistikdaten für die betrachteten Datenbankobjekte ermittelt. Diese dienen, zusammen mit den Basiswerten, als Grundlage für die Berechnung des durch die Degenerierungen theoretisch hervorgerufenen I/O-Mehraufwands.
- Weiterhin wurde nach jedem Simulationszyklus wieder die zu betrachtenden Suchoperationen ausgeführt und der durch sie verursachte I/O-Aufwand gemessen. Dieser wurde ins Verhältnis zum Basiswert gesetzt. Daraus ergibt sich der gemessene I/O-Mehraufwand. In den in den folgenden Abschnitten dargestellten Diagrammen wird nach jedem Simulationszyklus ein entsprechender Messpunkt gesetzt und der jeweils berechnete und gemessene Mehraufwandswert angegeben.

Die zielgerichtete Erzeugung bestimmter Degenerierungen während der Simulation des Datenbankbetriebs wurde über den jeweiligen Anteil von Einfüge-, Lösch- und Update-Anweisungen innerhalb des zwischen zwei Messpunkten ausgeführten Operationsmix gesteuert. Ein überwiegender Anteil Update-Anweisungen führt i.d.R. zu Satzauslagerungen. Ein hoher Anteil an Löschoperationen führt zur Entstehung von eingestreutem Freiplatz und ein hoher Anteil von Einfügeoperationen zur

Belegung von Überlaufbereichen. Die Entscheidung, ob als nächstes jeweils eine Einfüge-, Lösch- oder Update-Anweisung ausgeführt wird, wurde unter Berücksichtigung des vorgegebenen Verhältnisses der Operationen über einen Zufallszahlengenerator getroffen. Auch zum Erzeugen der einzufügenden bzw. zu ändernden Daten sowie zur Auswahl der Tupel, auf die eine Lösch- oder Update-Anweisung angewendet wird, wurde ein Zufallszahlengenerator verwendet. Durch diese Vorgehensweise war das Degenerierungsverhalten allerdings nicht vollständig steuerbar. Dies ist auch an der teilweise ungleichmäßigen Verteilung der Messpunkte in den folgenden Diagrammen (z.B. Abbildung 6.5) zu erkennen. Zwischen zwei Messpunkten wurde zwar jeweils die gleiche Anzahl Änderungsoperationen ausgeführt, allerdings ergaben sich Unterschiede in der Veränderung des Degenerierungsgrads.

Bei den anschließenden Betrachtungen werden sowohl für die Degenerierungen als auch für die Datenzugriffe Gleichverteilungen angenommen. „Schieflagen“ kann zu einem gewissen Teil durch die Verfeinerung des Analyse- und Reorganisationsgranulats (z.B. auf die Ebene von Partitionen) begegnet werden. Allerdings müssen dann auch entsprechende Statistikdaten vorliegen.

Der von vorhandenen Degenerierungen verursachte prozentuale Mehraufwand (M) kann allgemein gültig mit *Gleichung 6.1* abgeschätzt werden. Dabei steht A_Z für den zusätzlichen Aufwand, der durch Degenerierungen verursacht wird und A_I für den Aufwand im Idealfall, also bei der Anwendung der Operation auf Strukturen ohne Degenerierungen.

$$M = \frac{A_Z}{A_I}$$ Gleichung 6.1

Bei der Ausführung von *Index Lookups* verursachen migrierte Tupel einen Mehraufwand ($M_{LILookup}$) von jeweils einem zusätzlichen Blockzugriff. Dieser kann bei den üblichen B-Baum(artigen)-Indexen unter Berücksichtigung der Anzahl der Ebenen des Index (I_{LEV}), der Anzahl ausgelagerter Sätze (P_{OVFL}) und der Gesamtzahl der Tupel/Sätze (P_{ANZT}) der Tabelle mit *Gleichung 6.2* bestimmt werden.

$$M_{LILookup} = \frac{P_{OVFL}}{P_{ANZT} \cdot (I_{LEV} + 1)} \cdot 100$$ Gleichung 6.2

Da die betrachteten Satzauslagerungen lediglich in den Datenbereichen vorkommen, ist eine Unterscheidung in eindeutige und nicht eindeutige Indexe (unique versus non unique) hier nicht nötig. *Abbildung 6.5* zeigt die Auswirkungen von Satzauslagerungen auf den I/O-Aufwand bei Datenzugriffen nach der Methode Index Lookup.

Der Aufwand für die Datenzugriffe verhält sich erwartungsgemäß proportional zum Anteil ausgelagerter Sätze. Die Höhe des Indexbaums hat dabei Einfluss auf den Anstieg der Mehraufwandskurven. Bei einem höheren Baum ist der Einfluss der Satzauslagerungen geringer als bei einem niedrigen Indexbaum. Basis für Berechnungen und Messungen im in Abbildung 6.5 dargestellten Beispiel ist eine Tabelle mit fester Tupelzahl und ein Index mit zwei Ebenen. Die Satzauslagerungen wurden durch Änderungsoperationen an variabel langen Zeichenketten verursacht.

Für die Aufwandsmessung wurde eine der Tupelzahl der Tabelle entsprechende Anzahl von Zugriffen über den Index auf zufällig ausgewählte Tupel durchgeführt. Der Aufwand für die reinen Datenzugriffe steigt proportional zum Anteil ausgelagerter Sätze. Bei einem Anteil von 30% ausgelagerten Sätzen ergibt sich eine Steigerung der notwendigen Zugriffe auf Datenblöcke um ebenfalls 30%. Durch die bei einem Index Lookup jeweils auch anfallenden Zugriffe auf Indexblöcke, reduziert sich dieser Mehraufwand bei einem Indexbaum mit 2 Ebenen insgesamt gesehen auf 10%.

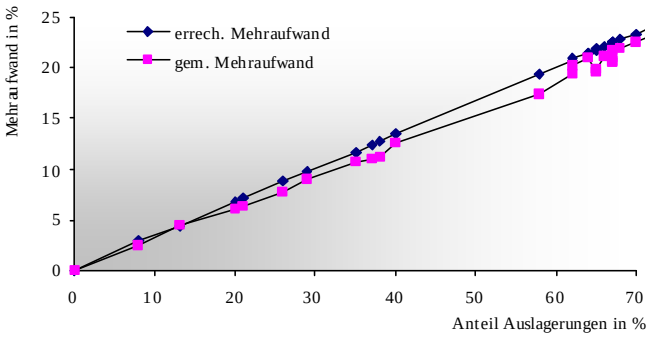


Abbildung 6.5: Auswirkungen von migrierten Tupeln bei Zugriffen über B-Baum-Index, basierend auf gemessenen Blockzugriffen

Wie aus Abbildung 6.5 ersichtlich, entspricht der gemessene Mehraufwand nahezu den berechneten Werten. Die geringen Abweichungen können als Messungenauigkeiten angesehen werden.

Durch die Berücksichtigung der unterschiedlichen Pufferungsgrade von Index- (I_{BRi}) und Datenblöcken (P_{BR}) ergibt sich für den Mehraufwand ($M_{PILookup}$) *Gleichung 6.3*. Unter der üblichen Annahme, dass der Pufferungsgrad von Indexknoten höher ist als der von Datenblöcken, ist zu erwarten, dass der durch ausgelagerte Sätze verursachte Mehraufwand bezüglich physischer Blockzugriffe höher ist als bei der Betrachtung logischer Blockzugriffe.

$$M_{PILookup} = \frac{P_{OVFL} \cdot (1 - P_{BR})}{P_{ANZT} \cdot \left(\underbrace{\sum_{i=1}^{I_{LEV}} (1 - I_{BRi})}_{Indexteil} + \underbrace{(1 - P_{BR})}_{Datenteil} \right)} \cdot 100 \qquad \text{Gleichung 6.3}$$

Aber auch bei *Index Scans* verursacht die Verfolgung eventuell vorhandener Auslagerungszeiger eine Erhöhung der notwendigen Anzahl Blockzugriffe, die mit *Gleichung 6.4* bestimmt werden kann.

$$M_{LIScan} = \frac{P_{OVFL}}{I_{LEV}\text{-}1 \; + I_{LEAF} + P_{ANZT}} \cdot 100 \qquad \text{Gleichung 6.4}$$

Für die Anzahl Blöcke auf der Blattebene steht dabei I_{LEAF}. Die Selektivität kann durch die getroffene Annahme einer Gleichverteilung bezüglich des Auftretens von Degenerierungen unberücksichtigt bleiben. Eine Unterscheidung in eindeutige und

nicht eindeutige Indexe ist hier ebenfalls nicht nötig. *Abbildung 6.6* zeigt die Auswirkungen von Satzauslagerungen auf Index Scans für eine Heap-Tabelle. Dabei entspricht die Vorgehensweise zur Aufnahme der Messreihe im Wesentlichen der beim Index Lookup. Auch hier ist eine hohe Übereinstimmung der berechneten und der gemessenen Mehraufwandswerte zu erkennen.

Der anfallende Mehraufwand entspricht bei Index Scans näherungsweise dem Anteil ausgelagerter Datensätze. Dies liegt daran, dass die wenigen Zugriffe zur Verarbeitung der Indexblöcke im Vergleich zu den Datenblockzugriffen, von denen ja je Satz ein oder zwei anfallen, gering ist. Damit sind auch die Auswirkungen der unterschiedlichen Pufferungsgrade von Index- und Datenblöcken wesentlich geringer als bei Index Lookups. Sollen die Pufferungsgrade trotzdem berücksichtigt werden, so ergibt sich *Gleichung 6.5*.

$$M_{PIScan} = \frac{P_{OVFL} \cdot (1\text{-}P_{BR})}{\underbrace{\sum_{i=1}^{I_{LEV}-1}(1 - I_{BRi}) + I_{LEAF} \cdot (1\text{-}I_{BRh})}_{Indexteil} + \underbrace{P_{ANZT} \cdot (1 - P_{BR})}_{Datenteil}} \cdot 100$$

Gleichung 6.5

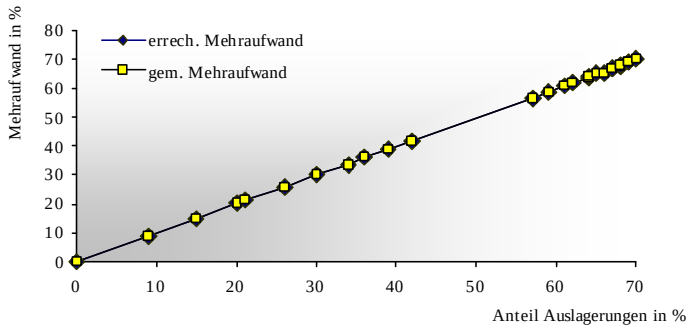


Abbildung 6.6: Auswirkungen von migrierten Tupeln auf Index Scans, basierend auf gemessenen Blockzugriffen

Die *sequenzielle Suche* wird insbesondere durch in Datenblöcke eingestreuten Freiplatz beeinflusst. Durch die damit verbundene Erhöhung der zur Datenspeicherung benötigten Anzahl Blöcke kommt es zu einer entsprechenden Erhöhung der Zahl der Blockzugriffe. Der damit verbundene prozentuale Mehraufwand kann mit *Gleichung 6.6* bestimmt werden.

$$M_{Scan} = \left(\frac{P_{USED} - P_{IDEALTab}}{P_{IDEALTab}} \right) \cdot 100$$

Gleichung 6.6

Er ergibt sich aus dem Verhältnis der Anzahl aktuell belegter Datenblöcke (P_{USED}) zur theoretisch minimal benötigten Anzahl Datenblöcke ($P_{IDEALTab}$), abzüglich des mindestens anfallenden Aufwands. $P_{IDEALTab}$ kann mit den allgemein bekannten Vorgehensweisen zur Abschätzung von benötigtem Speicherplatz (vgl.auch *Anhang B*) berechnet werden. Dabei muss der Freiplatz berücksichtigt werden, der zur Verringerung der Gefahr von zukünftigen Satzauslagerungen bei der Belegung von

Datenblöcken bewusst frei gehalten werden soll (in Oracle über PCTFREE spezifiziert). Das praktische Minimum liegt dann über dem theoretisch denkbaren Minimum. Die Berücksichtigung von Pufferungsgraden kann hier unter der Annahme entfallen, dass diese durch eine Reorganisation nicht wesentlich beeinflusst werden. Abbildung 6.7 zeigt einen Vergleich der errechneten Mehraufwandswerte mit gemessenen Werten. Dabei sind nur geringe Abweichungen zwischen den errechneten und den gemessenen Mehraufwandswerten zu sehen, die auf Messfehler zurückgeführt werden können.

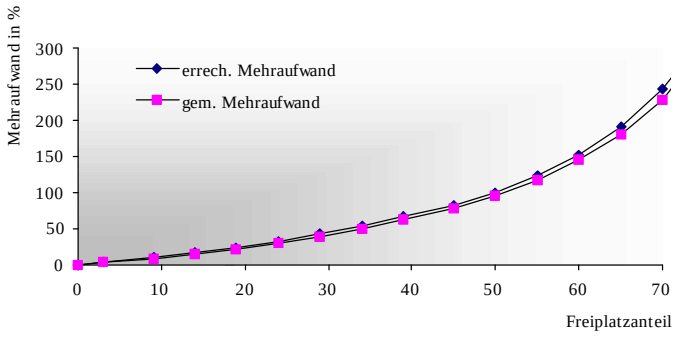


Abbildung 6.7: Auswirkungen von eingestreutem Freiplatz auf sequenzielle Suchoperationen, auf Basis gemessener Blockzugriffe

Der für das Durchsuchen von Überlaufbereichen bei Index Clustern anfallende I/O-Mehraufwand kann mit *Gleichung 6.7* abgeschätzt werden.

$$M_{LICLookup} = \frac{I_{CHAINS}}{\left(\underbrace{I_{LEV}}_{Indexteil} + \underbrace{1}_{Datenteil} \right) \cdot I_{UNQ}} \cdot 100$$

Gleichung 6.7

Abbildung 6.8 zeigt die Auswirkungen der Überlaufketten auf den notwendigen I/O-Aufwand bei Zugriffen auf Sätze in einem Index-Cluster. Während der Messreihen wurde zunächst der Primärbereich des Clusters gleichmäßig mit Daten gefüllt. Die anschließend ermittelten Aufwandswerte und Statistikdaten dienten wieder als Basiswerte. Danach wurden schrittweise weitere Tupel in die Cluster-Struktur eingefügt und somit auf extreme Weise Kollisionen hervorgerufen, die durch die Bildung von Auslagerungsketten behandelt wurden. Dabei sorgte die gleichmäßige Schlüsselverteilung bei den einzufügenden Daten für ein gleichmäßiges Anwachsen der Überlaufketten. Nachdem die Überlaufketten im Mittel um einen Block angewachsen waren, wurden wiederum Datenzugriffe ausgeführt und die entsprechenden Aufwandswerte sowie die Statistikdaten für die weiteren Messpunkte ermittelt. Hier ergibt sich das Problem, dass solche idealen Gleichverteilungen in der Praxis sehr selten sind. Damit treten Abweichungen zwischen den errechneten und real auftretenden Mehraufwandswerten auf. Um aber zu exakteren Abschätzungen des durch die Überlaufbereiche hervorgerufenen Mehraufwands zu gelangen, sind genaue statistische Informationen (z.B. über die einzelnen auftretenden Kettenlängen, Verteilung der Datenzugriffe) erforderlich, die von DBMS-Produkten i.d.R. nicht

geführt[12] und zur Verfügung gestellt werden. Ungenauigkeiten müssen deshalb in Kauf genommen werden.

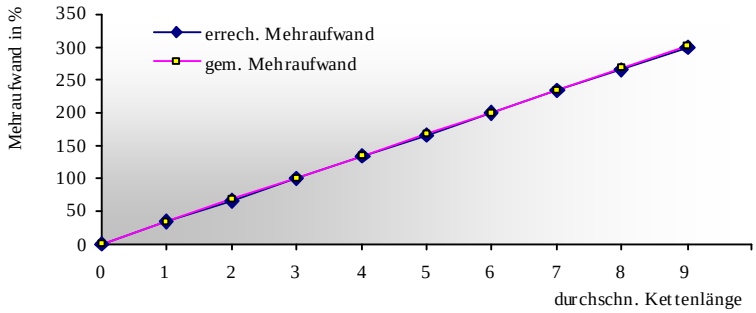


Abbildung 6.8: Auswirkungen von Überlaufbereichen auf Zugriffsoperationen über einen Sekundärschlüssel in einem Index-Cluster, basierend auf gemessenen Blockzugriffen

Bei einer Ebenenzahl im Indexbaum von zwei ergibt sich bei einer durchschnittlichen Kettenlänge von eins ein Mehraufwand von 33%. Bei einem Hash-Cluster würde bei dieser durchschnittlichen Kettenlänge ein Mehraufwand von 100% anfallen (zwei Zugriffe gegenüber einem Zugriff).

Die Berücksichtigung von Pufferungsgraden führt zu *Gleichung 6.8.*

$$M_{PICLookup} = \frac{I_{CHAINS} \cdot (1 - P_{BR})}{\left(\underbrace{\sum_{i=1}^{ILEV} (1 - I_{BR_i})}_{Indexteil} + \underbrace{(1 - P_{BR})}_{Datenteil} \right) \cdot I_{UNQ}} \cdot 100$$

Gleichung 6.8

Bisher wurden nur die Auswirkungen von Degenerierungen auf Suchoperationen betrachtet. Sie haben natürlich auch *Auswirkungen auf die Ausführung von Änderungsoperationen.* Bevor Daten geändert oder gelöscht werden können, müssen sie zunächst aufgefunden werden. Das heißt, auch hier fallen im Rahmen der Update-Operation Suchoperationen an, auf die Degenerierungen Einfluss haben. Zu beachten ist auch, dass bei Änderungs-, Einfüge- oder Löschoperationen sämtliche betroffenen Zugriffspfade aktualisiert werden müssen und damit Suchoperationen in den Indexen anfallen. Die Abschätzung des während der Suchoperationen anfallenden Mehraufwands kann mit den angegebenen Gleichungen erfolgen. Grundsätzlich neue Aspekte sind hier nicht enthalten.

Ob bei der eigentlichen Änderungsoperation ein Mehraufwand anfällt, ist allgemein gültig nur sehr schwer angebbar. So verursacht bspw. in Datenblöcke eingestreuter Freiplatz – wie erwähnt und gezeigt – beim sequenziellen Suchen Mehraufwand. Das Einfügen neuer Datensätze oder Indexeinträge wird durch eingestreuten Freiplatz aber u.U. erleichtert, weil sich die Wahrscheinlichkeit verringert, dass ein neuer Datenblock reserviert werden muss. Mit der Ausnahme von Einfügeoperationen in

[12] Dies würde den Umfang der Statistikdaten extrem „aufblähen“ und eine adäquate Verwendung der Detaildaten durch den jeweiligen DBMS-Optimizer ist zudem fraglich.

Tabellen ohne Indexe sind Änderungsoperationen immer auch mit Suchoperationen verbunden. Oft übersteigt dabei der Suchaufwand den reinen Änderungsaufwand deutlich. Deshalb und weil Mehraufwand, bezogen auf den reinen Änderungsaufwand, nur sehr schwer fassbar ist, wird dieser auf den reinen Änderungsaufwand bezogene Mehraufwand hier nicht berücksichtigt.

6.3.2.4 Nutzenermittlung

Bei der Ermittlung des Nutzens einer eventuell durchzuführenden Datenbankreorganisation werden die geschätzten Kosten vor der Reorganisation und die Mehraufwandswerte zusammengeführt, um die nach der Reorganisation zu erwartenden Kosten für die Abarbeitung der betrachteten Workload bestimmen zu können. Abbildung 6.9 zeigt die Pseudocode-Darstellung einer Funktion zur Ermittlung des Nutzens einer Datenbankreorganisation. Dabei wird auch auf die in Abbildung 6.2 dargestellten Schritte Bezug genommen. Als Parameter werden der Funktion das aufbereitete Workload-Protokoll (`wkl_log`) und eine Liste der Reorganisationskandidaten (`reorg_objects`) übergeben.

Einige Schritte im Pseudocode aus Abbildung 6.9 sollen hier kurz erläutert werden. Nach der Initialisierung des Gesamtplans in Anweisungszeile 5 werden schrittweise für alle Anweisungen des übergebenen Anweisungsprotokolls die Ausführungspläne bestimmt (Zeile 8 bzw. III. in Abbildung 6.2). Kommen in einem Ausführungsplan Operatoren vor, die bereits im Gesamtplan der Workload enthalten sind (Prüfung in Zeile 10), so wird der entsprechende Operator im Gesamtplan lokalisiert (Funktion `finde_gleichen_operator` - Zeile 11).

Anschließend werden die Kostenwerte für den aktuellen Operator ermittelt (**IV.** in Abbildung 6.2) und in den Gesamtplan eingearbeitet (Zeile 12 bzw. **V.** in Abbildung 6.2). Planoperatoren, die noch nicht im Gesamtplan vorkommen, werden samt Kostenwerten neu eingefügt (Zeilen 14 bis 16).

Nachdem der Gesamtplan erstellt wurde, können die Summen (Zeilen 22 bis 31) der anfallenden Blockzugriffe je Planoperator vor (K_{Pvor} - Zeile 23 bzw. **VI.** in Abbildung 6.2) und nach (K_{Pnach} - Zeilen 25 bis 30 bzw. **VII.** und **VIII.** in Abbildung 6.2) der Reorganisation gebildet werden.

Zur Berechnung der Anzahl Blockzugriffe nach der Reorganisation werden dabei für alle Planoperatoren, die auf zu reorganisierende Datenbankobjekte angewendet werden, die Kosten um die Mehraufwandsanteile (M_P), wie in *Gleichung 6.9* dargestellt, verringert (Zeilen 15 und 26).

$$K_{Pnach} = \frac{K_{Pvor}}{1 + M_P / 100}$$

Gleichung 6.9

Abschließend wird in der Anweisungszeile 32 der Nutzen (N) der Datenbankreorganisation, also die durch die Reorganisation zu erwartende prozentuale Veränderung der Anzahl notwendiger Blockzugriffe, über *Gleichung 6.10* berechnet (**IX**. in Abbildung 6.2).

$$N = \left(1 - \frac{K_{Wnach}}{K_{Wvor}}\right) \cdot 100$$ Gleichung 6.10

Dabei stehen K_{Wvor} und K_{Wnach} für die Gesamtkosten zur Workload-Abarbeitung vor und nach der Datenbankreorganisation.

```
1     function nutzen(wkl_log,reorg_objects)
2     begin
3           gesamtplan,plan : list of planop; { Variablen für Gesamt-Plan und Plan
                                                    einer Anweisung }
4           i statement;

5           gesamtplan:=initialisiere_plan();
6           for i in wkl_log do
7                 plan:=initialisiere_plan();

                  { Ausfuerung von Schritt III }
8                 plan.call_optimizer(i);

9                 for j in plan do
                  { Ausfuehrung der Schritte IV und V }
10                      if j in all_planops then
11                            operator:=gesamtplan.finde_gleichen_operator(j);
12                            operator.kosten:=j.cost() * i.anz_ausfuehrungen;
13                      else
14                            all_planops.append(j);
15                            operator:=gesamtplan.finde_gleichen_operator(j);
16                            operator.kosten:=j.cost() * i.anz_ausfuehrungen;
17                      fi;
18                od;
19          od;
20          gesamtkosten_vor_reorg:=0;
21          gesamtkosten_nach_reorg:=0;
22          for k in gesamtplan do
            { Ausfuehrung von Schritt VII }
23                gesamtkosten_vor_reorg:= gesamtkosten_vor_reorg + k.kosten;
            { Ausfuehrung der Schritte VI und VIII }
24                if k.object in reorg_objects then
25                      gesamtkosten_nach_reorg:=    gesamtkosten_nach_reorg +
26                                                   k.kosten/(1 + k.mehraufwand());
27                else
28                      gesamtkosten_nach_reorg:=    gesamtkosten_nach_reorg +
29                                                   k.kosten;
30                fi;
31          od;
            { Ausfuehrung von Schritt IX }
32          nutzen:=(1 - gesamtkosten_nach_reorg / gesamtkosten_vor_reorg) * 100;
33    end;
```

Abbildung 6.9: Pseudocode zur Ermittlung des zu erwartenden Nutzens einer geplanten Datenbankreorganisation

6.3.2.5 Überblick über eine prototypische Referenzimplementierung

Im Rahmen der angestellten Untersuchungen wurde ein einfacher Prototyp eines Werkzeugs zur Quantifizierung des durch Reorganisationen von Datenbankobjekten erreichbaren Nutzens für Oracle implementiert. Abbildung 6.10 zeigt einen Überblick über das Zusammenspiel der beteiligten Komponenten.

Die Protokollierung der gegen die Datenbankobjekte gerichteten Anweisungen muss in einem Zeitraum erfolgen, in dem eine als repräsentativ anzusehende Workload

anfällt. Der durch die Protokollierung anfallende zusätzliche Verarbeitungsaufwand kann als gering angesehen werden, weil die benötigten Informationen bei der Anfrageverarbeitung ohnehin anfallen und die Protokollierung nicht ständig erfolgt. Die weitere Verarbeitung der protokollierten Informationen findet dann unabhängig vom normalen Datenbankbetrieb statt. Sie kann durchaus auch „offline“ erfolgen.

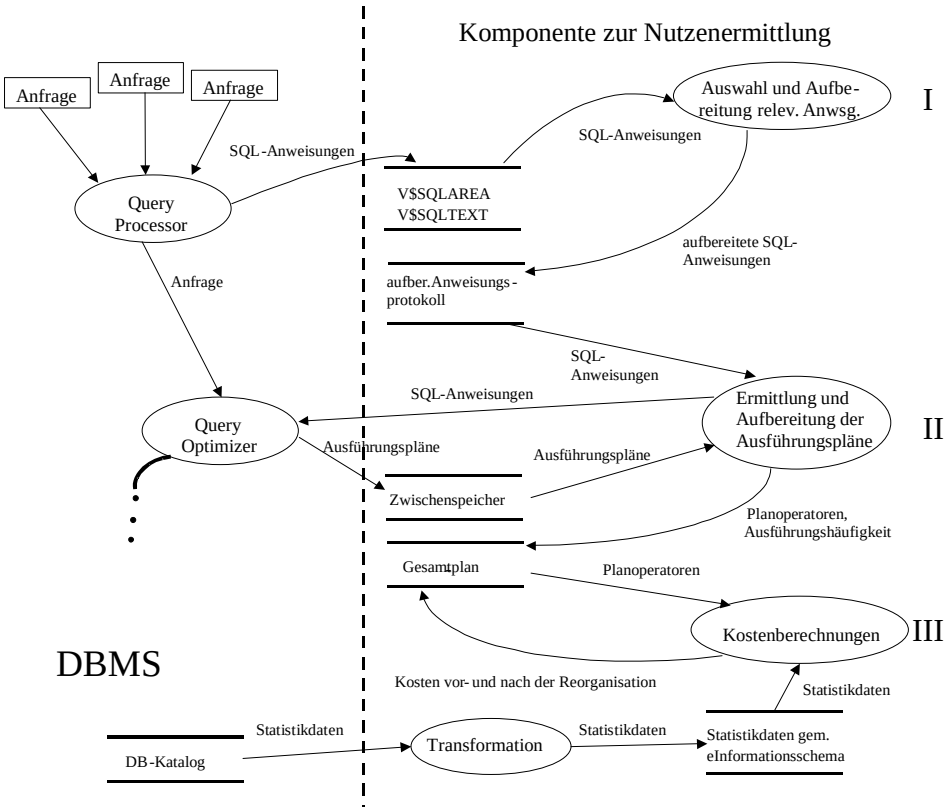


Abbildung 6.10: Komponenten des implementierten Prototyps und deren Zusammenwirken

Zur Erstellung des Anweisungsprotokolls werden die bei Oracle verfügbaren Performance-Sichten `V$SQLAREA` und `V$SQLTEXT` genutzt. Die Sicht `V$SQLAREA` enthält Informationen über die einzelnen im Anweisungs-Cache vorhandenen Anweisungen[13], wie bspw. Versionsnummern, belegte Ressourcen sowie deren Ausführungshäufigkeit. In der Sicht `V$SQLTEXT` werden (u.U. auf mehrere Zeilen verteilt) die SQL-Anweisungstexte gespeichert.

Beide Sichten können je Anweisung über einen Join miteinander verbunden werden. Ist die Anweisung auf mehrere Zeilen verteilt, so werden diese im Attribut `PIECE` nummeriert. *Abbildung 6.11* zeigt die SQL-Anweisung, mit der unter Oracle auf SQL-Texte und deren Ausführungshäufigkeit zugegriffen werden kann und einen Ausschnitt der Ausgabe, die von der Anweisung erzeugt wird.

Problematisch ist bei der Nutzung der Sichten, dass sie eine Schnittstelle zum Anweisungs-Cache darstellen. Damit besteht prinzipiell die Möglichkeit, dass

[13] Beim Einstellen von Anweisungen in den Anweisungs-Cache führt Oracle Zeichenkettenvergleiche durch. Das heißt, dass auch geringfügige Änderungen, wie z.B. das Einfügen von Leerzeichen, zu einem neuen Eintrag führen. Dies ist aber für unsere Zwecke unproblematisch, da für jede der Anweisungen die Anzahl der Ausführungen festgehalten wird.

ausgeführte Anweisungen aus dem Cache verdrängt wurden, bevor sie in die Nutzenermittlung einbezogen werden. Für die Implementierung des Prototyps wurde diese Schnittstelle aber trotzdem verwendet, besonders auch, weil die Informationen über die ausgeführten Anweisungen bereits in relationaler Form vorliegen und diese damit einfach weiterverarbeitet werden können. In [Wil04] werden SQL-Skripte vorgestellt, mit denen dieses Problem durch (rechtzeitiges) zyklisches Durchsuchen der Sichten nach neuen Anweisungen und deren Einarbeitung in das auf Seiten der Komponente zur Nutzenermittlung gehaltene aufbereitete Anweisungsprotokoll verringert werden kann. Für die angestellten Beispielmessungen wurde der SQL-Cache von der Größe her so dimensioniert, dass keine Verdrängungen auftreten.

Zugriff auf die ausgeführten SQL-Anweisungen

```
SELECT T.SQL_TEXT,A.EXECUTIONS,A.ADDRESS,A.HASH_VALUE FROM V$SQLAREA A,V$SQLTEXT T WHERE
A.ADDRESS=T.ADDRESS AND A.HASH_VALUE=T.HASH_VALUE ORDER BY T.HASH_VALUE,T.ADDRESS,T.PIECE;
```

Auszug aus dem Abfrageergebnis

```
SQL_TEXT                                                                  EXECUTIONS ADDRESS  HASH_VALUE
------------------------------------------------------------------------- ---------- -------- ----------
SELECT T.SQL_TEXT,A.EXECUTIONS,A.ADDRESS,A.HASH_VALUE FROM V$SQL                   1 02970BB8 3525743918
AREA A,V$SQLTEXT T WHERE A.ADDRESS=T.ADDRESS AND A.HASH_VALUE=T.                   1 02970BB8 3525743918
HASH_VALUE ORDER BY T.HASH_VALUE,T.ADDRESS,T.PIECE                                 1 02970BB8 3525743918
SELECT R.tmsi, R.lai FROM Teilnehmer T, Routing                                 1221 02763F54 2200198111
WHERE T.msrn= 100001 AND T.msisdn=R.msisdn                                      1221 02763F54 2200198111
```

Abbildung 6.11: Zugriff auf die Informationen für das Anweisungsprotokoll

Um die Informationen über die gegen Datenbankobjekte gerichteten Anweisungen mehrfach verwenden zu können, werden Anweisungstexte und Ausführungshäufigkeiten in aufbereiteter Form gespeichert (**I** in Abbildung 6.10). Dies vereinfacht auch die spätere Weiterverarbeitung. Dabei werden in das aufbereitete Anweisungsprotokoll auch nur die Anweisungen übertragen, die für die späteren Analysen relevant sind (vgl. *Abschnitt 6.3.2.2*). Die Auswahl erfolgt dabei durch den DBA oder auch (zumindest teilweise) bereits bei der Protokollierung, wenn das Protokollierungstool, wie z.B. der Microsoft SQL Server Profiler, es zulässt, die Menge der zu protokollierenden Anweisungen nach verschiedenen Kriterien (z.B. Datenbanknamen, Namen von Datenbankobjekten, Benutzernamen) einzuschränken. *Abbildung 6.12* zeigt einen Ausschnitt aus einem solchen aufbereiteten Protokoll.

```
ST_ID ZNR SQL_TEXT                                          ANZ_AUSF
----- --- ------------------------------------------------ --------
   17   1 SELECT R.tmsi, R.lai FROM Teilnehmer T, Routing R    1221
   17   2 WHERE T.msrn= 100001 AND T.msisdn=R.msisdn           1221
```

Abbildung 6.12: Ausschnitt aus dem aufbereiteten Anweisungsprotokoll

In Schritt **II** (in Abbildung 6.10) erfolgt die Ermittlung der zu den jeweiligen Anweisungen gehörenden Ausführungspläne. Dazu werden die zu analysierenden Anweisungen mit `EXPLAIN PLAN ...` an den Oracle-Anfrageoptimierer übergeben. Dieser ermittelt anhand ihm zur Verfügung stehender Statistik-Daten den kostengünstigsten Ausführungsplan und stellt diesen in relationaler Form (in der Tabelle `PLANTABLE` im Schema des jeweiligen Benutzers) bereit.

Abbildung 6.13 zeigt einen Ausschnitt aus dem für die Anweisung aus Abbildung 6.12 gelieferten Ausführungsplan und dessen Ermittlung.

```
SQLWKS> explain plan for
     2> SELECT R.tmsi, R.lai FROM Teilnehmer T, Routing R WHERE T.msrn= 100001 AND T.msisdn=R.msisdn;
Anweisung verarbeitet
SQLWKS> SELECT operation, options, object_name FROM plan_table;
OPERATION                      OPTIONS                        OBJECT_NAME
------------------------------ ------------------------------ ------------------------------
SELECT STATEMENT
HASH JOIN
TABLE ACCESS                   FULL                           TEILNEHMER
TABLE ACCESS                   FULL                           ROUTING
```

Abbildung 6.13: Auszug aus einem Ausführungsplan

Die vom Anfrageoptimierer gelieferten Informationen werden in den Gesamtplan eingearbeitet. Dieser wird im Prototyp ebenfalls in relationaler Form gespeichert. Abschließend werden für die im Gesamtplan enthaltenen Einträge jeweils die I/O-Kosten für deren Ausführung vor und nach einer Reorganisation berechnet, mit der Ausführungshäufigkeit multipliziert und in den Gesamtplan eingetragen (**III** in Abbildung 6.10). Dazu werden die vorgestellten Kostenfunktionen und aktuelle Statistikdaten, die zuvor in das Format des eInformationsschemas transformiert wurden, genutzt. *Abbildung 6.14* zeigt für die in den vorherigen Abbildungen verwendete Anweisung einen Ausschnitt aus dem Gesamtplan.

```
OBJECT_NAME       OPERATION                      OPTIONS                        ANZ_AUSF   KOSTEN_VORHER KOSTEN_NACHHER MEHRAUFWAND
----------------- ------------------------------ ------------------------------ ---------- ------------- ------------- -----------
TEILNEHMER        TABLE ACCESS                   FULL                               1221       2416359       1993893  ,211879976
ROUTING           TABLE ACCESS                   FULL                               1221       1555554       1488399   ,04511895
```

Abbildung 6.14: Ausschnitt aus dem Gesamtplan

6.3.2.6 Evaluierung an einem praxisnahen Beispiel

Zu Überprüfungszwecken wurde der in *Abbildung 6.15* dargestellte Umweltausschnitt unter Oracle9i und Oracle 10g implementiert. Hier wird ein vereinfachter Ausschnitt der Daten modelliert, die zur Abwicklung von mobiler Kommunikation im Global System for Mobile Communications (GSM) benötigt werden. Im sog. Home Location Register (HLR), einer zentralen Datenbank des Mobilfunkanbieters, werden die Daten über jeden Mobilfunkteilnehmer gespeichert. Dazu gehören u.a. die eigentlichen Teilnehmerdaten, Daten über die Dienste, die vom Mobilfunkanbieter zur Verfügung gestellt und von den Teilnehmern genutzt werden können sowie die Daten über geführte Anrufe, um die entsprechenden Abrechnungen zu erstellen. Im Unterschied zu Festnetzanschlüssen sind die Teilnehmer im Mobilfunkbereich frei beweglich. Diesem Aspekt wird dadurch Rechnung getragen, dass temporär Routing-Daten gespeichert werden, über die mit einer sog. MSISDN[14] die für die

[14] Die MSISDN ist die „normale“ Telefonnummer, die gewählt werden muss, um einen Teilnehmer zu erreichen.

Funkschnittstelle zum Mobiltelefon notwendigen Daten (LAI, TMSI)[15] zugeordnet werden können.

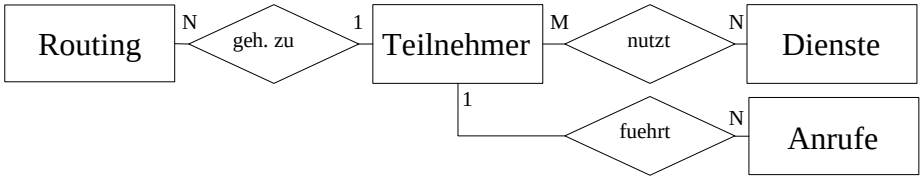


Abbildung 6.15: Umweltausschnitt des Beispiels

Zur Modellierung des in Abbildung 6.15 dargestellten Umweltausschnitts wurden die in Abbildung 6.16 gezeigten Tabellen erstellt. Primärschlüssel sind unterstrichen und Fremdschlüssel kursiv dargestellt. Indexiert wurden die Tabellen jeweils über die Primärschlüssel. Die Tabelle `ANRUFE` wurde noch zusätzlich über den Fremdschlüssel `MSISDN` und das Attribut `Call_dest` indexiert. Der M:N-Beziehungstyp zwischen den Tabellen `TEILNEHMER` und `DIENSTE` wird mit der Tabelle `TNDIENST` realisiert. Danach wurden gezielt Update-Operationen mit dem Zweck ausgeführt, Degenerierungen (z.B. eingestreuten Freiplatz durch Löschoperationen bzw. migrierte Tupel durch Änderungsoperationen) in den Speicherungsstrukturen der Datenbankobjekte zu erzeugen.

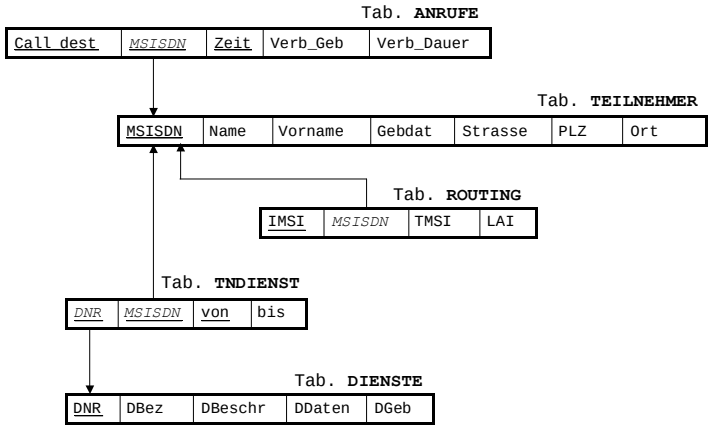


Abbildung 6.16: Tabellenschema des Beispiels

Einen Auszug aus den Statistik-Daten nach dem Erzeugen der Degenerierungen, die dem Katalog der verwendeten Oracle-Datenbank entnommen wurden, zeigt *Tabelle 6.1*. Der Fremdschlüsselindex über der Tabelle `ANRUFE` besitzt drei Ebenen und auf der Blattebene 2634 Knoten. Der Index über `Call_dest` besitzt drei Ebenen und auf der Blattebene 1804 Knoten.

[15] Die Location Area Identity (LAI) und die Temporary Mobile Station Identity (TMSI) dienen zur Identifizierung der Mobilfunkzelle, in der sich der Teilnehmer aufhält und zur Identifizierung des Mobiltelefons in der Zelle.

Tabelle	belegte Blöcke	Freiplatzanteil	Tupelzahl	migrierte Tupel	Zahl der Ebenen der Primärschlüsselindexe	Anz. Blätter
TEILNEHMER	6233	35 %	104762	2844	3	648
DIENSTE	9491	39 %	104789	2903	3	648
ANRUFE	4325	12 %	405005	0	3	3374
ROUTING	3924	12 %	404886	0	3	1802
TNDIENST	3628	13 %	406527	0	3	3241

Tabelle 6.1: Statistiken der Beispieltabellen

Aus den in *Abbildung 6.17* gezeigten SQL-Anweisungen wurden vier unterschiedliche Beispiel-Workloads mit je 5000 Anweisungen erzeugt. Dabei werden in den verschiedenen generierten Workloads nicht immer alle Anweisungen verwendet. Enthalten sind

- in der ersten Workload die Anweisungen 0 bis 6,
- in der zweiten Workload die Anweisungen 0 und 1,
- in Workload Nr. 3 die Anweisungen 0, 4, 5 und 6 und
- in der vierten Workload alle acht aufgeführten Anweisungen.

Ziel ist es, die Workload-Abhängigkeit des Nutzens von Datenbankreorganisationen zu verdeutlichen. Die Reihenfolge der Ausführung der enthaltenen Anweisungen wurde per Zufallszahlengenerator festgelegt.

```
0: SELECT * FROM teilnehmer WHERE msisdn=???;
1: SELECT msisdn, name, vorname, gebdat FROM teilnehmer;
2: SELECT * FROM dienste WHERE dbez=???;
3: SELECT * FROM anrufe WHERE call_dest=???;
4: SELECT T.msisdn, T.name, T.plz, T.ort, T.strasse, A.call_dest,
   A.zeit, A.Verb_geb,A.Verb_dauer FROM teilnehmer T, anrufe A
   WHERE T.msisdn= ??? AND T.msisdn=A.msisdn;
5: SELECT R.tmsi, R.lai FROM teilnehmer T, routing R
   WHERE T.msrn= ??? AND T.msisdn=R.msisdn;
6: SELECT T.name, T.vorname, T.ort, D.dbez, D.ddaten
   FROM teilnehmer T, tndienst TN, dienste D
   WHERE T.msisdn= ??? AND T.msisdn=TN.msisdn AND TN.dnr=d.dnr;
7: SELECT * FROM teilnehmer WHERE msisdn>=??? and msisdn<=???
   order by msisdn;
```

Abbildung 6.17: Anweisungen zur Generierung von Beispiel-Workloads

In der Beispielumgebung werden die Anweisungen aufgrund entsprechender Entscheidungen des kostenbasierten Oracle-Optimierers wie folgt realisiert:

- Bei Anweisung 0 wird ein *Index Lookup* ausgeführt.
- Die Anweisungen 1 und 2 führen jeweils eine *sequenzielle Suche* über den entsprechenden Tabellen aus.

- Anweisung 3 wird über einen *Index Lookup* realisiert.
- Anweisung 4 wird mittels eines *Nested Loop Joins* umgesetzt. Dabei wird zunächst über einen *Index Lookup* auf die Tabelle `TEILNEHMER` zugegriffen. Danach wird über einen *Index Range Scan* die Verbindung zur Tabelle `CALLS` hergestellt.
- Für Anweisung 5 werden die Tabellen `TEILNEHMER` und `ROUTING` zunächst *sequenziell* gelesen und danach über einen *Hash Join* verbunden.
- Bei Anweisung 6 wird zunächst über einen *Index Lookup* eine Selektion auf der Tabelle `TEILNEHMER` durchgeführt. Das Ergebnis wird über einen *Nested Loop Join* mit der Tabelle `TNDIENST` verbunden, auf die mittels sequenziellem Suchen zugegriffen wird. Über einen weiteren *Nested Loop Join* wird anschließend die Verbindung zur Tabelle `DIENSTE` hergestellt, auf die mittels *Index Lookup* zugegriffen wird.
- Anweisung 7 wird über einen *Index Range Scan* über der Tabelle `TEILNEHMER` realisiert.

Für die Beispiel-Workloads wurde der Nutzen einer Reorganisation der Beispieldatenbank wie vorn beschrieben berechnet. Notwendige Selektivitätswerte wurden größtenteils aus den Schätzungen des Anfrageoptimierers übernommen. Das Workload-Protokoll wurde wie in den vorigen Abschnitten beschrieben erzeugt. Eine überschlägige Beispielrechnung für die Workload 2, die Suchen über der Tabelle `TEILNEHMER` ausführt, zeigt *Abbildung 6.18*. Um die Messwerte mit den über Kosten- und Mehraufwandsfunktionen errechneten Werten besser vergleichen zu können, wurden die Werte der Pufferungsgrade in der Beispielrechnung mit 0 angenommen.

Anweisung 0 (Index Lookup) wurde ca. 2500 mal ausgeführt

$$C_P = \left(\underbrace{\sum_{i=1}^{3} (1-0)}_{Indexteil} + \underbrace{\left(1 + \frac{2844}{104762}\right) \cdot (1-0)}_{Datenteil} \right) \cdot 2500 = (3 + 1{,}027) \cdot 2500 = 10068$$

$$M_P = \frac{2844 \cdot (1-0)}{104762 \cdot \left(\underbrace{\sum_{i=1}^{3} (1-0)}_{Indexteil} + \underbrace{(1-0)}_{Datenteil} \right)} \cdot 100 = \frac{2844}{628572} \cdot 100 = 0{,}452\%$$

Berechnung der Kosten nach der Reorganisation

$$K_{P_{nach}} = \frac{10068}{1 + 0{,}00452} = 10023$$

Anweisung 1 (sequenzielle Suche) wurde ca. 2500 mal ausgeführt

$$C_P = 6233 \cdot (1-0) \cdot 2500 = 15582500$$

$$M = \left(\frac{6233}{4047} - 1 \right) \cdot 100 = 54\%$$

Berechnung der Kosten nach der Reorganisation

$$K_{P_{nach}} = \frac{15582500}{1 + 0{,}54} = 10313311$$

Berechnung des **Nutzens** der Datenbankreorganisation

$$N = \left(1 - \frac{10323334}{15592568}\right) \cdot 100 = 33{,}8\%$$

Abbildung 6.18: Beispielrechnung für Workload 2

Zur Überprüfung wurde die jeweilige Workload vor und nach einer Reorganisation auf die Beispieldatenbank angewendet. Dabei wurden die tatsächlich anfallenden logischen und physischen Blockzugriffe unter Nutzung der Zahlen aus den von Oracle zur Verfügung gestellten dynamischen Performance Views gemessen und den berechneten Werten gegenübergestellt. Die Ergebnisse zeigt *Abbildung 6.19*. Die unterschiedlichen Werte für die Workloads 1 und 4 verdeutlichen gut die Workload-Abhängigkeit des Nutzens von Datenbankreorganisationen. Beide Workloads unterscheiden sich darin, dass in Workload 4 die Anweisung Nr. 7 enthalten ist und in Workload 1 nicht. Anweisung 7 macht durch große Scan-Bereiche einen erheblichen Anteil am Gesamtaufwand für die Workload-Abarbeitung aus. Durch die relativ wenigen migrierten Tupel in der Tabelle `TEILNEHMER` ist der vor der Reorganisation bei der Ausführung von Anweisung 7 auftretende Mehraufwand jedoch gering, was sich deutlich auf den (somit geringeren) Nutzen der Datenbankreorganisation auswirkt.

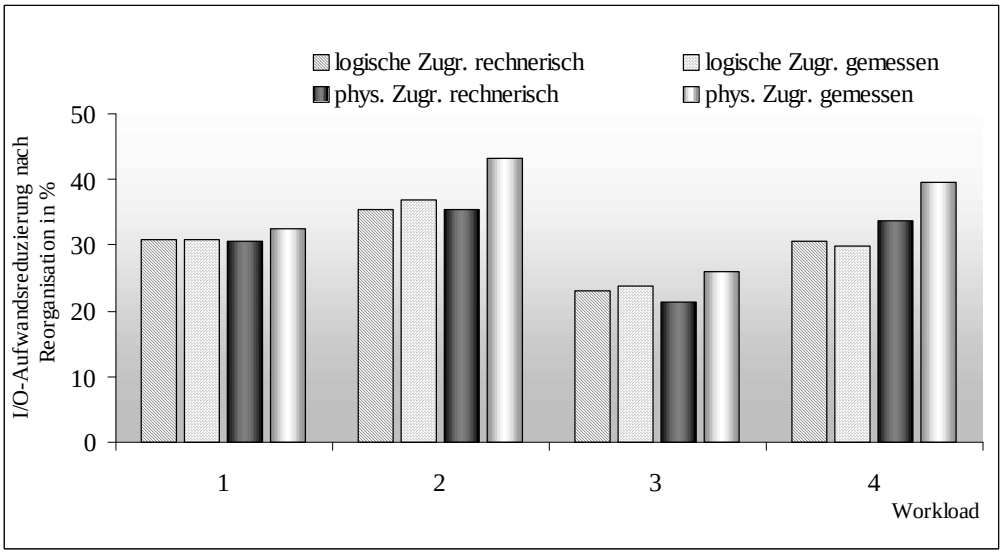


Abbildung 6.19: Vergleich errechneter und gemessener Nutzen von Datenbankreorganisationen bei gleichem Degenerierungsgrad und unterschiedlicher Workload-Zusammensetzung

Die Reorganisation der Datenbankobjekte wurde durch Entladen der Daten, Löschen und erneutes Anlegen der Objekte sowie anschließendes Laden der Daten realisiert. Satzauslagerungen und eingestreuter Freiplatz wurden dadurch beseitigt.

Der Vergleich der errechneten Werte mit dem tatsächlich erreichten Nutzen zeigt, dass es mit der vorgestellten Methode möglich ist, den voraussichtlichen Nutzen einer Datenbankreorganisation relativ genau vorab zu errechnen. Ungenauigkeiten sind hierbei im wesentlichen auf Ungenauigkeiten bei den Abschätzungen des Speicherplatzbedarfs der Tabellen `TEILNEHMER` und `DIENSTE`, deren Tupel jeweils mehrere variabel lange Zeichenketten enthalten, nach der Reorganisation zurückzuführen.

Die Abweichungen der errechneten Werte für physischen Blockzugriffe (jeweils dritte Säule) von den gemessenen Werten (jeweils vierte Säule) sind etwas größer als

bei den Werten für logische Blockzugriffe. Dies liegt hauptsächlich an Ungenauigkeiten bei der Ermittlung von datenbankobjektbezogenen Pufferungsgraden mit der in Abschnitt 5.4.4 vorgestellten Methode. Pufferungsgrade werden hier zu bestimmten Zeitpunkten ermittelt. Durch die Bildung des Mittelwerts der letzten drei Messwerte wird zwar versucht, über längere Zeiträume hinweg auftretende Schwankungen mit zu berücksichtigen und die Ungenauigkeiten zu verringern, allerdings lassen sie sich damit nicht gänzlich vermeiden.

Die Vorgehensweise zur Aufnahme der Messreihen, deren Ergebnisse in *Abbildung 6.20* dargestellt sind, war ähnlich. Allerdings wurde hier immer die Beispiel-Workload Nr. 1 verwendet. Die Reorganisation erfolgte schrittweise. Zunächst wurde nur die Tabelle `ROUTING` reorganisiert, danach die Tabelle `TEILNEHMER` und in einem dritten Durchlauf die übrigen Tabellen. Dabei wurde die teilweise reorganisierte Datenbank jeweils wieder als Ausgangspunkt für die nächste Messung verwendet. Die Summe der errechneten Nutzenwerte entspricht auch nahezu dem Nutzenwert für die Workload 1 in Abbildung 6.19.

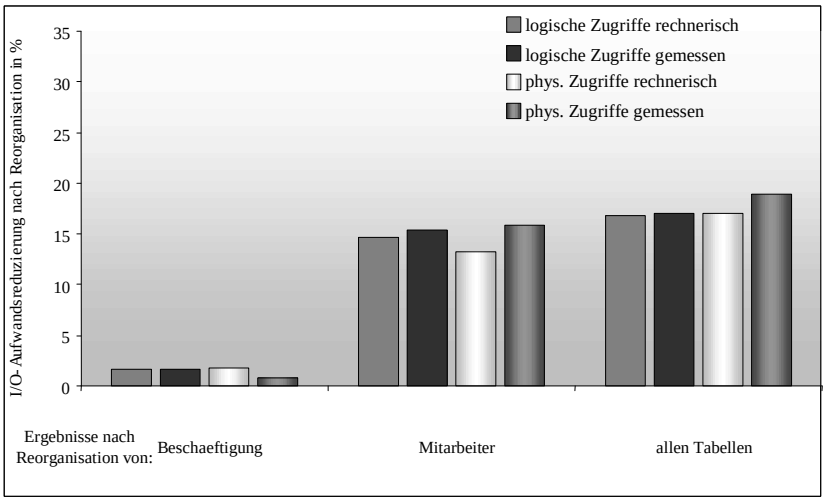


Abbildung 6.20: Vergleich errechneter und gemessener Nutzen von Datenbankreorganisationen bei schrittweiser Reorganisation einzelner Datenbankobjekte

Die Ergebnisse zeigen, dass mit der beschriebenen Vorgehensweise eine Abschätzung des Nutzens einer Datenbankreorganisation auch dann möglich ist, wenn (wie in der Praxis i.d.R. üblich) nur Teile der Datenbank reorganisiert werden.

Unter der Annahme, dass sich der Pufferungsgrad der Blöcke von Datenbankobjekten durch eine Reorganisation nicht oder nur unwesentlich ändert, hat dieser auf die relative Veränderung der I/O-Kosten nur dann einen Einfluss, wenn der Zugriff auf die Tupel einer Tabelle über einen Index erfolgt, da Indexblöcke i.d.R. einen Pufferungsgrad aufweisen, der deutlich höher ist, als der der Datenblöcke. Dies führt dazu, dass sich die Degenerierungen im Datenbereich (wie z.B. ausgelagerte Sätze) relativ gesehen deutlich stärker auf die Antwortzeit auswirken. Würde man also die durch eine Datenbankreorganisation verursachte Veränderung der Antwortzeit betrachten, so wäre diese prozentual teilweise sogar größer als die hier errechneten Werte.

Um sicherzustellen, dass errechnete und gemessene Werte vergleichbar sind, ist es weiterhin wichtig, dafür zu sorgen, dass die Veränderungen der Zahl der Blockzugriffe nur durch die Reorganisation hervorgerufen werden. Veränderungen an den Daten durch Einfüge- bzw. Löschoperationen in der Beispiel-Workload würden ebenfalls zu einer Veränderung des Aufwands führen und damit die Ergebnisse verfälschen. Deshalb enthält die Beispiel-Workload nur Retrieval-Operationen.

7 Schätzung von Reorganisationskosten

Dem von Datenbankreorganisationen zu erwartenden Nutzen sollten auch die Kosten für deren Durchführung gegenübergestellt werden. Die Betrachtungen der vorangegangenen Kapitel haben gezeigt, dass im Wesentlichen drei Methoden bei zyklisch durchzuführenden Reorganisationen von Datenbankobjekten angewendet werden (vgl. auch *Abbildung 2.6*). Dabei kann die datenbanksystembasierte Methode, bei der die Daten exportiert und anschließend wieder importiert werden, nur offline durchgeführt werden. Mit den beiden anderen Methoden (Reorganisation in eine Kopie und In-Place-Reorganisation) werden Verfahren zur Online-Reorganisation betrachtet. In [Wie05] wurden verschiedene Implementierungen der einzelnen Reorganisationsmethoden für die DBMS-Produkte DB2 und Oracle detailliert betrachtet. Die Ergebnisse dieser Arbeit bilden zu einem Teil die Basis der Ausführungen in diesem Kapitel.

Abschnitt 7.1 enthält Beschreibungen der Abläufe bei den verschiedenen Reorganisationsmethoden, bevor in *Abschnitt 7.2* Berechnungsfunktionen zur Abschätzung der anfallenden Kosten entwickelt werden. Besonders bei den beiden Online-Reorganisationsmethoden können sich die Implementierungen verschiedener DBMS-Produkte im Detail unterscheiden. Die Betrachtungen in diesem Kapitel orientieren sich an den Oracle-Implementierungen, sind aber zu großen Teilen auch auf andere Systeme übertragbar. In *Abschnitt 7.3* werden für die einzelnen Reorganisationsmethoden Beispielrechnungen durchgeführt und die Ergebnisse miteinander verglichen. Anhand der Beispielrechnungen ist auch gut erkennbar, wo die Stärken und Schwächen der einzelnen Reorganisationsmethoden liegen.

Da Wartungsmaßnahmen wie Reorganisationen nicht zum normalen Datenbankbetrieb gehören, sind bestimmte Einflussgrößen anders zu berücksichtigen, als im normalen Datenbankbetrieb. So können besonders für den normalen Datenbankbetrieb ermittelte Pufferungsgrade hier oft nicht verwendet werden, weil sich das Zugriffsverhalten auf Daten- und Indexbereiche bei der Reorganisationsdurchführung von dem der sonst auf den Datenbankobjekten arbeitenden Anwendungen unterscheidet. Deshalb arbeiten wir bei der Abschätzung der Reorganisationskosten teilweise mit Annahmen. Dies gilt stellenweise auch bezüglich der Implementierung, da detailliertere Informationen für uns nicht zu erlangen waren. Für die Hersteller von DBMS-Produkten sollte es allerdings kein größeres Problem darstellen, die hier entwickelten Kostenmodelle an die tatsächlichen Gegebenheiten anzupassen. Damit sind dann noch genauere Abschätzungen möglich.

Als Reorganisationsgranulat wird in den folgenden Betrachtungen zur Vereinfachung der Darstellung wieder von einzelnen Tabellen und den zugehörigen Indexen ausgegangen, die auch nicht weiter partitioniert wurden. Die Datenbereiche der Tabellen sind dabei als Heap organisiert.

7.1 Beschreibung der Vorgehensweisen

7.1.1 Datenbanksystembasierte Reorganisation

Ein wesentliches Merkmal datenbanksystembasierter Reorganisationsmethoden ist es, dass die Reorganisationsfunktionalität aufgesetzt auf das DBMS, also basierend auf der normalen externen DBMS-Schnittstelle, realisiert wird. Dazu werden Werkzeuge und Funktionalitäten kombiniert, die nicht primär für Reorganisationszwecke entwickelt wurden. Bei der hier betrachteten Methode werden die Daten der zu reorganisierenden Datenbankobjekte zunächst aus der Datenbank exportiert. Danach werden die zu reorganisierenden Datenbankobjekte gelöscht und neu angelegt (kreiert). Anschließend werden die Daten wieder importiert. Während der Reorganisation stehen die Daten der betroffenen Datenbankobjekte nicht zur Verfügung. Die Vorgehensweise zählt damit zu den Offline-Verfahren. Im konkreten Fall muss jeweils entschieden werden, ob die reorganisationsbedingten Einschränkungen im Benutzerbetrieb hinnehmbar sind. Vorteilhaft ist, dass die Vorgehensweisen für gängige DBMS-Produkte i.d.R. problemlos realisierbar sind, da die Produkte normalerweise über die benötigten Ex- und Importfunktionalitäten verfügen.

Während der Reorganisation stehen die Daten für einen bestimmten Zeitraum nicht unter der Kontrolle durch das DBMS. Damit können die internen Funktionen zur Sicherung der Konsistenz von Daten auch nicht greifen. Deshalb ist es zur Sicherung der Wiederherstellbarkeit im Falle des Auftretens von Fehlern während der Reorganisation dringend zu empfehlen, ein Backup der betroffenen Datenbankobjekte vor Beginn der Reorganisation zu erstellen. Diese Vorgehensweise empfehlen so i.d.R. auch die DBMS-Hersteller.

Abbildung 7.1 zeigt die einzelnen Schritte einer solchen datenbanksystembasierten Reorganisation. Zunächst werden die Daten einer zu reorganisierenden Tabelle gelesen und in eine Exportdatei geschrieben (**I**). Ein gängiges Format für solche Export-Dateien ist das CSV-Format (*Character Separated Values*). Die Daten werden in Textform in der Exportdatei gespeichert. Die einzelnen Attributwerte werden durch Trennzeichen begrenzt. Dies ermöglicht relativ problemlos auch ein Übertragen von Daten zwischen verschiedenen DBMS-Produkten. Je Tupel wird üblicherweise Datensatz verwendet. Einige Exportwerkzeuge verwenden auch proprietäre Formate für die Exportdateien. Dann wird für den Import allerdings in jedem Fall auch ein passendes Importwerkzeug benötigt. Dies stellt bei Reorganisationen zur Beseitigung von Degenerierungen allerdings i.d.R. kein Problem dar, da mit der Reorganisation kein Wechsel des DBMS-Produkts verbunden ist und die DBMS-Hersteller üblicherweise zu den angebotenen Exportwerkzeugen passende Importwerkzeuge zur Verfügung stellen.

Im nächsten Schritt erfolgt das Löschen der Tabellen- und Indexstrukturen (**II**). Dies kann mit den üblichen Datendefinitionsanweisungen effizient realisiert werden. Dabei wird der von den Datenbankobjekten zuvor belegte Speicher freigegeben und die Beschreibungsinformationen über die Tabelle und die jeweiligen Indexe werden aus dem Datenbankkatalog entfernt.

Sollen mehrere Tabellen reorganisiert werden, so müssen die Schritte **I** und **II** für jede Tabelle wiederholt werden.

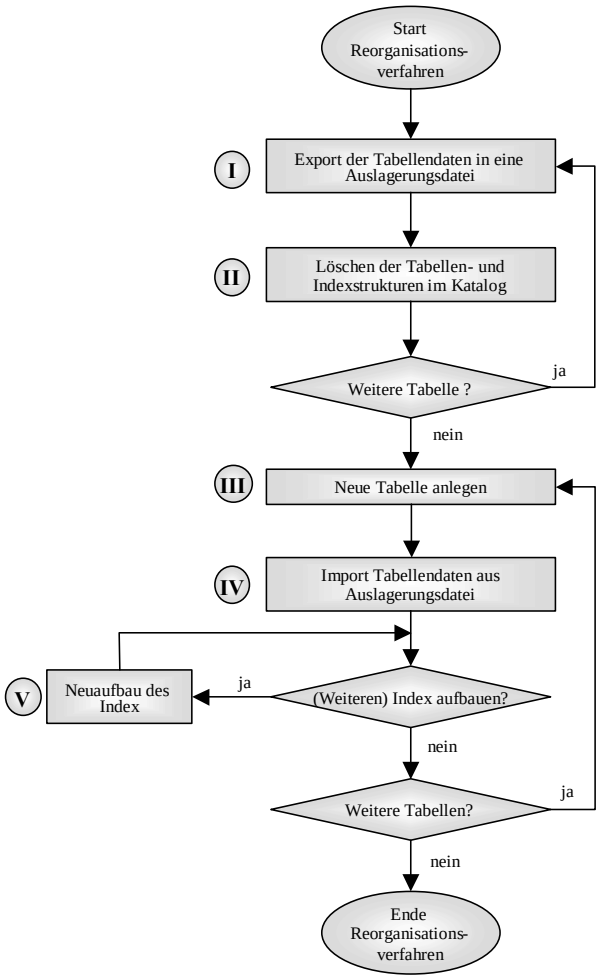


Abbildung 7.1: Datenbanksystembasiertes Verfahren

Nachdem die Daten der zu reorganisierenden Tabellen exportiert und die Tabellen- und Indexstrukturen selbst gelöscht wurden, erfolgt jetzt deren erneutes Anlegen (**III**) und der Import der Daten (**IV**). Hier sollten für jede einzelne Tabelle zunächst alle Schritte ausgeführt werden, bevor die nächste Tabelle samt Indexen neu aufgebaut wird (**V**). Damit kann die Gefahr der Entstehung von Fragmentierungen (hier Segmentfragmentierung, vgl. Abbildung 3.2) während der Reorganisation deutlich verringert werden. Eine Ausnahme bilden hier Cluster, bei denen die Daten mehrerer Tabellen in einem Segment untergebracht werden.

Beim Kreieren der Datenbankobjekte werden die entsprechenden Einträge im Datenbankkatalog erzeugt und Speicherbereiche zur Aufnahme von Daten und auch Indexinformationen reserviert. Hier könnte (wenn möglich) auch die Größe der Speicherbereiche so gewählt werden, dass alle Daten bzw. Indexinformationen in dem jeweils neu reservierten Speicherbereich zusammenhängend untergebracht werden können.

Das Kreieren der Indexe könnte zwar auch vor dem Import der Daten erfolgen, allerdings fällt dann für den Import der Daten, durch die notwendige Pflege der Indexe (Schritt für Schritt), wesentlich mehr Aufwand an. Ein nachträglich durchzuführender Bottom-Up-Aufbau der Indexe kann deutlich effizienter realisiert werden. Entsprechende Funktionalitäten stellen die DBMS i.d.R. zur Verfügung.

Zur Sicherung der späteren Wiederherstellbarkeit sollte nach Abschluss der Reorganisation baldmöglichst ein Backup der reorganisierten Datenbankobjekte erstellt werden. Dies ist notwendig, da sich die physischen Speicherorte der Daten (zumindest der einzelnen Datensätze und Indexeinträge) geändert haben und damit auch die Werte von deren intern verwendeten Identifikatoren (TID/RID/ROWID). Da diese Werte meist auch zur Identifizierung der Sätze im Log verwendet werden, ist ein Anwenden von vor der Reorganisation geschriebenen Log-Einträgen auf von der Reorganisation betroffene Datensätze nicht mehr möglich.

7.1.2 Reorganisation in eine Kopie

Die Abläufe bei Reorganisationen von Datenbankobjekten unter Verwendung einer internen Kopie wurden bereits in *Abschnitt 4.1.5* anhand einer für DB2 vorgeschlagenen Implementierung kurz vorgestellt. Die Ausführungen in diesem Abschnitt orientieren sich an der für Oracle verfügbaren Variante. Die Reorganisationsfunktionalität ist dort im `DBMS_REDEFINITION`-Paket [Ora03e] implementiert. Die Realisierung erfolgt größtenteils mit (erweiterten) SQL-Anweisungen unter Nutzung bei Oracle vorhandener Konzepte.

Zum besseren Verständnis enthält *Anhang E* einen Quelltext mit (erweiterten) SQL-Anweisungen, mit denen die Reorganisationsfunktionalität nachgebildet werden kann. Diese Beispielimplementierung basiert zum Teil auf den Ausführungen in [Gen02]. Dabei wurden auch Vereinfachungen vorgenommen. Die Implementierung erfolgt anhand eines Beispiels und ohne jegliche Fehlerprüfung oder die im `DBMS_REDEFINITION`-Paket enthaltene Behandlung von auf die Originaltabelle verweisenden Referenz-Constraints, Triggern etc.

Neben der Beseitigung von Degenerierungen können mit Hilfe des `DBMS_REDEFINITION`-Pakets auch Veränderungen an der Definition der Struktur der bearbeiteten Tabellen vorgenommen werden. Die Vorgehensweise wird in mehreren technischen Beiträgen skizziert [Gen02, HT05, LS02]. Das Verfahren ist als Online-Verfahren konzipiert. Das heißt, dass die Daten der zu reorganisierenden Tabelle (Originaltabelle) während der Reorganisation nahezu uneingeschränkt zur Verfügung stehen.

Als Schnittstelle zur Reorganisationsfunktionalität werden mehrere Prozeduren zur Verfügung gestellt. Mit der Prozedur

```
DBMS_REDEFINITION.CAN_REDEF_TABLE(...)
```

kann zunächst überprüft werden, ob die Reorganisationsmethode auf eine Tabelle angewendet werden kann. So muss bspw. ein Primärschlüssel für die Tabelle definiert sein[16]. Dieser wird u.a. für das Nacharbeiten zwischenzeitlich an den Daten der Originaltabelle erfolgter Änderungen an den Daten der Kopie benötigt. Die Abläufe während der Reorganisation sind in *Abbildung 7.2* dargestellt.

Zunächst muss eine sog. Interim-Tabelle angelegt werden (**I**). Die Bezeichnung Interim-Tabelle ist dabei etwas irreführend, weil es sich dabei um die Kopie der Ausgangstabelle handelt, an der die Reorganisation vorgenommen wird und die nach deren Abschluss den Platz der Originaltabelle einnimmt. Da der Begriff so aber auch in der Oracle-Dokumentation benutzt wird, soll er auch hier verwendet werden. Die Interim-Tabelle wird mit der gleichen Struktur angelegt wie die Originaltabelle, wenn mit der Reorganisation nur Degenerierungen beseitigt werden sollen. Aus Effizienzgründen sollten noch keine Constraints für die Interim-Tabelle definiert oder Trigger zugeordnet werden. Eventuell vorgesehene Indexe sollten ebenfalls erst später nach der Bottom-Up-Methode aufgebaut werden.

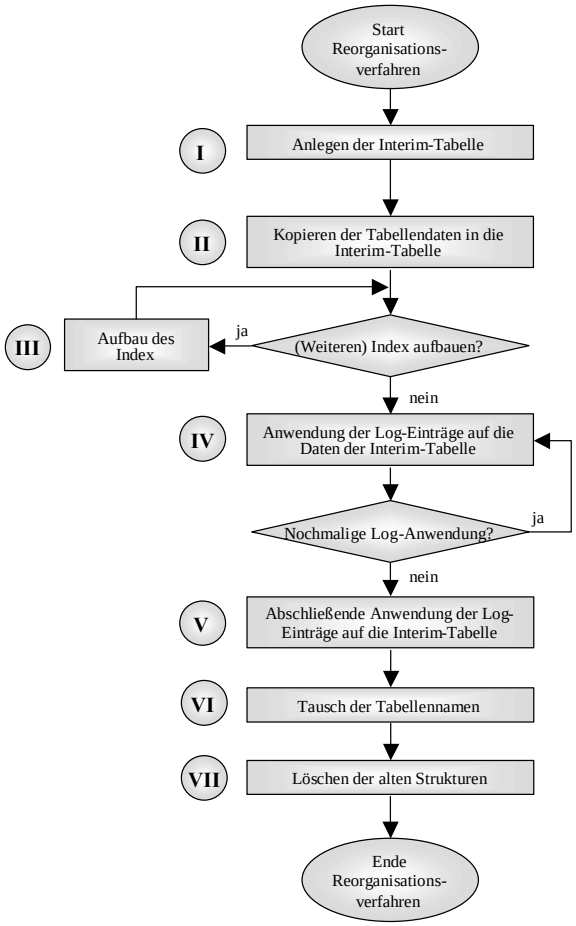


Abbildung 7.2: Reorganisation in eine Kopie

16 Alternativ kann auch ein eindeutiger Index verwendet werden, der keinerlei NULL-Werte enthalten darf [Ora03e].

Nach dem Kreieren der Interim-Tabelle kann die eigentliche Reorganisation mit der Prozedur

```
DBMS_REDEFINITION.START_REDEF_TABLE(...)
```

gestartet werden. Über die Parameter der Prozedur erfolgt die Zuordnung der Originaltabelle zur Interim-Tabelle. Hier kann auch angegeben werden, ob die Daten vor dem Einfügen in die Interim-Tabelle sortiert werden sollen.

Bevor das Kopieren der Daten vorgenommen wird, wird von der Prozedur für die Originaltabelle eine Log-Tabelle (Materialized View Log) erzeugt. Zum Nachvollziehen während der Reorganisation an den Daten der Originaltabelle vorgenommener Änderungen werden hier die bei Oracle verfügbaren Mechanismen zur Aktualisierung von materialisierten Sichten genutzt. In der Log-Tabelle (auf die auch mit normalen SQL-Anweisungen zugegriffen werden kann) wird für jede Änderungsoperation ein Eintrag erzeugt, der den Primärschlüsselwert des betroffenen Datensatzes, einen Zeitstempel sowie weitere Flags zur Beschreibung der Operation enthält.

Anschließend werden die Daten der Originaltabelle in die Interim-Tabelle kopiert (**II**). Damit die während der Reorganisation an den Daten der Originaltabelle vorgenommenen Änderungen später an den Daten der Kopie nachvollzogen werden können, wird unter Nutzung der physischen Speicherungsstrukturen der Interim-Tabelle noch eine materialisierte Sicht erzeugt. Dazu wird die bei Oracle verfügbare Möglichkeit genutzt, eine existierende Tabelle quasi „zur materialisierten Sicht zu erklären“.

Anschließend können die zur Tabelle gehörenden Indexe nach der Bottom-Up-Methode aufgebaut werden (**III**). Auch Constraints können jetzt definiert und Trigger zugeordnet werden. Fremdschlüssel-Constraints müssen allerdings noch deaktiviert bleiben. Sie werden beim Abschluss der Reorganisation von der entsprechenden Prozedur automatisch aktiviert. Datensätze, auf die in der Log-Tabelle enthaltene Einträge anzuwenden sind, werden bei der hier beschriebenen Vorgehensweise in der Originaltabelle und in der Interim-Tabelle über ihren Primärschlüsselwert identifiziert. Deshalb ist es wichtig, dass auch für die Interim-Tabelle ein entsprechender Primärschlüssel definiert wird.

Während die Daten kopiert und Indexe neu aufgebaut werden, haben die Nutzer vollen Zugriff auf die Daten in der Originaltabelle und können damit deren Inhalt verändern. Um diese Änderungen an den Daten der Interim-Tabelle nachzuvollziehen, werden mit der Prozedur

```
DBMS_REDEFINITION.SYNC_INTERIM_TABLE(...)
```

die in der Log-Tabelle aufgezeichneten Einträge auf die Daten der Interim-Tabelle angewendet und anschließend aus der Log-Tabelle entfernt (**IV**). Auch während des Anwendens der Einträge aus der Log-Tabelle können weitere Änderungen an den Daten der Originaltabelle vorgenommen werden, für die wieder Einträge in der Log-Tabelle erzeugt werden. Kann die Reorganisation nicht zeitnah abgeschlossen werden, so muss evtl. ein weiterer Lauf zur Anwendung der neu angefallenen Einträge der Log-Tabelle auf die Daten der Interim Tabelle ausgeführt werden.

Dieses zwischenzeitliche Synchronisieren der Daten der Tabellen muss nicht zwingend durchgeführt werden. Werden die Daten der Originaltabelle bspw. nur selten geändert, so reicht es u.U. aus, wenn die Tabellen beim Abschluss der Reorganisation synchronisiert werden. Allerdings wird die Zeit für den Abschluss der Reorganisation und damit die Zeit, in der Nutzer keinen Zugriff auf die Daten der Tabelle haben, durch zwischenzeitliche Synchronisationen verkürzt.

Der Abschluss der Reorganisation erfolgt mit der Prozedur

```
DBMS_REDEFINITION.FINISH_REDEF_TABLE(...).
```

Dazu wird die Originaltabelle kurzzeitig für weitere Benutzerzugriffe gesperrt. Anschließend erfolgt noch einmal eine Synchronisation beider Tabellen (**V**). Die Inhalte der Tabellen sind jetzt gleich. Durch Veränderungen der Einträge im Datenbankkatalog werden nun die Namen beider Tabellen und der zugehörigen Indexe getauscht (**VI**). Die reorganisierte Tabelle kann wieder für Benutzerzugriffe freigegeben werden. Enthält die Log-Tabelle beim abschließenden Synchronisieren keine oder nur wenige Einträge, so ist der Zeitraum kurz, in dem keine Zugriffe auf die Daten der zu reorganisierenden Tabelle vorgenommen werden können, da für einen Namenstausch nur einige wenige Katalogzugriffe notwendig sind.

Abschließend werden die Log-Tabelle, die Kataloginformationen der materialisierten Sicht über der Interim-Tabelle sowie die alten Index- und Tabellenstrukturen gelöscht (**VII**). Dazu werden die entsprechenden Einträge entfernt, und der von den Datenbankobjekten zuvor belegte Speicher wird freigegeben.

Zur Sicherung der späteren Wiederherstellbarkeit sollte nach Abschluss der Reorganisation baldmöglichst ein Backup der reorganisierten Datenbankobjekte erstellt werden.

Mit der Prozedur

```
DBMS_REDEFINITION.ABORT_REDEF_TABLE(...)
```

kann eine begonnene Reorganisation abgebrochen werden. Dies ist insoweit unkritisch, da lediglich die zwischenzeitlich erzeugten Strukturen (Interim-Tabelle, Log-Tabelle usw.) gelöscht werden. Die Originaltabelle samt Indexen, Constraints etc. bleibt erhalten.

7.1.3 In-Place-Reorganisation

Die beiden zuvor beschriebenen Methoden zur Reorganisation von Datenbankobjekten haben den Nachteil, dass während der Reorganisation ausreichend Speicherplatz für Export-Dateien oder Interim-Tabellen zur Verfügung stehen muss. In-Place-Reorganisationen erfolgen hingegen, wie der Name schon sagt, innerhalb der Speicherbereiche, die von den zu reorganisierenden Datenbankobjekten belegt sind.

Vorrangiges Ziel einer In-Place-Reorganisation kann entweder die Wiederherstellung interner Sortierreihenfolgen sein oder die Beseitigung von migrierten Tupeln und eingestreuten Freiplatzfragmenten, die im Rahmen der Reorganisation zusammengeführt werden. Am Ende der Reorganisation kann der gewonnene

Speicherplatz der Speicherverwaltung wieder zur Verfügung gestellt werden. Die Reorganisation selbst wird über das Verschieben einzelner Datensätze durchgeführt. Die vordergründige Zielstellung der Reorganisation hat dabei hauptsächlich Einfluss auf die Auswahl der zu verschiebenden Sätze und auf die Auswahl der Blöcke, in die Sätze verschoben werden. Unter Oracle werden In-Place-Verfahren zur Beseitigung von eingestreuten Freiplatzfragmenten eingesetzt. Migrierte Tupel werden nicht extra beseitigt. Ihre Zahl reduziert sich während einer Reorganisation lediglich dadurch, dass zum Auffüllen von Datenblöcken auch ausgelagerte Sätze verwendet werden. Da die Verschiebefunktionalität bei Oracle über Löschen und Wiedereinfügen von Sätzen realisiert wird [GK03, Ora03d], werden migrierte Tupel beim Verschieben wieder zu „normalen“ Tupeln. Die Prüfung von Constraints und die Ausführung von Triggern muss dabei unterdrückt werden. Bei DB2 wird das Verschieben von Sätzen über die zwischenzeitliche Verwendung von Auslagerungszeigern realisiert. Diese werden zyklisch in Bereinigungsphasen beseitigt, die in den Reorganisationsprozess eingebettet sind. Detaillierte Beschreibungen dazu finden sich in [Wie05, Win05].

Die Vorgehensweise zur Gewinnung von Speicherplatz zeigt *Abbildung 7.3* schematisch an einer Tabelle mit sechs Datenblöcken. Freier Speicher ist in der Abbildung weiß dargestellt.

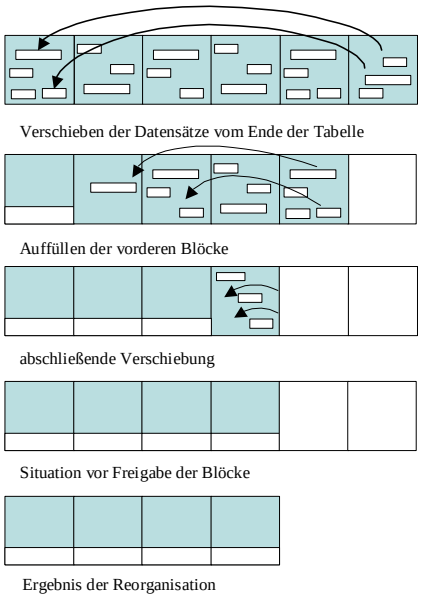


Abbildung 7.3: Verschieben von Datensätzen zur Beseitigung von eingestreutem Freiplatz

Zunächst werden in den vorderen Datenblöcken vorhandene Freispeicherlücken aufgefüllt. Dazu werden Sätze aus den hinteren Datenblöcken nach und nach in die vorderen Blöcke verschoben (wie im oberen Teil von Abbildung 7.3 angedeutet). Beim Auffüllen wird der in den vorderen Blöcken vorhandene Freispeicher zunächst zusammengeführt. Dies ist im Datenbankpuffer effizient realisierbar. Beim Auffüllen der Blöcke werden auch definierte Füllungsgrade berücksichtigt und die vorgesehen

Speicherreserven freigehalten[17]. Durch das Verschieben werden die hinteren Datenblöcke geleert und können am Ende der Reorganisation freigegeben werden.

Im Gegensatz zu den anderen beiden Verfahren werden Indexe im Rahmen der In-Place-Reorganisation einer Tabelle nicht neu aufgebaut. Sie werden innerhalb der Operationskette (Lesen, Einfügen, Löschen) für jeden Datensatz einzeln aktualisiert. Die Vorgehensweise lässt sich auch nicht in größere Schritte unterteilen, wie bei den vorher beschriebenen Verfahren. Vielmehr werden viele *kleinere* Aktionen für einzelne Datensätze jeweils innerhalb einer Transaktion ausgeführt. Das Verfahren ist als Online-Verfahren konzipiert. Das heißt, dass jeweils einzelne Datensätze und Indexknoten (über die normalen Mechanismen des DBMS hierfür) kurzzeitig gesperrt werden müssen, um die Sätze verschieben zu können. In *Abbildung 7.4* sind die einzelnen Aktionen aufgeführt.

Wenn *zusätzlich* zu den von Oracle implementierten Funktionalitäten auch eine Beseitigung von migrierten Tupeln erfolgen soll, müssten die eventuell in einem Block befindlichen Zeiger auf ausgelagerte Sätze beseitigt werden, bevor der Block aufgefüllt wird. Ist über der zu reorganisierenden Tabelle kein Index definiert, so können die Auslagerungszeiger einfach gelöscht werden. Sind Indexe über der Tabelle definiert, so müssen vor dem Löschen der Auslagerungszeiger die entsprechenden Indexeinträge aktualisiert werden. Dazu müssen die Zeiger verfolgt und die entsprechenden Datensätze gelesen werden. Das Lesen der Datensätze ist notwendig, um die Werte der indexierten Attribute zu ermitteln. Über diese Schlüsselwerte kann dann auf die zu aktualisierenden Indexeinträge zugegriffen werden.

Die Beseitigung von Auslagerungszeigern ist in Abbildung 7.4 nicht berücksichtigt, um den eigentlichen Reorganisationsvorgang (das Verschieben der Sätze) in den Vordergrund zu stellen. Zur besseren Darstellung werden die hinteren Blöcke, aus denen jeweils die zu verschiebenden Sätze entnommen werden, mit B_H bezeichnet. Als Bezeichnung für die vorderen Blöcke, in die Sätze eingefügt werden, wird B_V verwendet. Das Verschieben von Datensätzen wird abgeschlossen, wenn es sich bei B_H und B_V um benachbarte Blöcke handelt. Für das gesamte Verfahren wird zur Berücksichtigung der Einflüsse der Datenbankpufferung angenommen, dass Blöcke nur geschrieben werden, wenn sie vollständig aufgefüllt wurden. Für die geleerten Blöcke wird angenommen, dass lediglich die von der Speicherverwaltung verwendeten Informationen (Bitmap-Listen etc.) aktualisiert werden.

Zunächst wird der erste zu leerende Block (B_H) gelesen. Solange dieser Block nicht vollständig geleert ist, wird versucht, nacheinander alle Sätze in weiter vorn liegende Blöcke (B_V) zu verschieben. Dazu wird zunächst ein passender Block gesucht. Die Speicherverwaltung von Oracle führt dazu bspw. intern Bitmap-Listen, aus denen näherungsweise der aktuelle Füllungsgrad von Datenblöcken ohne größeren Aufwand abgelesen werden kann. Ist ein passender Block gefunden, so wird der zu verschiebende Datensatz in diesen eingefügt. Da sich der RID des Datensatzes geändert hat, müssen alle Indexeinträge angepasst werden, die diesen Satz

[17] Dies ist jeweils im unteren Teil der bereits aufgefüllten Blöcke angedeutet.

referenzieren. Wurde ein ausgelagerter Satz nach vorn verschoben, so muss im Rahmen der Aktualisierung der Indexe noch der Auslagerungszeiger beseitigt werden. Anschließend kann der Datensatz aus B_H gelöscht und die Verschiebeoperation abgeschlossen werden. Die an den Datenblöcken und Indexeinträgen während der Verschiebeoperation vorgenommenen Änderungen werden jeweils im Log aufgezeichnet.

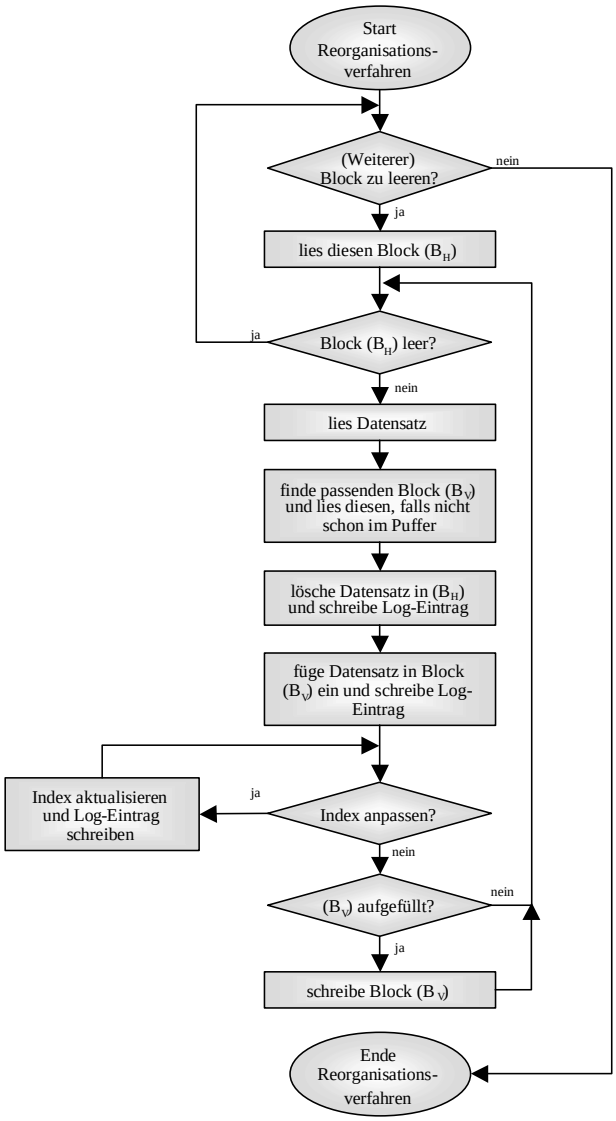


Abbildung 7.4: Abläufe bei einer In-Place-Reorganisation

Wurde durch das Verschieben der Block B_V vollständig aufgefüllt, so wird er anschließend (durch die normalen Mechanismen des DBMS zum Speichern geänderter Blöcke) auf den Datenträger geschrieben. Gleiches gilt für den Block B_H, wenn dieser durch das Verschieben des Satzes vollständig geleert wurde. Es muss dann der nächste zu leerende Block gelesen werden.

Da bei diesem Verfahren alle Operationen durch Log-Einträge dokumentiert werden, ist ein Backup im Rahmen des Reorganisationsverfahrens (vorher und nachher) nicht zwingend erforderlich.

Ein Nachteil des Verfahrens kann sicherlich darin gesehen werden, dass im Rahmen der Reorganisation u.U. nicht alle Degenerierungen beseitigt werden. So werden bspw. durch migrierte Tupel verursachte Indirektionen in Datenbereichen teilweise nicht beseitigt. Im praktischen Einsatz werden aber vor allem die möglichen Behinderungen des laufenden Datenbankbetriebs während der Reorganisation durch den zusätzlichen Verbrauch von Ressourcen und das Auftreten von Sperrkonflikten bei der Entscheidung über die Anwendung von In-Place-Verfahren berücksichtigt werden müssen.

7.2 Kostenabschätzungen

Als Maß für die bei Datenbankreorganisationen anfallenden Kosten werden, wie bei der Quantifizierung des zu erwartenden Nutzens auch, I/O-Kosten verwendet. Damit sind Nutzen- und Kostenwerte vergleichbar[18]. Von den ermittelten Werten können allerdings nicht zwingend direkte Rückschlüsse auf Operationslaufzeiten gezogen werden. Durch eine Parallelisierung bestimmter Operationen (z.B. Backup oder Indexaufbau) können deren Laufzeiten u.U. deutlich verkürzt werden. Die zu bearbeitende Datenmenge und damit der Verarbeitungsaufwand selbst wird durch die Parallelisierung aber nicht verringert.

Bei den folgenden Abschätzungen können Einflüsse der Datenbankpufferung meist nicht über datenbankobjektspezifische Pufferungsgrade berücksichtigt werden, da diese mit der in *Abschnitt 5.4.4* vorgestellten Methode für den normalen Datenbankbetrieb ermittelt werden, der durch die Reorganisation gestört wird. Deshalb werden bei der näherungsweisen Ermittlung der bei Datenbankreorganisationen anfallenden Kosten diesbezüglich Annahmen getroffen, die jeweils erläutert werden.

Ein großer Teil der Kosten wird (eine hinreichende Größe der zu reorganisierenden Datenbankobjekte voraussetzend) durch das Bewegen von Daten sowie durch Pflege bzw. Neuaufbau von Indexen verursacht. Die Kosten für die Aktualisierung von Daten der Freispeicherverwaltung und eine eventuelle Anpassung von Metadaten (bspw. beim Löschen oder Kreieren von Tabellen und Indexen) fallen demgegenüber gering aus, fließen der Vollständigkeit halber aber in die folgenden Betrachtungen mit ein. Allerdings könnten diese Kostenblöcke bei den Abschätzungen auch vernachlässigt werden, ohne das Gesamtergebnis wesentlich zu verfälschen (vgl. auch *Abschnitt 7.3*). Hier fließt auch die Annahme ein, dass besonders die Daten der Freispeicherverwaltung mit hoher Wahrscheinlichkeit im Puffer gehalten werden. Kosten für die Aktualisierung der im Datenbankkatalog enthaltenen Statistikdaten werden aber nicht mit einbezogen.

[18] Ein Schätzwert für die absolute Veränderung (Verringerung) der Kosten zur Workload-Abarbeitung kann im Rahmen der Nutzenermittlung problemlos berechnet werden.

7.2.1 Datenbanksystembasierte Reorganisation

Die I/O-Kosten, die bei der Reorganisation einer Tabelle nach der datenbanksystembasierten Methode über Export und Import von Daten anfallen, lassen sich in folgende Kostenblöcke unterteilen:

- Kosten für das Lesen aller Datensätze der zu reorganisierenden Tabelle und für das Schreiben der Sätze in eine Exportdatei
- Kosten für das Laden der Datensätze aus der Exportdatei
- Kosten für den Neuaufbau der Indexe
- Kosten für das Erstellen des abschließenden Backups
- Kosten für Katalogzugriffe zum Löschen der alten Tabellen- und Indexstrukturen sowie für das Kreieren der Tabellen und Indexe

Die Kosten für die Erstellung eines Backups vor Beginn der Reorganisation, wie es in *Abschnitt 7.1.1* empfohlen wird, werden hier nicht berücksichtigt, da dieses Backup zwar empfehlenswert, für die Reorganisation aber nicht zwingend notwendig ist. Am Ende der Reorganisation muss jedoch ein Backup erstellt werden, um zukünftig die Wiederherstellbarkeit der Tabelle im Fehlerfall zu gewährleisten, da sich im Rahmen der Reorganisation die internen Identifikatoren (RID/TID/ROWID) der Datensätze ändern.

Zum Export der Daten müssen alle Datensätze der Tabelle gelesen und in die Exportdatei geschrieben werden. Für die Leseoperation wird sequenzielles Scannen angenommen. Damit sind alle von der Tabelle belegten Blöcke (P_{USED}) zu lesen. Bei der Abschätzung des Schreibaufwands ist zu berücksichtigen, dass die Exportdatei eine normale Datei auf Betriebssystemebene darstellt. Die vom Betriebssystem verwendete Blockgröße (BS_{PS}) kann sich von der vom DBMS verwendeten Blockgröße unterscheiden. Die Größe BS_{BW} steht für einen Blockungsfaktor. Dieser gibt an, wie viele Blöcke von der Dateiverwaltungskomponente des Betriebssystems bei einer I/O-Operation geschrieben werden, wenn ein sequenzielles Schreiben großer Datenmengen erfolgt. Beim sequenziellen Lesen zusammenhängender Datenbereiche wird der gleiche Blockungsfaktor angenommen.

Die Daten werden beim Export satzweise in die Auslagerungsdatei geschrieben. Die Ermittlung der Satzlänge (LT) kann wie in *Anhang B* beschrieben erfolgen (bspw. mit Gleichung B.1 für Tabellen mit als Heap organisierten Datenbereichen). Aus der Satzlänge und der Anzahl der zu schreibenden Sätze kann die Zahl der Blöcke errechnet werden, die zur Erstellung der Exportdatei zu schreiben sind. Unter Berücksichtigung des Blockungsfaktors für sequenzielle Schreiboperationen können mit *Gleichung 7.1* die Kosten zum Schreiben (bzw. Lesen) der Exportdatei bestimmt werden.

$$C_{WriteExport} = \left\lceil \frac{LT \cdot P_{ANZT}}{BS_{PS} \cdot BS_{BW}} \right\rceil \qquad \text{Gleichung 7.1}$$

Sollen die Daten der Tabelle nach der Reorganisation eine bestimmte interne Sortierung aufweisen, so muss die Datenmenge im Rahmen des Ex- oder Imports

geordnet werden. Zur Sortierung nach dem Ordnungskriterium eines über der Tabelle definierten Index besteht die Möglichkeit (entsprechende Funktionalitäten des Exportwerkzeugs vorausgesetzt), die Daten beim Export mittels eines Index Scans zu lesen. Die Vorgehensweisen zur Schätzung der bei Index Scans anfallenden Kosten (C_{IScan}) wurde in *Abschnitt 5.4.1.2* und *Anhang C* beschrieben.

Ist die Möglichkeit der Durchführung eines Index Scans nicht gegeben, so sind Sortierverfahren auf die Datenmenge anzuwenden. Da hier ganze Datensätze sortiert werden müssen und nicht nur die Werte von einzelnen oder wenigen Attributen, wie bspw. beim Neuaufbau von Indexen, kann die Sortierung meist nicht vollständig im Arbeitsspeicher erfolgen.

Externe Sortierverfahren basieren oft auf der Sort/Merge-Strategie [EN02, KE04]. *Gleichung 7.2* entspricht der in [EN02] angegebenen Gleichung zur Schätzung der Kosten solcher Sortieroperationen, angepasst an die im eInformationsschema verwendeten Größen.

$$C_{Sort} = \underbrace{\left\lceil \frac{P_{USED}}{DB_{PRE}} \right\rceil + P_{TEMP}}_{Sortierphase} + \underbrace{2 \cdot P_{TEMP} \cdot \log_{d_M}(P_{TEMP})}_{Mischphase} \qquad \text{Gleichung 7.2}$$

In der Sortierphase wird die Ausgangstabelle zunächst stückweise in den Arbeitsspeicher gelesen. Die Größe der Stücke ist von der Größe des verfügbaren Puffers abhängig. Die einzelnen Stücke werden im Arbeitsspeicher sortiert und als sog. Runs zwischengespeichert. Im Rahmen der Sortierung wird eingestreuter Freiplatz implizit beseitigt. Deshalb müssen nur P_{TEMP} Blöcke geschrieben werden. Die Berechnung P_{TEMP} erfolgt analog zur Berechnung von P_{IDEAL} (wie in *Anhang B* beschrieben). Allerdings kann ein Füllungsgrad von 100% angenommen werden. Danach folgt die Mischphase. Für jeden Mischvorgang müssen alle Sätze einmal gelesen und geschrieben werden. Die Anzahl der Durchläufe richtet sich nach dem Mischgrad (d_M), der angibt, wie viele Runs in einem Durchlauf gemischt werden können. Nach Abschluss der Sortierung sind die Daten aus dem temporären Bereich, in dem das Mischen erfolgte, in die Exportdatei zu übertragen.

Die verschiedenen Möglichkeiten des Datenexports finden in *Gleichung 7.3* Berücksichtigung.

$$C_{Export} = \begin{cases} \underbrace{\left\lceil \frac{P_{USED}}{DB_{PRE}} \right\rceil}_{Lesen} + \underbrace{C_{WriteExport}}_{Schreiben} & Export\ ohne\ Sortierung \\ \underbrace{C_{Sort} + P_{TEMP}}_{Lesen\ und\ Sortieren} + \underbrace{C_{WriteExport}}_{Schreiben} & Export\ unter\ Nutzung\ externer\ Sortierverfahren \\ \underbrace{C_{IScan}}_{Lesen} + \underbrace{C_{WriteExport}}_{Schreiben} & Export\ unter\ Nutzung\ eines\ Index \end{cases} \qquad \text{Gleichung 7.3}$$

Die Kosten für das Lesen der Ausgangstabelle sind bei der Kostenschätzung für die Variante unter Nutzung externer Sortierverfahren in den Sortierkosten enthalten. Im Rahmen der Kostenermittlung wird auch davon ausgegangen, dass bei der Anwendung sequenzieller Leseoperationen auf Daten- und Indexbereiche

Prefetching-Mechanismen angewendet werden, wenn dies nicht durch zu starke Fragmentierungen verhindert wird. Deshalb ist auch die vom DBMS verwendete Prefetch-Rate (DB_{PRE}) in den entsprechenden Gleichungen enthalten.

Zum Laden der Daten muss nach dem Neuanlegen der Tabelle die Exportdatei gelesen und die einzelnen Datensätze müssen in die neue Tabelle geschrieben werden. Zum Lesen der Exportdatei fällt die gleiche Anzahl Blockzugriffe an wie zum Füllen. Da die neue Tabelle zunächst als frei von Degenerierungen angenommen werden kann, kann deren Größe (P_{IDEAL}) wie in *Anhang B* beschrieben ermittelt werden. Import-Werkzeuge (bspw. der Oracle SQL*Loader) umgehen u.U. auch die normalen Datenbankschnittstellen und schreiben die importierten Daten direkt in die entsprechenden Speicherbereiche [Hei05]. Anschließend werden lediglich die Daten der Freispeicherverwaltung sowie evtl. Statistikdaten aktualisiert. Hierbei können mit einer I/O-Operation durchaus auch mehrere Datenblöcke geschrieben werden. Deshalb wird in *Gleichung 7.4*, mit der die Kosten für den Datenimport errechnet werden können, ein Blockungsfaktor (DB_{BW}) berücksichtigt, der angibt, wie viele Blöcke bei einer I/O-Operation geschrieben werden.

$$C_{Import} = \underbrace{C_{WriteExport}}_{Lesen} + \underbrace{\frac{P_{IDEAL}}{DB_{BW}}}_{Schreiben} \qquad \text{Gleichung 7.4}$$

Im nächsten Schritt werden die Indexe nacheinander (nach der Bottom-Up-Methode) neu aufgebaut. Dazu müssen für jeden Index die Daten der jeweiligen Tabelle gelesen und die jeweiligen Indexierungsschlüssel sortiert werden. Da nur die Werte des jeweiligen Schlüssels und die zugehörigen RIDs sortiert werden müssen, wird angenommen, dass der Puffer hinreichend groß ist und diese Sortierung im Arbeitsspeicher vorgenommen werden kann. Damit fallen für die eigentliche Sortierung keine weiteren I/O-Kosten an. Danach werden die einzelnen Indexeinträge nacheinander in den Blöcken untergebracht, die die Blätter des Indexbaums aufnehmen sollen. Erreicht ein Block den zuvor definierten Füllungsgrad, so muss ein weiterer (Blatt)Block reserviert und im Vaterknoten entsprechend eingebunden werden. Ist der Speicherplatz im Vaterknoten erschöpft, muss auch auf dessen Ebene ein neuer Knoten erzeugt werden, der wiederum in der entsprechenden übergeordneten Ebene vermerkt werden muss etc.

Es wird für die Kostenanalyse angenommen, dass der Puffer hinreichend groß ist, um die jeweils benötigten Knoten der inneren Ebenen während des Aufbaus der Indexe zwischenzuspeichern. Auf Grund der normalerweise relativ geringen Anzahl Knoten in den inneren Ebenen wird der Fall vernachlässigt, dass benötigte Knoten zwischenzeitlich aus dem Puffer verdrängt werden und erneut gelesen werden müssen. Die Anzahl der beim Indexaufbau zu reservierenden und zu schreibenden Blöcke (LI_{IDEAL}) kann wie in *Anhang B* beschrieben ermittelt werden. Die Kosten für den Neuaufbau der Indexe können mit *Gleichung 7.5* abgeschätzt werden. Bezüglich des Schreibaufwands (LI_{IDEAL}) wird angenommen, dass die Baumknoten blockweise geschrieben werden. Die Schreiboperationen erfolgen zwar durchaus asynchron, durch die Baumstruktur dürften aber kaum sequenzielle Schreiboperationen großer Bereiche auftreten, wie bspw. beim Laden großer Datenmengen.

$$C_{CreateIndexe} = \sum_{\forall Indexe} \left(\left\lceil \frac{P_{IDEAL}}{DB_{PRE}} \right\rceil + LI_{IDEAL} \right)$$ Gleichung 7.5

Für die abschließende Backup-Erstellung müssen alle Daten- und Indexblöcke gelesen und in eine Backup-Datei geschrieben werden. Dabei ist zu berücksichtigen, dass vom Betriebssystem beim Schreiben der Backup-Datei evtl. eine andere Blockgröße verwendet wird als vom DBMS. Zur Schätzung der anfallenden Kosten kann *Gleichung 7.6* verwendet werden.

$$C_{BackupEI} = \underbrace{\left\lceil \frac{P_{IDEAL}}{DB_{PRE}} \right\rceil + \sum_{\forall Indexe} \left\lceil \frac{LI_{IDEAL}}{DB_{PRE}} \right\rceil}_{Leseaufwand} + \underbrace{\left\lceil \frac{\left(P_{IDEAL} + \sum_{\forall Indexe} \left(LI_{IDEAL} \right) \right) \cdot DB_{PS}}{BS_{PS} \cdot BS_{BW}} \right\rceil}_{Schreibaufwand}$$ Gleichung 7.6

Das Löschen sowie das Neuanlegen von Tabellen und Indexen wird im Wesentlichen durch Entfernen bzw. Hinzufügen entsprechender Einträge aus dem und in den Datenbankkatalog realisiert. Weiterhin werden Speicherbereiche freigegeben bzw. reserviert. Das erfordert die Aktualisierung interner Strukturen der Freispeicherverwaltung. Die dabei anfallenden Kosten sind von der Struktur des Datenbankkatalogs, der Implementierung der Speicherverwaltung und nicht zuletzt von der Struktur der Datenbankobjekte selbst (z.B. von Anzahl und Typ der einzelnen Attribute) abhängig. So können sich die Kosten selbst bei den verschiedenen zu einer Tabelle gehörenden Indexen leicht unterscheiden. Diese Einflussfaktoren sind allgemein gültig kaum zu erfassen. Deshalb werden hier die Größen $C_{DropTabelle}$ und $C_{DropIndex}$ verwendet, die für die beim Löschen von Tabellen bzw. Indexen anfallenden Kosten stehen. Die Werte der Größen müssen, wenn die Kosten für die Katalogzugriffe (wegen der typischerweise geringen Auswirkungen auf das Gesamtergebnis) nicht einfach vernachlässigt werden sollen, in der konkreten Systemumgebung ermittelt bzw. geschätzt werden. Bezüglich der Kosten für das Anlegen von Datenbankobjekten gelten im Wesentlichen die gleichen Aussagen. Deshalb werden auch hier die Größen $C_{CreateTabelle}$ und $C_{CreateIndex}$ als bekannt vorausgesetzt. Die Kosten für die anfallenden Katalogzugriffe können mit *Gleichung 7.7* ermittelt werden.

$$C_{CatalogEI} = C_{DropTabelle} + C_{CreateTabelle} + \sum_{\forall Indexe} C_{DropIndex} + C_{CreateIndex}$$ Gleichung 7.7

Die Gesamtkosten für die Durchführung einer Reorganisation mit der Export/Import-Methode können mit *Gleichung 7.8* ermittelt werden.

$$C_{ReorgExportImport} = C_{Export} + C_{Import} + C_{CreateIndexe} + C_{BackupEI} + C_{CatalogEI}$$ Gleichung 7.8

7.2.2 Reorganisation in eine Kopie

Bei der hier beispielhaft betrachteten Oracle-Implementierung der Reorganisation von Tabellen in eine Kopie sind die folgenden Kostenblöcke zu berücksichtigen:

- Kosten für das Kopieren der Datensätze in die neue Tabelle
- Kosten für das Schreiben der Einträge in die Log-Tabelle
- Kosten für den Aufbau der Indexe
- Kosten für das Anwenden der Einträge der Log-Tabelle auf die Daten der Interim-Tabelle
- Kosten für die Erstellung des Backups
- Kosten für Katalogzugriffe für das Anlegen einer neuen Tabellenstruktur (Interim-Tabelle) und für das Anlegen und Löschen der Log-Tabelle sowie für das Anlegen der materialisierten Sicht über der Interim-Tabelle, für das Umschalten der Namen und für das Löschen der alten Tabellen- und Indexstrukturen

Zum Kopieren der Daten in die Interim-Tabelle wird die Originaltabelle einmal vollständig gelesen. Die Prozedur

```
DBMS_REDEFINITION.START_REDEF_TABLE(...)
```

ermöglicht eine Sortierung der Daten vor dem Einfügen in die Interim-Tabelle. Dazu werden, wie bei der datenbanksystembasierten Methode, entweder externe Sortierverfahren angewendet oder die sortierte Datenfolge wird durch die Anwendung eines Index Scans beim Lesen der Originaltabelle erreicht. Eine Berücksichtigung der unterschiedlichen Vorgehensweisen erfolgt in *Gleichung 7.9*. Zum eigentlichen Einfügen der Daten in die Interim-Tabelle besteht bei Oracle durch die Angabe von Hinweisen (Hints) an den Optimierer auch hier die Möglichkeit der Umgehung der normalen Datenbankschnittstellen. Das Schreiben der Daten erfolgt dann direkt und sequenziell in die jeweiligen Speicherbereiche. Der Schreibaufwand entspricht damit, von den wenigen (größtenteils im Puffer stattfindenden) Blockzugriffen zur Aktualisierung der Daten der Speicherverwaltung (und evtl. Statistikdaten) abgesehen, der Anzahl Datenblöcke der Interim-Tabelle nach dem Kopiervorgang (P_{IDEAL}).

$$C_{Copy} = \begin{cases} \left\lceil \frac{P_{USED}}{DB_{PRE}} \right\rceil + \frac{P_{IDEAL}}{DB_{BW}} & \textit{einfaches Kopieren} \\ C_{Sort} + P_{TEMP} + \frac{P_{IDEAL}}{DB_{BW}} & \textit{Kopieren mit Sortierung} \\ C_{IScan} + \frac{P_{IDEAL}}{DB_{BW}} & \textit{Kopieren unter Nutzung eines Index} \end{cases} \qquad \text{Gleichung 7.9}$$

Für jede während der Reorganisation an den Daten der Originaltabelle vorgenommene Veränderung wird ein Satz in die Log-Tabelle eingetragen. Der Aufwand für das Einfügen eines Log-Satzes wird mit zwei Blockzugriffen (einer zum Lesen und einer zum Schreiben) angenommen. Durch die Einflüsse der Datenbankpufferung und asynchroner Ausführung von Schreiboperationen stellt diese Zahl eher eine obere Grenze dar. Hier müssen u.U. Annahmen zum Pufferungsgrad der Blöcke der Log-Tabelle berücksichtigt werden, mit denen der über *Gleichung 7.10* ermittelte Aufwand gewichtet wird. Dieser Aufwand ist auch von der Anzahl der auf die Originaltabellen angewendeten Einfüge-, Lösch- und Änderungsoperationen

abhängig, die über die Größen (N_{Insert}, N_{Delete} und N_{Update}) einfließen. Deren Werte müssen vorab (z.B. durch Schätzung) ermittelt werden. Dabei ist zu beachten, dass die Genauigkeit der Schätzung entscheidenden Einfluss auf die Genauigkeit der nachfolgenden Rechnungen hat. Die Ergebnisse der Schätzungen der Kosten für das Logging von an den Daten der Originaltabelle erfolgten Änderungen und die Synchronisation der Tabellen können daher u.U. mit größeren Ungenauigkeiten behaftet sein.

$$C_{Logging\,C} = 2 \cdot \left(N_{Insert} + N_{Delete} + N_{Update}\right) \qquad \text{Gleichung 7.10}$$

Das Aufbauen der Indexe erfolgt, wie auch bei der datenbanksystembasierten Vorgehensweise, mit einzelnen Datendefinitionsanweisungen nach der Bottom-Up-Methode. Zur Abschätzung der Kosten kann *Gleichung 7.5* verwendet werden.

Der nächste Kostenblock umfasst den Aufwand, der beim Anwenden der in der Log-Tabelle enthaltenen Einträge auf die Daten der Interim-Tabelle anfällt. Dazu sind alle Einträge der Log-Tabelle zu lesen. Weil die Log-Tabelle wie eine normale Datenbanktabelle aufgebaut ist, kann die Anzahl der von ihr belegten Blöcke ($P_{USEDLogTab}$ – mit den in *Anhang B* angegebenen Vorgehensweisen) geschätzt werden. Die Anzahl der Tupel in der Log-Tabelle entspricht der Anzahl aller während der Reorganisation auf die Originaltabelle angewendeten Einfüge-, Lösch- und Änderungsoperationen. Die Log-Tabelle enthält neben dem Primärschlüssel der Originaltabelle noch weitere Attribute zur Beschreibung der mit den einzelnen Einträgen jeweils verbundenen Änderung.

Für jede in der Log-Tabelle aufgezeichnete Einfüge- und Änderungsoperation wird über den über den Primärschlüsselindex das jeweilige Tupel der Originaltabelle (quasi als Image) gelesen und die Änderungen werden dann an den Daten der Interim-Tabelle nachvollzogen. Die Ermittlung der dabei anfallenden Kosten erfolgt wie in *Abschnitt 5.4.2* bzw. *Anhang C* beschrieben. Mit *Gleichung 7.11* können die Kosten für das Nachvollziehen der während der Reorganisation an den Daten der Originaltabelle vorgenommenen Änderungen abgeschätzt werden.

$$C_{Sync} = \left\lceil \frac{\underbrace{P_{USEDLogTab} + \left(N_{Insert} + N_{Update}\right) \cdot C_{ILookupUOrigTab} +}_{Lesen\,der\,Log-Tabelle\,und\,der\,Images}}{\underbrace{+ N_{Insert} \cdot C_{Insert} + N_{Delete} \cdot C_{Delete} + N_{Update} \cdot C_{Update}}_{Anwenden\,der\,Log-Einträge}} \right\rceil \qquad \text{Gleichung 7.11}$$

Bei der Berechnung der Kosten für die abschließende Erstellung des Backups ist in *Gleichung 7.12* zu berücksichtigen, dass die Tabelle durch evtl. erfolgte Einfügeoperationen mehr Blöcke umfassen kann, als vorab mit P_{IDEAL} errechnet. Bezüglich der Indexe wird angenommen, dass durch eine überlegte Wahl des Füllungsgrads für Indexknoten während des Neuaufbaus beim Nacharbeiten der Log-Einträge keine Knoten geteilt werden müssen. Die in Gleichung 7.12 benötigte Anzahl Sätze, die in einem Block untergebracht werden können (*TP*), ist mit *Gleichung B.2* (auch im Rahmen der Ermittlung von P_{IDEAL}) zu ermitteln.

$$C_{BackupC} = \underbrace{\left\lceil \frac{P_{IDEAL}}{DB_{PRE}} \right\rceil + \sum_{\forall Indexe} \left\lceil \frac{LI_{IDEAL}}{DB_{PRE}} \right\rceil + \underbrace{\left\lceil \frac{N_{Insert}}{TP \cdot DB_{PRE}} \right\rceil}_{neu\ eingefügte\ Sätze} +}_{Leseaufwand}$$

$$\underbrace{+ \left\lceil \frac{\left(P_{IDEAL} + \sum_{\forall Indexe} \left(LI_{IDEAL} \right) + \left\lceil \frac{N_{Insert}}{TP} \right\rceil \right) \cdot DB_{PS}}{BS_{PS} \cdot BS_{BW}} \right\rceil}_{Schreibaufwand} \qquad \text{Gleichung 7.12}$$

Die durch notwendige Katalogzugriffe verursachten Kosten können mit *Gleichung 7.13* abgeschätzt werden. Zunächst sind die für die Reorganisationsdurchführung benötigten Strukturen (Interim-Tabelle, Log-Tabelle und die materialisierte Sicht über den Strukturen der Interim-Tabelle) anzulegen. Die Größe ($C_{CreateView}$) steht für die Anzahl Blockzugriffe, die beim Kreieren der materialisierten Sicht über den Strukturen der Interim-Tabelle anfallen. Der Tausch der Namen von Tabellen und Indexen erfolgt über Änderungen der entsprechenden Einträge im Datenbankkatalog. Die Größen $C_{RenameTabelle}$ und $C_{RenameIndex}$ stehen dabei in Gleichung 7.13 für die Kosten für das Umbenennen einer Tabelle bzw. eines Index. N_{INDEXE} steht für die Anzahl der Indexe, die über der zu reorganisierenden Tabelle definiert sind. Insgesamt sind für den Tausch der Namen jeweils drei Umbenennungen nötig. Abschließend müssen Katalogzugriffe für das Löschen der alten Tabellen- und Indexstrukturen, der Log-Tabelle und der Beschreibungsinformationen der materialisierten Sicht erfolgen.

$$C_{CatalogC} = \underbrace{2 \cdot C_{CreateTabelle} + C_{CreateView}}_{Anlegen\ der\ ben.\ Strukturen} + \underbrace{3 \cdot \left(C_{RenameTabelle} + N_{INDEXE} \cdot C_{RenameIndex} \right)}_{Tausch\ der\ Namen} + \qquad \text{Gleichung 7.13}$$

$$+ \underbrace{2 \cdot C_{DropTabelle} + C_{DropView} + N_{INDEXE} \cdot C_{DropIndex}}_{Löschen\ der\ alten\ bzw.\ nicht\ mehr\ benötigten\ Strukturen}$$

Die Gesamtkosten für die Durchführung einer Reorganisation in eine Kopie kann mit *Gleichung 7.14* ermittelt werden.

$$C_{ReorgKopie} = C_{Copy} + C_{LoggingC} + C_{CreateIndexe} + C_{Sync} + C_{BackupC} + C_{CatalogC} \qquad \text{Gleichung 7.14}$$

7.2.3 In-Place-Reorganisation

Für die von Oracle implementierte Methode zur In-Place-Reorganisation lässt sich der anfallende Aufwand relativ einfach bestimmen, wenn die Zahl der zu verschiebenden Datensätze bekannt ist. Durchgeführte Untersuchungen und Messungen haben auch die beschriebene Vorgehensweise größtenteils bestätigt, bei der Datensätze aus den „hinteren“ Blöcken der Tabelle zum Auffüllen weiter vorn gelegener Blöcke verwendet werden. Zur Abschätzung der Anzahl zu verschiebender Sätze wird zunächst die Anzahl der frei werdenden Datenblöcke benötigt. Mit der in *Anhang B* beschriebenen Vorgehensweise kann die Zahl der nach der Reorganisation belegten Datenblöcke (P_{IDEAL}) näherungsweise bestimmt werden. Die Differenz zur aktuell belegten Anzahl Datenblöcke (P_{USED}) entspricht der Anzahl frei werdender

Blöcke. Daraus kann mit *Gleichung 7.15* die Anzahl der zu verschiebenden Datensätze geschätzt werden.

$$T_{MOVE} = \left\lceil \left(P_{USED} - P_{IDEAL}\right) \cdot \frac{P_{ANZT}}{P_{USED}} \right\rceil$$ Gleichung 7.15

Durch die Multiplikation der Zahl zu verschiebender Sätze mit dem bei Lösch- und Einfügeoperationen anfallenden Aufwand (vgl. *Abschnitt 5.4.2)* können die Kosten für die Reorganisationsdurchführung berechnet werden (*Gleichung 7.16*).

$$C_{ReorgInPlaceOra} = \left(C_{Insert} + C_{Delete}\right) \cdot T_{MOVE}$$ Gleichung 7.16

Diese einfache Vorgehensweise lässt allerdings keine weitere Differenzierung der anfallenden Kosten zu. Es können keine Rückschlüsse darauf gezogen werden, bei welchen im Rahmen der Reorganisation durchzuführenden Operationen evtl. besonders hohe Kosten entstehen (bspw. bei der Pflege der Indexeinträge – vgl. auch *Abschnitt 7.3*). Solche Rückschlüsse können aber u.a. von Interesse sein, wenn im konkreten Fall beurteilt werden soll, welche Methode zur Reorganisation einer Tabelle besser geeignet ist. In den folgenden Ausführungen werden die bei In-Place-Reorganisationen nach der in *Abschnitt 7.1.3* beschriebenen Vorgehensweise anfallenden Kosten differenzierter betrachtet. Dabei wird auch die Beseitigung migrierter Tupel mit berücksichtigt.

In-Place-Reorganisationen können nicht in einzelne Phasen unterteilet werden. Trotzdem können auch hier die Kostenblöcke gebildet werden. Dies sind Kosten

- für das Lesen und Schreiben der geänderten Blöcke im Rahmen des Verschiebens von Datensätzen zur Beseitigung von eingestreutem Freiplatz,
- für die Aktualisierung der Indexeinträge,
- für das Verfolgen von Auslagerungszeigern und das Lesen der jeweiligen Sätze im Rahmen der Beseitigung von Auslagerungszeigern aus Datenbereichen,
- für das Schreiben von Log-Einträgen und
- für das Backup der durch die Reorganisation verursachten zusätzlichen Log-Einträge.

Die Kostenschätzungen basieren auf der Annahme, dass in der zu reorganisierenden Tabelle vorhandene Freispeicherfragmente und migrierte Tupel gleichmäßig verteilt vorkommen. Wenn jetzt (wie in *Abschnitt 7.1.3* beschrieben) ein Datenblock nur geschrieben wird, wenn er bis zum definierten Füllungsgrad aufgefüllt wurde, müssen zum Verschieben der Datensätze alle betroffenen Datenblöcke genau einmal gelesen und P_{IDEAL} Blöcke geschrieben werden (*Gleichung 7.17*). Zur Aktualisierung der Freispeicherinformationen ist je bearbeitetem Datenblock mindestens ein Zugriff notwendig. Wegen des angenommenen sehr hohen Pufferungsgrads werden die durch die Aktualisierung der Freispeicherinformationen verursachten tatsächlichen I/O-Kosten aber als eher gering angenommen und hier vernachlässigt.

$$C_{VerschiebeDaten} = \underbrace{P_{USED} \cdot \left(1 - P_{BR}\right)}_{Leseaufwand} + \underbrace{P_{IDEAL}}_{Schreibaufwand}$$ Gleichung 7.17

Wenn ein Datensatz im Rahmen der Reorganisation verschoben wird, müssen auch die den Satz referenzierenden Verweise in allen Indexen aktualisiert werden. Dazu müssen in den betroffenen Indexen die entsprechenden Einträge gesucht und aktualisiert werden. Da die Datensätze i.d.R. nicht nach dem Ordnungskriterium der Indexe sortiert vorliegen, wird hier angenommen, dass die Blattknoten (Blöcke) nach jeder Änderung geschrieben werden. Die Anzahl der Schreiboperationen wird aber meist auf Grund der Einflüsse der Datenbankpufferung unter dem hier prognostizierten Wert liegen. Wenn konkrete Werte für den Pufferungsgrad einzelner Datenbankobjekte bei Schreiboperationen vorliegen, können diese noch berücksichtigt werden oder es fließen, abhängig von vorliegenden Erfahrungswerten, andere Annahmen in die Kostenschätzungen ein. Mit der in *Abschnitt 5.4.4* vorgestellten Methode können Pufferungsgrade bezüglich Schreiboperationen leider nicht ermittelt werden. Aus dem Verhältnis von im Puffer befindlichen Blöcken zur Gesamtzahl Blöcke, die ein Datenbankobjekt belegt, lassen sich keine Annahmen über die Wahrscheinlichkeit ableiten, mit der nach einer Änderung an einem Block auch tatsächlich eine Schreiboperation ausgeführt wird.

Der für die Pflege von Indexen während des Verschiebens von Datensätzen anfallende Aufwand kann mit *Gleichung 7.18* ermittelt werden. Dabei fließt in den zweiten Faktor die Zahl der in den aufzufüllenden Blöcken enthaltenden migrierten Tupel mit ein. Deren Beseitigung erfordert jeweils die Aktualisierung von Indexeinträgen. Werden migrierte Tupel nicht beseitigt, so muss der Aufwand zur Aktualisierung der Indexe lediglich mit der Anzahl zu verschiebender Sätze (T_{MOVE}) multipliziert werden.

$$C_{Pflege_{Indexe}} = \left\lceil \underbrace{\sum_{\forall Indexe} \left(\underbrace{\sum_{i=1}^{I_{LEV}} (1 - I_{BR_i})}_{Durchmustern\ des\ Index} + \underbrace{1}_{Schreibaufwand} \right)}_{Aufwand\ je\ Satz} \cdot \underbrace{\left(T_{MOVE} + \left\lceil \underbrace{\frac{P_{OVFL}}{P_{USED}} \cdot P_{IDEAL}}_{migr.Tupel\ in\ vorderen\ Blöcken} \right\rceil \right)}_{Anzahl\ Sätze} \right\rceil$$

Gleichung 7.18

Nach den Annahmen in *Abschnitt 3.3.2* werden die zu Indexeinträgen gehörenden Verweise auf Datensätze in den Blattknoten der Indexbäume gespeichert. Zur Vereinfachung wird in Gleichung 7.18 angenommen, dass die zu einem Schlüsselwert gehörenden Verweise nicht auf mehrere Blätter verteilt vorliegen. Soll dieser Umstand berücksichtigt werden, so muss die Berechnung der Kosten für das Durchmustern eines Index wie in *Gleichung C.3* erweitert werden.

Zum Löschen von Auslagerungszeigern beim Verschieben von Sätzen muss jeweils noch ein zusätzlicher Zugriff auf die Datenblöcke erfolgen, die die Zeiger auf die ausgelagerten Sätze enthalten. Sollen Auslagerungszeiger aus den aufzufüllenden Blöcken entfernt werden, so muss der Block gelesen werden, der den ausgelagerten Satz enthält, um die zur Pflege der Indexe benötigten Attributwerte zu ermitteln. Sind über der zu reorganisierenden Tabelle keine Indexe definiert (was wahrscheinlich eher selten der Fall sein wird), so kann der mit einer der nächsten beiden

Gleichungen errechnete Aufwand entfallen, weil Auslagerungszeiger dann einfach gelöscht werden können, wenn auf Blöcke zum Auffüllen bzw. Leeren zugegriffen wird.

Beim Vorhandensein von Indexen muss also je migriertem Tupel ein weiterer Zugriff auf einen Datenblock erfolgen. Hier wird vereinfachend angenommen, dass Sätze nicht auf mehrere Blöcke verteilt abgespeichert werden. DBMS-Produkte lassen teilweise eine Verteilung auf mehrere Datenblöcke ohnehin nicht zu. Der zusätzliche Aufwand zum Lesen der Datensätze bzw. der Auslagerungszeiger kann mit *Gleichung 7.19* berechnet werden.

$$C_{Read_{migTupel}} = P_{OVFL} \cdot (1 - P_{BR}) \qquad \text{Gleichung 7.19}$$

Mit *Gleichung 7.20* kann der zusätzliche Aufwand abgeschätzt werden, der für das Lesen der Auslagerungszeiger auf zu verschiebende Sätze anfällt. Der Wert ist dann von Interesse, wenn eine Beseitigung migrierter Tupel nur im Rahmen des Verschiebens von Datensätzen vorgenommen wird.

$$C_{Read_{ForwardPointer}} = \left\lceil \underbrace{\frac{P_{OVFL}}{P_{USED}} \cdot (P_{USED} - P_{IDEAL})}_{ausgel.\,Sätze in\,zu\,leerenden\,Blöcken} \right\rceil \cdot (1 - P_{BR}) \qquad \text{Gleichung 7.20}$$

Zur Ermittlung des für das Schreiben der Log-Einträge anfallenden Aufwands fließen die Annahmen aus *Abschnitt 5.4.5* ein. Hierbei sind die Längen der Log-Einträge für das Löschen und Einfügen von Datensätzen ($LOG_{DeleteDaten}$ und $LOG_{InsertDaten}$) interessant, weil das Verschieben über Löschen und Einfügen realisiert wird. Zusätzlich werden Log-Sätze für die Aktualisierung von Indexeinträgen geschrieben. Da hier lediglich die Verweise auf die jeweiligen Datensätze geändert werden, steht für deren Länge die Größe $LOG_{UpdateIndex}$.

Für jeden verschobenen Datensatz wird ein Log-Eintrag für das Löschen und ein Log-Eintrag für das Einfügen geschrieben. Hinzu kommt der Eintrag für die Aktualisierung der Indexe. Das Löschen eines Auslagerungszeigers erfolgt wie das Löschen eines Datensatzes. Deshalb wird für jeden beseitigten Auslagerungszeiger ebenfalls ein Log-Eintrag geschrieben.

Bezüglich des Schreibens der Log-Einträge wird angenommen, dass dies zunächst in einen Blockpuffer (Log-Buffer) erfolgt, der auf den entsprechenden Datenträger geschrieben wird, nachdem er gefüllt ist. Dieser Umstand wird bereits bei den Längenangaben der Log-Einträge berücksichtigt (vgl. *Abschnitt 5.4.5*). Der Aufwand zum Schreiben der Log-Einträge kann mit *Gleichung 7.21* abgeschätzt werden. Dabei wird auch die Beseitigung von migrierten Tupeln aus den vorderen Datenblöcken im zweiten und dritten Summanden berücksichtigt.

$$C_{LoggingIP} = \left\lceil \begin{array}{l} \underbrace{T_{MOVE} \cdot \left(LOG_{DeleteDaten} + LOG_{InsertDaten}\right)}_{Verschieben\ der\ Datensätze} + \\ + \underbrace{\sum_{\forall Indexe} LOG_{UpdateIndex} \cdot \left(T_{MOVE} + \left\lceil \underbrace{\frac{P_{OVFL}}{P_{USED}} \cdot P_{IDEAL}}_{Auslagerungszeiger\ in\ vorderen\ Blöcken} \right\rceil \right)}_{Aktualisierung\ der\ Indexeinträge} + \\ + \underbrace{P_{OVFL} \cdot LOG_{DeleteDaten}}_{Beseitigen\ Auslagerungszeiger} \end{array} \right\rceil$$

Gleichung 7.21

Werden migrierte Tupel nur im Rahmen des Verschiebens von Datensätzen beseitigt, so kann der Aufwand für das Schreiben der Log-Einträge mit *Gleichung 7.22* berechnet werden.

$$C_{LoggingIP} = \left\lceil \begin{array}{l} \underbrace{T_{MOVE} \cdot \left(LOG_{DeleteDaten} + LOG_{InsertDaten}\right)}_{Verschieben\ der\ Datensätze} + \underbrace{\sum_{\forall Indexe} LOG_{UpdateIndex} \cdot T_{MOVE}}_{Aktualisierung\ der\ Indexeinträge} + \\ + \underbrace{\left\lceil \underbrace{\frac{P_{OVFL}}{P_{USED}} \cdot \left(P_{USED} - P_{IDEAL}\right)}_{ausgel.\ Sätze\ in\ zu\ leerenden\ Blöcken} \right\rceil \cdot LOG_{DeleteDaten}}_{Beseitigen\ Auslagerungszeiger} \end{array} \right\rceil$$

Gleichung 7.22

Da die Änderungen an Daten und Indexen im Rahmen des normalen Transaktions-Logging erfasst werden, ist die Erstellung eines Backups zur Sicherung der Wiederherstellbarkeit nach In-Place-Reorganisationen nicht zwingend erforderlich. Somit fallen hierfür keine Kosten an. Lediglich für die Sicherung der zusätzlichen Log-Einträge im Rahmen eines Backups fallen Kosten an, die der Reorganisation zuzuordnen sind. Diese werden in *Gleichung 7.23* erfasst.

$$C_{BackupLog} = \frac{C_{LoggingIP} \cdot DB_{PS}}{BS_{PS}}$$

Gleichung 7.23

Die Gesamtkosten für die Durchführung einer In-Place-Reorganisation können mit *Gleichung 7.24* ermittelt werden.

$$C_{ReorgInPlace} = C_{VerschiebeDaten} + C_{PflegeIndexe} + C_{ReadmigTupel} + C_{LoggingIP} + C_{BackupLog}$$

Gleichung 7.24

Werden migrierte Tupel nur im Rahmen des Verschiebens von Datensätzen beseitigt, so muss die Größe $C_{ReadmigTupel}$ durch $C_{ForwardPointer}$ ersetzt werden.

7.3 Beispielrechnungen/Vergleich der Methoden

Als Beispiel für die Ermittlung von Reorganisationskosten und für einen Vergleich der drei beschriebenen Reorganisationsmethoden werden in diesem Abschnitt die I/O-Kosten betrachtet, die für die Reorganisation der Tabelle `TEILNEHMER` des

Beispiels aus *Abschnitt 6.3.2.6* anfallen würden. In *Anhang F* finden sich weitere Informationen zu den verwendeten Ausgangsgrößen. Diese Werte bilden die Basis der Beispielrechnungen. Neben dem bereits in Abschnitt 6.3.2.6 verwendeten Index über dem Primärschlüsselattribut `MSISDN` wurde die Tabelle während der zur Überprüfung der errechneten Werte aufgenommenen Messreihen zusätzlich über die Attribute `Name` und `Ort` indexiert. In den Beispielrechnungen wird ebenfalls von drei Indexen ausgegangen.

Im Rahmen der Überprüfung der in diesem Abschnitt angestellten Berechnungen wurden an der Puffer- und an der Dateischnittstelle (unter Nutzung der von Oracle über die Sichten `V$SESS_IO` und `V$FILESTAT` zur Verfügung gestellten Informationen) jeweils die Zahl gelesener bzw. geänderter/geschriebener Blöcke ermittelt. Die Ergebnisse der Messreihen werden ebenfalls in Anhang F präsentiert. Während der Messungen ermittelte Werte bezüglich der Einflüsse von Datenbankpufferung und asynchroner Ausführung von Schreiboperationen fließen in die Beispielrechnungen dieses Abschnitts ein. Die errechneten Aufwandswerte stellen den jeweils zu erwartenden physischen I/O-Aufwand dar. Weiterhin wird angenommen, dass der Datenbereich der Tabelle vor der Reorganisation **8050** Blöcke belegt. Das entspricht einem Freiplatzanteil ca. **50%** (vgl. auch *Tabelle F.2*). Die Kosten zum Erzeugen bzw. Aktualisieren von Katalogeinträgen (bspw. beim Löschen oder Anlegen von Datenbankobjekten) werden hier vernachlässigt, da die aufgenommenen Messreihen auch gezeigt haben, dass die notwendigen Zugriffe nahezu ausschließlich im Datenbankpuffer erfolgen.

Für die durchgeführten Messreihen wurden die Beispieldaten aus Abschnitt 6.3.2.6 verwendet. Die relativ geringe Zahl an Blöcken ermöglicht eine einfachere und schnellere Durchführung der Messreihen. Die Ergebnisse lassen sich aber durchaus auch auf größere Datenbankobjekte übertragen. Lediglich für die Größen, mit denen die Einflüsse von Datenbankpufferung und asynchroner Ausführung von Schreiboperationen berücksichtigt werden, würden sich andere Werte ergeben. Diese sind aber in jedem Fall von der konkreten Systemumgebung abhängig und müssen dort ermittelt werden.

Die Kosten, die für eine Reorganisation nach der *datenbanksystembasierten Methode* anfallen würden, zeigt *Abbildung 7.5*. Die unterschiedlichen Kostenwerte für Export und Import der Daten ergeben sich aus der reduzierten Anzahl Blöcke, die das Datensegment der Tabelle nach der Reorganisation durch die Beseitigung von vorher vorhandenem eingestreuten Freiplatz belegt. Ein erheblicher Aufwand (ca. **31%** des Gesamtaufwands von **47632** gelesenen bzw. geschriebenen Blöcken) fällt für den Neuaufbau der drei Indexe an. Dies liegt daran, dass die Indexe einzeln mit normalen `CREATE INDEX`-Anweisungen aufgebaut werden und somit neben den Operationen zum Schreiben der Indexblöcke alle Datenblöcke der Tabelle je Index einmal gelesen werden müssen. Es dominieren hier somit die Leseoperationen.

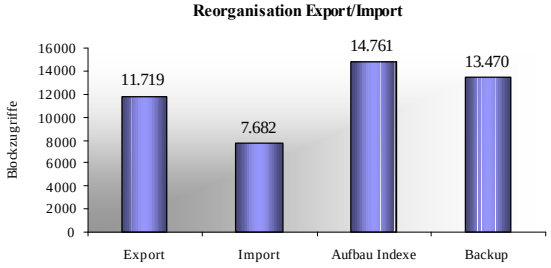


Abbildung 7.5: Kosten für eine Reorganisation über Export und Import

Auch für die Erstellung des abschließenden Backups fallen deutlich merkbare Kosten an, weil jeder Daten- bzw. Indexblock einmal *gelesen und geschrieben* werden muss. Die Anteile der einzelnen Schritte am Gesamtaufwand zeigt *Abbildung 7.6*.

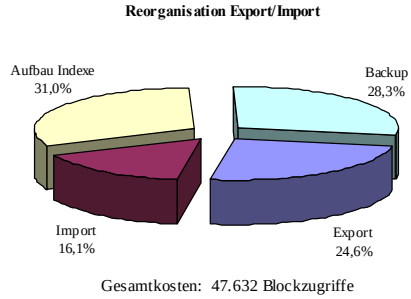


Abbildung 7.6: Anteile der einzelnen Schritte an den Gesamtkosten bei einer Reorganisation über Export und Import

Die Kostenwerte in Abbildung 7.5 stehen für einzelne gelesene bzw. geschriebene Blöcke. Die Zahl der bei der datenbanksystembasierten Methode anfallenden I/O-Operationen kann deutlich reduziert werden, wenn bei sequenziell arbeitenden Lese- bzw. Schreiboperationen Mechanismen zum vorausschauenden Lesen (Prefetching) und (zusammenhängenden) sequenziellen Schreiben genutzt werden. Für die in *Abbildung 7.7* dargestellten Aufwandswerte wurde angenommen, dass mit einer I/O-Operation zehn Blöcke gelesen bzw. geschrieben werden können. Die dann anfallenden Gesamtkosten würden im Beispiel nur noch **7216** statt **47632** Lese- bzw. Schreiboperationen (das sind ca. **15%**) betragen. Allerdings ist zu beachten, dass die errechneten Aufwandswerte nur unter günstigen Voraussetzungen erreicht werden können. Eventuell auftretende Fragmentierungen von Daten- und Indexbereichen können dazu führen, dass nicht mit jeder I/O-Operation tatsächlich auch zehn Blöcke (wie angenommen) bearbeitet werden können.

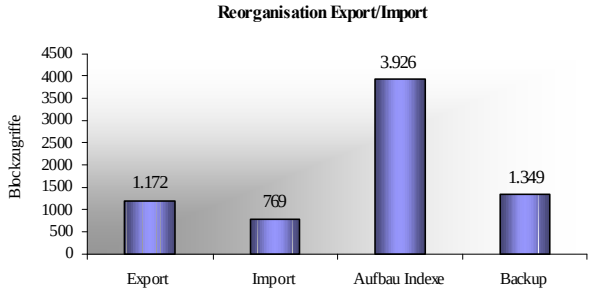


Abbildung 7.7: Kosten für eine Reorganisation über Export und Import bei der Nutzung von Mechanismen zum vorausschauenden Lesen und sequenziellen Schreiben

Die veränderten Kostenanteile der einzelnen Reorganisationsschritte zeigt *Abbildung 7.8*. Der Aufwandsanteil für den Aufbau der Indexe ist deutlich gestiegen. Dies liegt an dem jetzt relativ hohen Aufwand für das blockweise Schreiben der Indexknoten, weil angenommen wird, dass diese Schreiboperationen zwar durchaus asynchron erfolgen, durch die Baumstruktur aber kaum sequenziellen Schreiboperationen großer Bereiche auftreten dürften (vgl. *Abschnitt 7.2.1*).

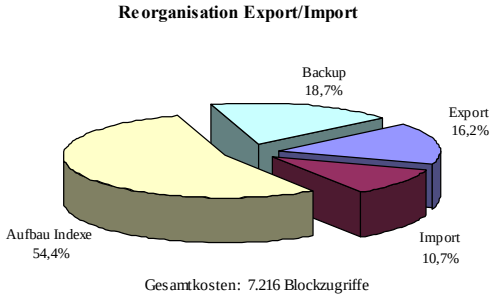


Abbildung 7.8: Anteile der einzelnen Schritte an den Gesamtkosten bei einer Reorganisation über Export und Import bei der Nutzung von Mechanismen zum vorausschauenden Lesen und sequenziellen Schreiben

Die Kosten, die bei blockweisem Lesen und Schreiben bei einer *Online-Reorganisation* der Beispieltabelle *in eine Kopie* anfallen würden, zeigt *Abbildung 7.9*.

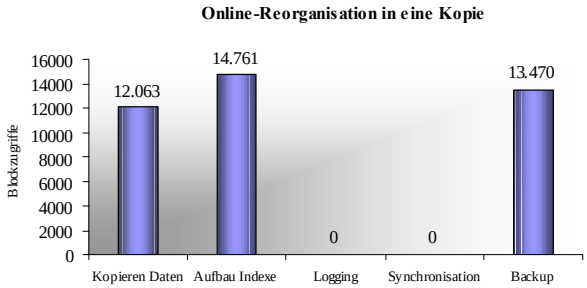


Abbildung 7.9: Kosten für eine Online-Reorganisation der Beispieltabelle in eine Kopie

Relevante Kostenblöcke stellen hier zunächst das Kopieren der Daten, der Neuaufbau der Indexe und die abschließende Erstellung des Backups dar. Eventuell während der Reorganisation an den Daten der Originaltabelle vorgenommene Änderungen werden noch nicht berücksichtigt. Dies ermöglicht auch einen Vergleich mit den bei der datenbanksystembasierten Methode anfallenden Kosten. Der Aufwand für die Erstellung des Backups und zum Aufbau der Indexe entspricht dem der datenbanksystembasierten Methode (vgl. Abbildung 7.5). Die Gesamtkosten für eine Online-Reorganisation in eine Kopie sind geringer als bei der datenbanksystembasierten Methode, weil die Kosten für das Schreiben und Lesen der Exportdatei entfallen. Die Kosten für den Export der Daten in Abbildung 7.5 sind etwas niedriger als die Kosten für das Kopieren der Daten in Abbildung 7.9. Dies liegt daran, dass angenommen wird, dass die ausgelagerten Datensätze bei der datenbanksystembasierten Methode sequenziell in die Exportdatei geschrieben werden und dass bspw. kein Speicherplatz zur Verwaltung der einzelnen Blöcke der Exportdatei (z.B. für Block-Header o.ä.) benötigt wird. Damit belegt die Exportdatei weniger Datenblöcke (die geschrieben werden müssen) als die Interim-Tabelle.

Durch die Anwendung von Mechanismen zum vorausschauenden Lesen und sequenziellen Schreiben kann der Aufwand auch bei Reorganisationen in eine Kopie auf ca. **16%** (von **40294** auf **6482** Lese- und Schreiboperationen) verringert werden, wenn wieder angenommen wird, dass mit einer I/O-Operation (bspw. beim Kopieren der Daten) zehn Blöcke gelesen bzw. geschrieben werden können (vgl. auch *Abbildung 7.10*).

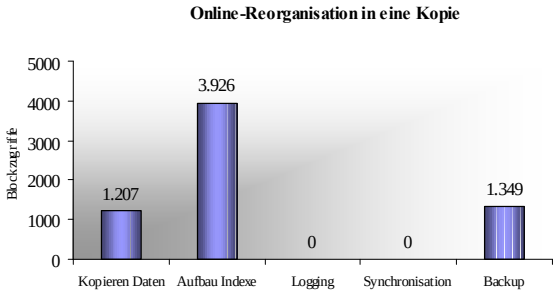


Abbildung 7.10: Kosten für eine Online-Reorganisation der Beispieltabelle in eine Kopie bei der Nutzung von Mechanismen zum vorausschauenden Lesen und sequenziellen Schreiben

Die Verfügbarkeit der Daten für Benutzerzugriffe während der Reorganisation ist eine wesentliche Eigenschaft von Online-Reorganisationsmethoden. Anhand einer weiteren Beispielrechnung wird der Einfluss von während der Reorganisation vorgenommenen Änderungen an den Daten der Originaltabelle gezeigt. Dabei wird in den Berechnungen (wo möglich) wieder Nutzung von Mechanismen zum vorausschauenden Lesen und sequenziellen Schreiben unterstellt. Bezüglich der Einflüsse von Datenbankpufferung und asynchroner Ausführung von Schreiboperationen werden die in Anhang F getroffenen Annahmen verwendet. *Abbildung 7.11* zeigt die Veränderungen der Kosten, wenn während der Reorganisation **1040** Änderungsoperationen (bezogen auf die Tupelzahl der Beispieltabelle vor der Reorganisation ist das ca. **1%**) an den Daten der

Originaltabelle vorgenommen werden. Dabei werden **520** `INSERT`-, **340** `UPDATE`- und **180** `DELETE`-Anweisungen angenommen. Im Rahmen der `UPDATE`-Anweisungen werden keine Änderungen an indexierten Attributen vorgenommen.

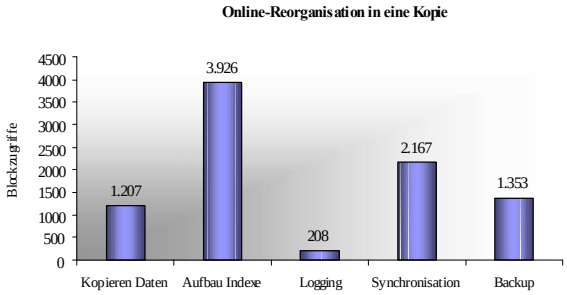


Abbildung 7.11: Kosten für eine Online-Reorganisation der Beispieltabelle in eine Kopie bei einer Änderungsrate von 1% während der Reorganisation

Ein Vergleich der Kostenwerte aus Abbildung 7.10 und Abbildung 7.11 zeigt einen Anstieg der Gesamtkosten um ca. **36,7%.** Diese trotz der geringen Anzahl vorgenommener Änderungen doch relativ starken Auswirkungen ergeben sich daraus, dass weder beim Aufzeichnen der Änderungen in der Log-Tabelle, noch beim Synchronisieren der beiden Tabellen Mechanismen zum vorausschauenden Lesen und sequenziellen Schreiben angewendet werden können. Die Änderungen werden für einzelne Tupel aufgezeichnet. Die Synchronisation erfolgt über die jeweiligen Primärschlüsselwerte. Sequenzielle Lese- oder Schreiboperationen finden nicht statt. Die Anteile der einzelnen Kostenblöcke am Gesamtaufwand von **8861** Lese- bzw. Schreiboperationen im Beispiel zeigt *Abbildung 7.12*.

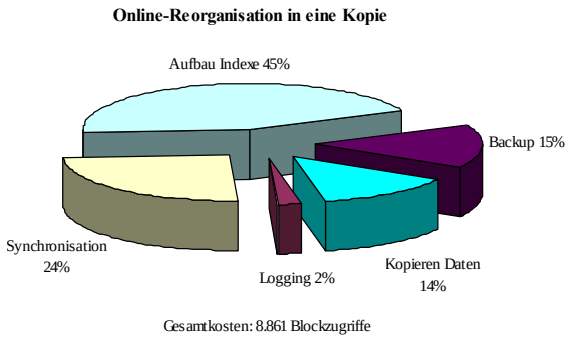


Abbildung 7.12: Anteile der einzelnen Kostenblöcke bei einer Online-Reorganisation der Beispieltabelle in eine Kopie bei einer Änderungsrate von 1%

Ein großer Teil des Aufwands für die Synchronisation entfällt auf das Erzeugen bzw. die Pflege von Indexeinträgen. Je Index muss auf die betroffenen Indexeinträge zugegriffen und diese müssen geändert werden. Zusätzlich fallen I/O-Operationen zum Schreiben der geänderten Blätter und der Aufwand für das Schreiben der entsprechenden Einträge in das Transaktionsprotokoll (Log) des DBMS an.

Die bei einer *In-Place-Reorganisation* der Beispieltabelle anfallenden Kosten zeigt *Abbildung 7.13.* Auch hier fließen die in Anhang F getroffenen Annahmen bezüglich

der Einflüsse von Datenbankpufferung und asynchroner Ausführung von Schreiboperationen ein.

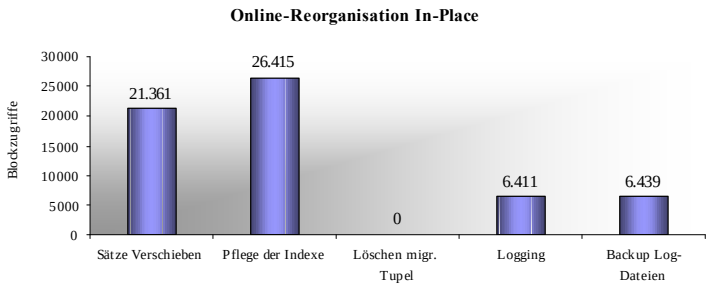


Abbildung 7.13: Kosten für eine In-Place-Reorganisation der Beispieltabelle

Als Basis wird für die Berechnung wieder ein Freiplatzanteil von ca. **50%** angenommen. Dies entspricht ca. **52100** zu verschiebenden Sätzen. Die Grundlage für die Beispielrechnung bildet die von Oracle implementierte Vorgehensweise, die Verschiebefunktionalität über Löschen und Einfügen von Sätzen zu realisieren. Der von der Reorganisation verursachte Aufwand ist damit vorrangig von der Zahl der zu verschiebenden Sätze abhängig. Um die Zuordnung des Aufwands zu den einzelnen Kostenblöcken zu ermöglichen, wurden die Kosten für das Löschen bzw. Einfügen von Datensätzen (vgl. Gleichung 7.16) in den Beispielrechnungen weiter aufgeteilt in

- Kosten, die durch die notwendigen Änderungen an den Datenbereichen verursacht werden und
- Kosten, die bei der Pflege der Indexe entstehen.

Bei der Betrachtung der Werte in Abbildung 7.13 fällt zunächst auf, dass die Gesamtkosten im Vergleich zu den beiden anderen Methoden mit **62411** I/O-Operationen deutlich höher sind. Dies gilt zumindest, wenn bei den beiden anderen Methoden die Ausnutzung von Mechanismen zum vorausschauenden Lesen und sequenziellen Schreiben unterstellt wird. Da In-Place-Reorganisationen über kleine Einzeloperationen realisiert werden, können solche Mechanismen hier nicht zur Anwendung kommen.

Im betrachteten Beispielfall werden über **43%** des Aufwands durch die notwendige Pflege der Einträge der drei Indexe verursacht. Besonders wenn eine große Zahl an Datensätzen zu bewegen ist, kann der Reorganisationsaufwand gesenkt werden, indem Indexe vor der Reorganisation gelöscht und danach wieder neu aufgebaut werden. Allerdings stehen die Indexe während dieser Zeit dann auch nicht für den parallel laufenden normalen Datenbankbetrieb zur Verfügung, was u.U. zu erheblichen Einschränkungen führen kann. Eine Verteilung des anfallenden Aufwands auf die einzelnen Kostenblöcke zeigt *Abbildung 7.14*.

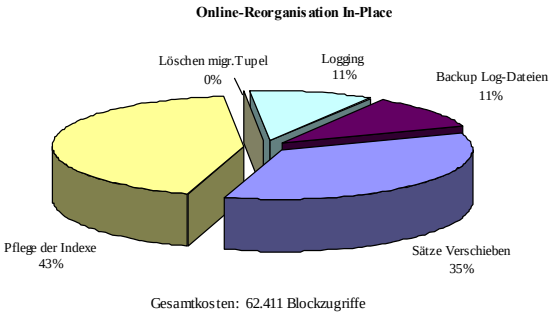


Abbildung 7.14: Verteilung der Kosten bei einer In-Place-Reorganisation der Beispieltabelle

Die Anzahl der während der Reorganisation von Transaktionen des normalen Datenbankbetriebs an den Daten der Tabelle vorgenommenen Änderungen hat hier, von eventuellen Einflüssen auf die Pufferungsgrade von Index- und Datenblöcken abgesehen, keinen direkten Einfluss auf die während der Reorganisation anfallenden I/O-Kosten. Eine spürbare Beeinflussung der Dauer der Reorganisation kann sich aber durch (hier nicht berücksichtigte) *Sperrkonflikte* zwischen Reorganisationsprozess und normalem Datenbankbetrieb und die damit verbundene Verzögerung der Ausführung von Operationen ergeben.

Die bei In-Place-Reorganisationen anfallenden I/O-Kosten können aber durch den zu Beginn der Reorganisation vorliegenden Anteil migrierter Tupel beeinflusst werden. Dies gilt besonders, wenn diese im Rahmen der Reorganisation auch aus den aufzufüllenden Datenblöcken beseitigt werden sollen.

Eine vollständige Beseitigung migrierter Tupel im Rahmen von In-Place-Reorganisationen wird von Oracle nach den Ergebnissen unserer Untersuchungen nicht durchgeführt. Es handelt sich hier also um eine reine Beispielrechnung. Diese soll die Auswirkungen demonstrieren, die eine solche angestrebte vollständige Beseitigung migrierter Tupel auf die gesamten Reorganisationskosten hätte.

Bei den bisherigen Betrachtungen wurde die Anzahl migrierter Tupel mit **0** angenommen. Liegt deren Anteil zu Beginn der Reorganisation bei ca. **20%**, ergeben sich die anfallenden Kosten wie in *Abbildung 7.15* gezeigt.

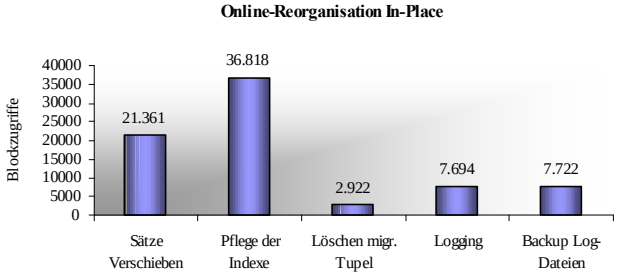


Abbildung 7.15: Kosten für eine In-Place-Reorganisation der Beispieltabelle mit einem Anteil von 20% migrierten Tupeln zu Beginn der Reorganisation

Der absolute Aufwand für die Pflege der Indexe steigt spürbar. Dies ist darauf zurückzuführen, dass die Umwandlung eines ausgelagerten Datensatzes in einen „normalen“ Datensatz, inklusive der Aktualisierung der Indexeinträge, nahezu die gleiche Anzahl Blockzugriffe verursacht, wie das Verschieben eines Datensatzes im Rahmen der Reorganisation. Hinzu kommen die Kosten für das Löschen der mit migrierten Tupeln verbundenen zusätzlichen Zeiger aus den Datenblöcken. Die veränderten Kostenanteile zeigt *Abbildung 7.16*.

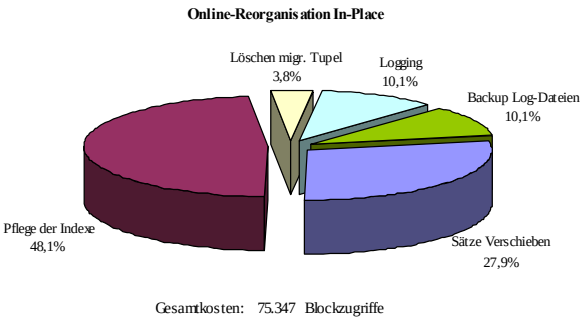


Abbildung 7.16: Verteilung der Kosten bei einer In-Place-Reorganisation der Beispieltabelle mit einem Anteil von 20% migrierten Tupeln zu Beginn der Reorganisation

Generell wird an den Kostenbetrachtungen für alle drei Methoden deutlich, dass vorhandene Indexe einen erheblichen Aufwand für das Neuanlegen bzw. für das Pflegen der Indexeinträge verursachen. Das heißt, dass auch aus Sicht der Durchführung von Datenbankreorganisationen auf wenig frequentierte Indexe möglichst verzichtet werden sollte.

Die Ergebnisse der angestellten Beispielrechnungen konnten im Rahmen von Messreihen (vgl. auch *Anhang F*) in großen Teilen bestätigt werden. Allerdings ist die Genauigkeit der durchgeführten Schätzungen teilweise geringer als bei den Aufwandsschätzungen zur Workload-Abarbeitung und zur Ermittlung des Nutzens von Datenbankreorganisationen. Eine Ursache hierfür sind sicherlich die getroffenen Annahmen bezüglich des Einflusses von Datenbankpufferung und asynchroner Ausführung von Schreiboperationen. Weiterhin sind uns nicht alle Details der Implementierung der Reorganisationsmethoden durch die DBMS-Hersteller (hier speziell Oracle) bekannt, was ebenfalls zu Ungenauigkeiten führt. Zur Verbesserung der Genauigkeit der Schätzungen wären ggf. noch weitere Untersuchungen notwendig.

8 Maximierung des Nutzens von Datenbankreorganisationen

In diesem Kapitel wird ein Ansatz zur Unterstützung von Datenbankadministratoren bei der Auswahl der Reorganisationskandidaten skizziert, der eingebettet ist in die Gesamtuntersuchungen der vorliegenden Arbeit und im Rahmen von [Wie05] näher erarbeitet wurde.

Üblicherweise stehen bei einer Datenbankreorganisation mehrere (viele) Datenbankobjekte (Reorganisationskandidaten) zur Auswahl. Zur Reorganisationsdurchführung stehen (wenn überhaupt) oftmals nur bestimmte Zeitfenster und Ressourcen zur Verfügung. Innerhalb der Zeitfenster muss die Reorganisation jeweils abgeschlossen sein. Damit ist der betreibbare Aufwand begrenzt. Dies gilt i.d.R. auch, wenn die Reorganisation parallel zum laufenden Betrieb durchgeführt wird. Hier behindert die Belegung von Ressourcen den laufenden Datenbankbetrieb. Ein möglicher Ansatz zur Unterstützung von Datenbankadministratoren bei der Auswahl von Reorganisationskandidaten ist, den mit dem betreibbaren Aufwand erreichbaren Nutzen zu maximieren. Das Ziel der Maximierung des Nutzens soll mit Hilfe eines Optimierungsverfahrens erreicht werden. In *Abschnitt 8.1* werden einige grundlegende Betrachtungen zum in dieser Arbeit zur Anwendung vorgeschlagenen Verfahren angestellt. Das Optimierungsproblem wird in *Abschnitt 8.2* definiert. Die Beschreibung des zur Lösung eingesetzten Branch-and-Bound-Verfahrens erfolgt in *Abschnitt 8.3*.

Ein Ansatz zur Unterstützung von Datenbankadministratoren bei der Auswahl von Reorganisationskandidaten wird in dem im Produkt *Quest Central for Databases 4.0* enthaltenen `Reorg Xpert` von Quest Software verfolgt [Hei04]. Von dem Produkt wird aus statischen Kennzahlen über die Speicherplatzauslastung und die Anzahl migrierter Tupel ein Reorganisationsbedarf errechnet. Bei der Zusammenstellung der Reorganisationskandidaten werden diese von der Reihenfolge her nach dem Kosten-Nutzen-Verhältnis angeordnet, also so, dass die Objekte bevorzugt werden, bei deren Reorganisation möglichst viele Degenerierungen mit möglichst wenig Datenbewegungen beseitigt werden. Der DBA kann dann festlegen, bei welchem Nutzen (Anteil an beseitigten Degenerierungen) die Reorganisation beendet werden soll. Allerdings ermittelt der `Reorg Xpert` in der betrachteten Version den Reorganisationsnutzen ohne Workload-Berücksichtigung. Damit kann der Nutzen bezüglich der Systemleistung nicht quantifiziert werden. Wie die Werte für den mit einer Reorganisation erreichbaren Nutzen vom `Reorg Xpert` ermittelt werden, ist zudem von außen nicht vollständig nachvollziehbar.

8.1 Grundlegende Betrachtungen

Mit dem in der vorliegenden Arbeit betrachteten Ansatz sollen aus einer Menge von Reorganisationskandidaten diejenigen ausgewählt werden, mit deren Reorganisation sich der größtmögliche Nutzen erzielen lässt. Dabei wird angenommen, dass der betreibbare Aufwand begrenzt ist. Ursache hierfür können bspw. begrenzte Zeiträume sein, in denen Wartungsoperationen durchgeführt werden können. Hier kann bspw.

die Verwendung von Benchmarks zur Ermittlung von Basiswerten (z.B. die Anzahl I/O-Operationen, die in einem bestimmten Zeitraum ausgeführt werden können) nützlich sein. Aber auch wenn Wartungsoperationen online durchgeführt werden können, ist der durch die Reorganisation verursachte Aufwand ein Störfaktor bezüglich des parallel laufenden normalen Datenbankbetriebs.

Die Problemstellung entspricht einem aus der mathematischen Optimierung bekannten Rucksackproblem [ECS93]. Die einzelnen Reorganisationskandidaten stellen dabei die Gegenstände dar, die in den Rucksack gepackt werden können. Die Beschränkung des betreibbaren Aufwands entspricht der Kapazität des Rucksacks. Die genaue Definition des hier zu betrachtenden Optimierungsproblems erfolgt in *Abschnitt 8.2*.

Die zum Packen des Rucksacks verwendbaren Gegenstände sind *unteilbar* und stehen jeweils nur einmal zur Verfügung. Auf die Reorganisationsproblematik übertragen bedeutet die „Unteilbarkeit“, dass sich das Reorganisationsgranulat nach dem bei den Kosten- und Nutzenanalysen verwendeten Granulat richtet. Wurden bspw. bei einer solchen Analyse eine Tabelle und die ihr zugeordneten Indexe als Einheit betrachtet, so wird für die Optimierung angenommen, dass eine Reorganisation auch auf die Tabelle und die ihr zugeordneten Indexe angewendet wird. Sollen evtl. auch nur einzelne Indexe reorganisiert werden, so müssen die Kosten- und Nutzenanalysen von vornherein einzeln für die Indexe und (evtl.) den Datenbereich der Tabelle durchgeführt werden.

Bei der Ermittlung der *Nutzenwerte* werden die einzelnen Reorganisationskandidaten als unabhängig angesehen. Die Reorganisation eines Kandidaten zur Beseitigung von Degenerierungen hat keinen Einfluss auf den durch die Reorganisation eines anderen Kandidaten erreichbaren Nutzen. Wird eine Reorganisation zur Veränderung der physischen Repräsentation von Datenbankobjekten durchgeführt, so können (bspw. bei hierarchisch strukturierten Objekten) hier durchaus Abhängigkeiten existieren.

Die einzelnen Nutzenwerte stehen für die durch die Reorganisation erreichbare Reduzierung des I/O-Aufwands in Prozent bezogen auf die betrachtete Workload. Nutzenwerte kleiner als 0 treten i.d.R. nicht auf, da dies bedeuten würde, dass der Aufwand zur Workload-Abarbeitung durch die Reorganisation steigt. Da dies normalerweise nicht das Ziel von Datenbankreorganisationen ist, wird davon ausgegangen, dass Kandidaten, deren Reorganisation eine Aufwandserhöhung verursachen würde, vorab von den Betrachtungen ausgeschlossen werden.

Für die Optimierung wird *je Kandidat ein Kostenwert* berücksichtigt. Das heißt, dass vorab feststehen muss, mit welcher Methode (vgl. Kapitel 7) die Reorganisation eines Datenbankobjekts durchgeführt wird. Damit kann das Modell für das Optimierungsproblem einfach gehalten werden, was die Zahl der möglichen Lösungen in engeren Grenzen hält. Aus praktischer Sicht stellt diese Vereinfachung sicherlich kein größeres Problem dar.

Die Entscheidung für eine Reorganisationsmethode wird i.d.R. nicht nur auf der Basis der zur Reorganisationsdurchführung anfallenden I/O-Kosten gefällt. Verfügbarkeitsanforderungen lassen oftmals die Nutzung kostengünstig

durchführbarer Offline-Verfahren nicht zu. Auch der Umstand, ob genügend Speicherplatz zur Verfügung steht, um eine Online-Reorganisation in eine Kopie durchzuführen, kann im Optimierungsmodell der Problemstellung nicht berücksichtigt werden. Ähnlich verhält es sich mit anderen Nebenbedingungen (wie bspw. dem Vorhandensein eines definierten Primärschlüssels). Bevor also das Optimierungsverfahren angewendet werden kann, muss zunächst im konkreten Fall geprüft werden, mit welchen Methoden die einzelnen Datenbankobjekte überhaupt reorganisiert werden können. Solche Prüfungen können oftmals mit den derzeit verfügbaren Werkzeugen zur Reorganisationsdurchführung (teilweise in unterschiedlicher Tiefe) durchgeführt werden (vgl. *Abschnitt 4.4*). Für die *anwendbaren Methoden* können dann jeweils die anfallenden Kosten abgeschätzt und dem Administrator zur Entscheidung angeboten werden. Dieser kann festlegen, welche Reorganisationsmethode unter den gegebenen Umständen den Verfügbarkeitsanforderungen und Forderungen nach geringen Kosten bestmöglich entspricht. Der Kostenwert für die ausgewählte Methode fließt dann in das Optimierungsverfahren ein.

8.2 Definition des Optimierungsproblems

Ergebnis der Optimierung (dargestellt durch die Zielfunktion) soll es sein, aus einer Menge von Datenbankobjekten, die für eine Reorganisation eingeplant werden können, diejenigen auszuwählen, mit deren Reorganisation der größtmögliche Nutzen erzielt werden kann. Dabei gilt die Nebenbedingung, dass der betreibbare Aufwand begrenzt ist.

Die Menge $K=\{k_1,k_2,\ldots,k_n\}$ steht für die Menge der möglichen Reorganisationskandidaten. Die als Basis für die Durchführung der Optimierung notwendige Ermittlung der Nutzenwerte der einzelnen Reorganisationskandidaten muss dabei bezogen auf die betrachtete Gesamt-Workload (also global – vgl. auch *Abschnitt 6.3.2.2*) erfolgen. Dabei steht N_i (mit $1 \leq i \leq n$) für den durch die Reorganisation des Kandidaten k_i erreichbaren Nutzen. C_{MAX} steht für die Kosten (hier: Anzahl I/O-Operationen), die im Rahmen der Reorganisationsdurchführung maximal anfallen dürfen. Der Wert von C_{MAX} muss vorab festgelegt werden. Wird ein Kandidat k_i zur Reorganisation eingeplant, so muss der damit verbundene „Verbrauch“ an aufwendbaren Kosten berücksichtigt werden. Die Ermittlung der durch die Reorganisation eines Datenbankobjekts k_i verbrauchten Kosten C_i kann wie in *Kapitel 7* dargestellt erfolgen.

Ausgehend von diesem Modell kann das Optimierungsproblem dann wie folgt formuliert werden. *Gleichung 8.1* stellt die Zielfunktion dar, deren Ergebnis maximiert werden soll.

$$\max f(x) = \sum_{i=1}^{n} x_i \cdot N_i \quad mit\ x_i \in \{0,1\}\ und\ \ Nebenbed.\ \ \sum_{i=1}^{n} x_i \cdot C_i \leq C_{MAX} \qquad \text{Gleichung 8.1}$$

Das Ergebnis der Optimierung ist also ein Vektor $X=\{x_1,x_2,\ldots,x_n\}$, in dem angegeben ist, ob ein Kandidat in die Reorganisation einbezogen wird oder nicht. Dabei gilt:

$$x_i = \begin{cases} 1 & \textit{Kandidat wird einbezogen} \\ 0 & \textit{Kandidat wird nicht einbezogen} \end{cases} \qquad \text{mit } 1 \le i \le n$$

Die Maximierung des Zielfunktionswerts muss unter Beachtung der Nebenbedingung erfolgen, dass die maximalen Kosten nicht überschritten werden dürfen.

Die Aufgabenstellung gleicht der des Rucksackproblems, bei dem verschiedene Gegenstände k_i (Reorganisationskandidaten) betrachtet werden, die in einen Rucksack gepackt (bei uns: reorganisiert) werden sollen. Die Kapazität des Rucksacks ist allerdings begrenzt (C_{max}). Jeder Gegenstand hat ein bestimmtes Volumen (C_i), das den bei der Reorganisationsdurchführung anfallenden Kosten entspricht und einen bestimmten Nutzwert (N_i – hier: der von der Reorganisationsdurchführung zu erwartenden Nutzen). Der Rucksack soll nun so gepackt werden, dass die maximale Kapazität nicht überschritten wird, der Gesamtwert aller Gegenstände allerdings so hoch wie möglich ist. Die Gegenstände selbst können nicht geteilt und dürfen höchstens einmal eingepackt werden.

8.3 Lösungsweg

Zur Lösung der beschriebenen Problemstellung können Enumerationsverfahren angewendet werden, bei denen die zulässigen Lösungen aufgezählt und die beste Lösung, d.h. die mit dem größten Zielfunktionswert, ausgewählt wird. Zur Aufzählung möglicher Lösungen wird hier ein Branch-and-Bound-Verfahren verwendet. Dabei wird die Optimallösung systematisch durch Aufspalten des zulässigen Bereichs eines Originalproblems gesucht [EBL98]. Durch das Aufspalten ergibt sich ein Entscheidungsbaum (vgl. auch *Abbildung 8.2*). Im ungünstigsten Fall müssen alle möglichen Belegungen aufgezählt werden. Allerdings wird versucht, den zu untersuchenden Lösungsraum zu begrenzen, indem Zweige, die nicht zur optimalen Lösung führen können, erkannt und in den weiteren Betrachtungen nicht mehr berücksichtigt werden. In [SKK+97] werden für das Rucksackproblem verschiedene Strategien zum Bilden von Teilproblemen und zur Reihenfolge der Betrachtung der Teilprobleme gegenübergestellt. Durch das Einpacken eines Gegenstands bzw. das Verbieten des Einpackens werden *disjunkte Teilprobleme* erzeugt. Lösungsmöglichkeiten, bei denen nur ein geringerer Nutzen als bei der bisher gefundenen besten Lösung erreicht werden kann, werden mit Hilfe von *Auslotregeln*, die logische Tests darstellen, erkannt und zukünftig nicht mehr betrachtet.

Die Grundprinzipien des Verfahrens sind das Verzweigen (*Branching*) und Begrenzen (*Bounding*).

- Beim *Branching* wird die zu lösende Aufgabenstellung in zwei Teilprobleme aufgeteilt. Jedes der so entstandenen Teilprobleme kann wieder aufgeteilt werden. Damit ergibt sich ein Entscheidungsbaum.

- Über das *Bounding* wird eine Beschränkung des Verzweigungsprozesses auf der Grundlage von Auslotregeln erreicht. Dazu wird für die einzelnen Knoten des Entscheidungsbaums der jeweilige Zielfunktionswert errechnet. Dieser stellt eine obere Schranke für den in diesem Teilast maximal zu erreichenden Nutzen dar. Anschließend wird versucht, das Problem *auszuloten*. Gelingt dies, muss nicht weiter verzweigt werden. Dadurch wird der Baum beschränkt.

Ein Problem wird im hier betrachteten Fall als *ausgelotet* bezeichnet, wenn eine der beiden folgenden Bedingungen zutrifft.

1. Die obere Schranke für den Zielfunktionswert des Teilproblems ist nicht besser als die bisher beste bekannte zulässige Lösung. Damit kann in diesem Teilast keine bessere Lösung gefunden werden.
2. Die zulässige Lösung eines Teilproblems ist die beste bisher gefundene Lösung und wird gespeichert.

Anhand eines einfachen Beispiels soll die verwendete Vorgehensweise der klassischen Branch-and-Bound-Methodik dargestellt werden. Dabei wird so vorgegangen, dass nach dem Verzweigen zuerst die beiden entstehenden Teilprobleme gelöst und der jeweilige Zielfunktionswert bestimmt werden. Erst danach wird eventuell neu verzweigt. Da durch diese Vorgehensweise schneller Kenntnisse über Zielfunktionswerte in *beiden* Teilästen erlangt werden, können die jeweils betrachteten Teilprobleme u.U. schneller ausgelotet werden.

Die zur Darstellung des Branch-and-Bound-Verfahrens verwendeten Beispieldaten enthält *Tabelle 8.1*.

Kandidat k_i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Nutzen N_i (in %)	2,1	2,4	1,2	10,8	3,5	2,6	1,4	3,2	1,2	5,3
Kosten C_i	4	2	25	41	12	17	19	23	10	9

Tabelle 8.1: Beispieldaten für die Erläuterung des Branch-and-Bound-Verfahrens

Die Tragfähigkeit des Rucksacks (die maximal aufwendbaren Kosten) soll im Beispiel 45 Einheiten betragen.

Aus den in Tabelle 8.1 angegebenen Nutzenwerten lässt sich die in *Gleichung 8.2* angegebene *Zielfunktion* ableiten.

$$f(x) = 2{,}1k_1 + 2{,}4k_2 + 1{,}2k_3 + 10{,}8k_4 + 3{,}5k_5 + 2{,}6k_6 + 1{,}4k_7 + 3{,}2k_8 + 1{,}2k_9 + 5{,}3k_{10}$$

Gleichung 8.2

Die Problemstellung besteht nun darin, k_1 bis k_{10} so mit den Werten 1 bzw. 0 zu belegen (Kandidaten in die Reorganisation einzubeziehen oder auszuschließen), dass der Wert von $f(x)$ maximiert wird. Die maximalen Kosten von 45 Einheiten dürfen dabei nicht überschritten werden. Damit lässt sich die *Nebenbedingung* mit *Gleichung 8.3* angeben.

$$45 \geq 4k_1 + 2k_2 + 25k_3 + 41k_4 + 12k_5 + 17k_6 + 19k_7 + 23k_8 + 10k_9 + 9k_{10}$$ Gleichung 8.3

Zur Lösung wird zunächst das *Originalproblem* betrachtet. Wenn die Kosten zur Reorganisation eines bestimmten Kandidaten höher sind als die maximal aufwendbaren Gesamtkosten, wird dieser von zukünftigen Betrachtungen ausgeschlossen. Dazu wird die entsprechende Variable k_i auf den Wert 0 gesetzt und fixiert[19].

Nachdem evtl. die (im Beispiel nicht enthaltenen) Kandidaten ausgeschlossen wurden, die bei einer Reorganisation in keinem Fall berücksichtigt werden können, erfolgt die Aufzählung der unterschiedlichen Möglichkeiten zum „Packen des Rucksacks". Damit dies zielgerichtet erfolgt, wird eine *Greedy-Heuristik* [ECS93] angewendet, bei der die Kandidaten entsprechend ihrem relativen Nutzen eingeplant werden. Dazu wird für jeden verbliebenen Reorganisationskandidat mit *Gleichung 8.4* das Nutzen-Kosten-Verhältnis (relativer Nutzen) berechnet.

$$D_i = \frac{N_i}{C_i}$$ Gleichung 8.4

Anschließend werden die Reorganisationskandidaten in absteigender Reihenfolge nach dem relativen Nutzen sortiert. Für unser Beispiel ergibt sich damit die in *Tabelle 8.2* gezeigte Situation.

Kandidat k_i	**2**	**10**	**1**	**5**	**4**	**6**	**8**	**9**	**7**	**3**
Nutzen N_i (in %)	2,4	5,3	2,1	3,5	10,8	2,6	3,2	1,2	1,4	1,2
Kosten C_i	2	9	4	12	41	17	23	10	19	25
rel. Nutzen D_i	0,012	0,05889	0,00525	0,00292	0,00263	0,00153	0,00139	0,0012	0,00074	0,00048

Tabelle 8.2: Beispieldaten nach relativem Nutzen sortiert

Zur besseren Darstellbarkeit der weiteren Schritte wird ein Greedy-Vektor $G=(k_2, k_{10}, k_1, k_5, k_4, k_6, k_8, k_9, k_7, k_3)$ definiert, der die für die Kandidaten stehenden Variablen in der Reihenfolge enthält, in der sie zur Reorganisationsdurchführung eingeplant werden können.

Das Einplanen erfolgt von links beginnend. Ein Kandidat *i* wird eingeplant, indem die Variable k_i auf den Wert 1 gesetzt wird. Durch das Einplanen eines Kandidaten werden Kosten verbraucht, die von den noch aufwendbaren Kosten abgezogen werden. Kandidaten werden so lange zur Reorganisation eingeplant, solange die Kosten für die Reorganisation des Kandidaten kleiner sind als die noch aufwendbaren Kosten (sozusagen solange der nächste Kandidat noch in den Rucksack „passt"). Neben der Verringerung der noch aufwendbaren Kosten erhöht sich mit dem Einplanen eines Reorganisationskandidaten der erreichbare Nutzen, d.h. der Wert der Zielfunktion. Kann der nächste Kandidat nicht mehr eingeplant werden und sind die noch aufwendbaren Kosten vollständig verbraucht (was wohl eher einen „Sonderfall"

[19] Zum Fixieren eines Werts wird vermerkt, dass der Wert für alle zukünftigen Betrachtungen im entsprechenden Teil des Entscheidungsbaums festgelegt ist.

darstellen dürfte), so ist die optimale Lösung gefunden und das Verfahren kann beendet werden.

Sind die aufwendbaren Kosten nicht vollständig verbraucht, so wird der Kandidat zur Berechnung einer oberen Schranke für den erreichbaren Nutzen (Zielfunktionswert) entsprechend dem Rest der noch zur Verfügung stehenden aufwendbaren Kosten anteilig eingeplant. Das heißt, die strenge Ganzzahligkeitsforderung für die Variablen k_i wird hierfür aufgehoben. Alle noch nicht belegten Variablen werden auf den Wert 0 gesetzt. Die so gefundene Lösung ist oft (genau dann, wenn Kandidaten anteilig eingeplant wurden) nicht zulässig, da einer Variablen häufig ein gebrochener Wert zugewiesen wurde, eine teilweise Reorganisation von Kandidaten wurde jedoch in Abschnitt 8.2 ausgeschlossen. Der Zielfunktionswert einer zulässigen Lösung muss also unterhalb der errechneten oberen Schranke liegen. Der beste bisher bekannte Zielfunktionswert einer zulässigen Lösung des betrachteten Problems entspricht dem erreichten Wert vor dem anteiligen Einplanen des letzten Kandidaten. Dieser Wert wird als beste bisher bekannte Lösung gemerkt.

Für das Beispiel ergibt sich ausgehend vom Originalproblem die folgende Situation.
G=(1,1,1,1,18/41,0,0,0,0,0)

Nach dem Einplanen von k_2, k_{10}, k_1 und k_5 stehen nur noch 18 Kosteneinheiten zur Verfügung. Eine vollständige Reorganisation des nächsten zu betrachtenden Kandidaten k_4 würde allerdings 41 Kosteneinheiten verbrauchen. Daher wird k_4 anteilig eingeplant. Die obere Schranke für den erreichbaren Nutzen liegt damit bei 18,04% und der derzeit beste erreichte Zielfunktionswert liegt bei 13,3%.

Ist die gefundene Lösung (wie im Beispiel) nicht zulässig, dann existiert genau eine Variable mit einem gebrochenen Wert. Diese Variable (k_4) wird als *Branching-Variable* zum Aufspalten des Problems in zwei Teilprobleme (**P1L** und **P1R**) verwendet. Dabei steht in der Bezeichnung die Ziffer für die jeweilige Ebene im Entscheidungsbaum und die Buchstaben am Ende für „links“ bzw. „rechts“. Das Originalproblem wird mit **P0** bezeichnet. Im linken Teilproblem wird die Branching-Variable mit dem Wert *0* und im rechten Teilproblem mit dem Wert *1* fixiert. Anschließend werden in beiden Teilproblemen die Kandidaten analog zur Vorgehensweise beim Originalproblem zur Reorganisation eingeplant.

Bei der Anwendung des Verfahrens auf das rechte Teilproblem muss allerdings berücksichtigt werden, dass die Branching-Variable mit dem Wert 1 fixiert, also im betrachteten Beispiel k_4 für dieses und alle sich eventuell daraus weiter ergebenden Teilprobleme fest zur Reorganisation eingeplant wurde. Damit wird bereits ein Teil der noch aufwendbaren Kosten verbraucht. Bevor jetzt weitere Kandidaten für eine Reorganisation eingeplant werden, werden die Variablen derjenigen Kandidaten mit dem Wert 0 fixiert, bei denen die Kosten zur Reorganisationsdurchführung den verbliebenen Rest an aufwendbaren Kosten überschreiten. Bei Teilproblem **P1R** betrifft dies die Variablen k_3, k_5, k_6, k_7, k_8, k_9 und k_{10}.

Allgemein heißt das, dass für jedes Teilproblem zunächst die Kosten ermittelt werden müssen, die durch bereits fest eingeplante (fixierte) Reorganisationskandidaten verbraucht werden. Um diesen Wert sind die zur Verfügung stehenden „restlichen“

aufwendbaren Kosten zu verringern. Anschließend sind alle Kandidaten fest von einer Reorganisation auszuschließen, deren Reorganisation mehr als die noch verfügbaren aufwendbaren Kosten verursacht.

Das Ergebnis der Anwendung des Verfahrens auf das Originalproblem und die nach dem ersten Aufspalten entstandenen Teilprobleme zeigt *Abbildung 8.1*. Innerhalb des Greedy-Vektors sind die Variablenbelegungen der einplanbaren Kandidaten kursiv dargestellt. Die Werte der Branching-Variablen, die beim Aufspalten fixiert wurden, sind im Fettdruck und größer dargestellt.

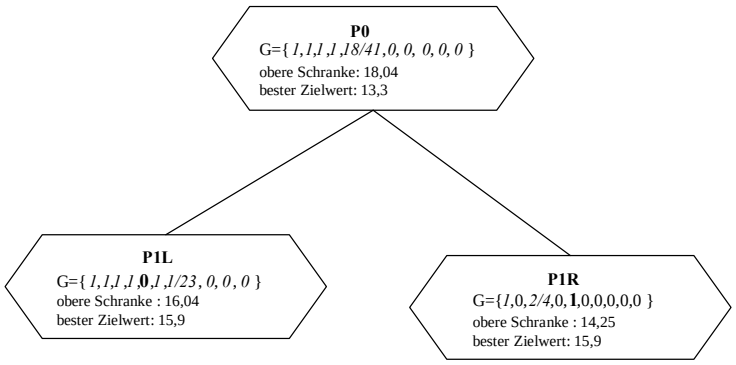


Abbildung 8.1: Aufteilung des Originalproblems und Bearbeitung der Teilprobleme

Bei der Betrachtung des linken Teilproblems ergibt sich eine obere Schranke für den erreichbaren Nutzen von 16,4%. Der größte Zielfunktionswert einer zulässigen Lösung wurde im linken Teilproblem mit 15,9% ermittelt. Da der Wert höher ist als der bisher ermittelte beste Zielfunktionswert einer zulässigen Lösung (13,3% in **P0**), wird die Lösung als neue beste bisher bekannte Lösung gemerkt. Bei der anschließenden Betrachtung des rechten Teilproblems ergibt sich eine obere Schranke von 14,25%. Dieser Wert ist kleiner als der der besten bisher bekannten zulässigen Lösung. Das heißt, auch durch weiteres Verzweigen kann in diesem Teilbaum keine bessere Lösung gefunden werden. Damit braucht das Teilproblem nicht weiter betrachtet werden. Es ist *ausgelotet.* Da die Lösung des linken Teilproblems nicht zulässig, die obere Schranke aber höher als der Zielfunktionswert der besten bisher bekannten Lösung ist, wird dieses Teilproblem mit der Branching-Variable k_8 weiter verzweigt und das Verfahren wird auf die sich ergebenden Teilprobleme angewendet. *Abbildung 8.2* zeigt den gesamten Entscheidungsbaum für das behandelte Beispiel.

Die rechten Teilprobleme müssen jeweils nicht weiter verzweigt werden, da die oberen Schranken für den Zielfunktionswert unter dem Zielfunktionswert der besten bisher bekannten Lösung liegen. Für das Teilproblem **P5L** wird eine zulässige Lösung gefunden. Der Zielfunktionswert ist nicht kleiner als der der besten bisher bekannten Lösung. Damit ist auch dieses Teilproblem ausgelotet und die gefundene Lösung stellt die beste Lösung für die betrachtete Problemstellung dar. Bei der gefundenen Lösung werden k_1, k_2, k_5, k_6 und k_{10} zur Reorganisation eingeplant. Der mit der Reorganisation erreichbare Nutzen liegt bei **15,9%**. Die zu erwartenden Kosten liegen bei **44** Einheiten. Damit wird das zur Verfügung stehende

„Kostenbudget“ zwar nicht vollständig ausgeschöpft, eine bessere Lösung ist unter den gegebenen Umständen jedoch nicht zu erreichen.

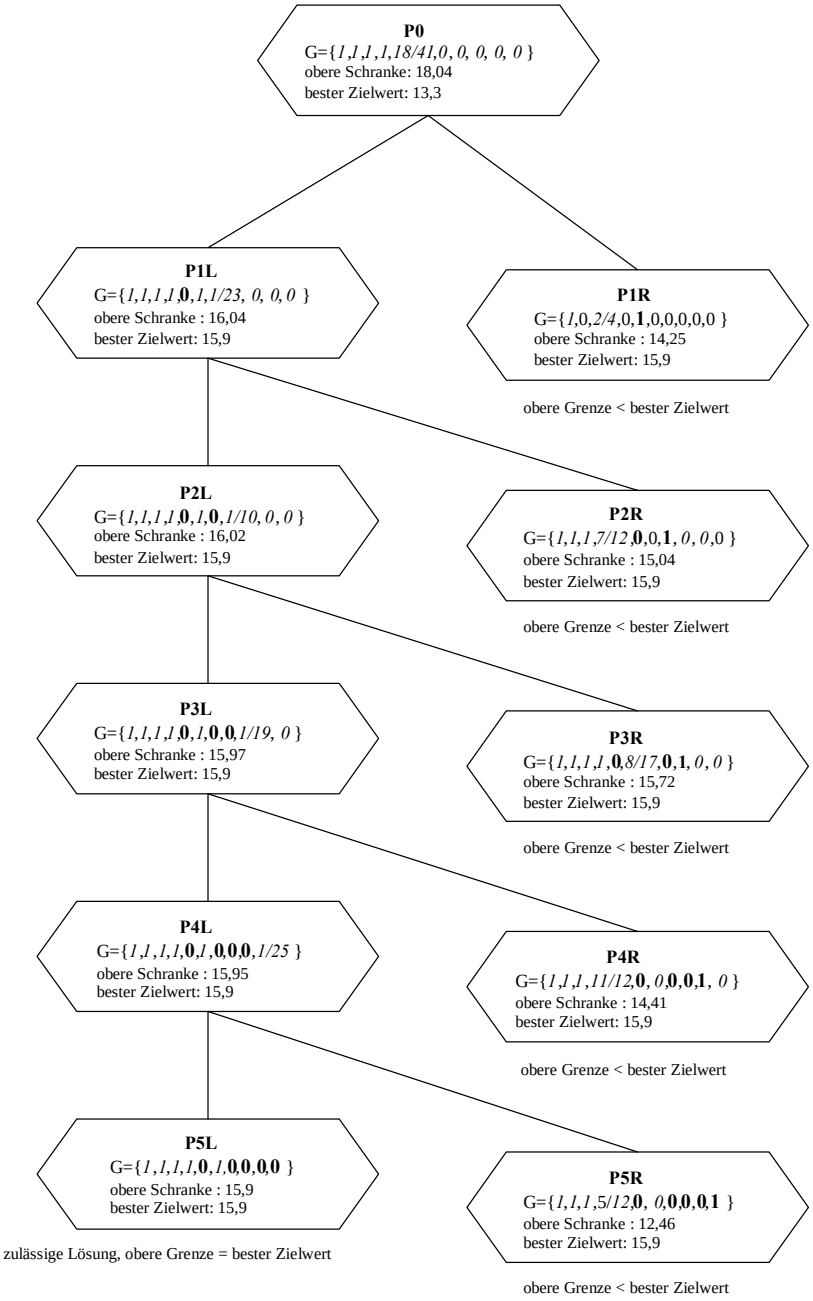


Abbildung 8.2: Verzweigung des Beispielproblems

Die hier angewendete Vorgehensweise zur Lösung des Optimierungsproblems lässt sich relativ einfach implementieren. Eine Umsetzung mit der Sprache C wurde vorgenommen. Eine Pseudocode-Darstellung der einzelnen Funktionen enthält *Anhang G.*

9 Optimierung von Speicherungsstrukturen

Bei objektrelationalen Datenbank-Management-Systemen ergibt sich schnell die Frage nach einer aus Performance-Gesichtspunkten effizienten Abbildung der auf der Anwendungs- und Modellebene existierenden komplexen Objekte auf zur Verfügung stehende physische Speicherungsstrukturen. Zur Beurteilung, ob eine Abbildung als effizient angesehen werden kann, muss wiederum die gegen entsprechenden Objekte gerichtete Workload berücksichtigt werden. Ändert sich diese Workload, so kann eine Änderung der Abbildung (also eine Reorganisation auf der Definitionsebene) zur Anpassung der physischen Struktur vorhandener Daten an ein geändertes physisches Datenbankdesign notwendig werden. Bevor allerdings eine mit u.U. erheblichem Aufwand verbundene physische Neustrukturierung von Datenbankobjekten (Datenbank(teil)reorganisation) erfolgt, sollte überprüft werden, ob die neue Struktur tatsächlich zu einer Verminderung des anfallenden Verarbeitungsaufwands gegenüber der Beibehaltung der aktuellen Struktur führen würde.

In diesem Kapitel wird damit ein weiteres Anwendungsgebiet der ermittelten Informationen über die gegen Datenbankobjekte gerichtete Workload an dem in *Abschnitt 3.4* eingeführten Beispiel des Telekommunikationsdienstes „Universal Number“ vorgestellt. Ziel ist es hierbei, die physische Speicherungsstruktur von Datenbankobjekten so zu gestalten, dass insbesondere die I/O-Kosten für die Abarbeitung der gegen die Datenbankobjekte gerichteten typischen Workload reduziert werden können. Auch hier spielt die Berücksichtigung der Gegebenheiten der konkreten Systemumgebung eine zentrale Rolle. Erste Ergebnisse der Untersuchungen wurden auch in [SD03] vorgestellt.

9.1 Workload-Ermittlung

Um das Potenzial der Umstellung der physischen Speicherung von komplexen Objekten abschätzen zu können, ist es notwendig, die auf die Datenbankobjekte angewendete Workload möglichst genau zu kennen. Die Vorgehensweise zur Ermittlung der Workload-Informationen entspricht in großen Teilen jener bei der Quantifizierung des Nutzens von Datenbankreorganisationen bezüglich der Systemleistung. Lediglich die weitere Nutzung der gewonnenen Informationen erfolgt etwas anders. Bei einer möglichen Implementierung der Verfahren kann ein großer Teil der Komponenten für beide Zwecke genutzt werden.

Einen Überblick über die Methode zur Bewertung alternativer physischer Repräsentationen von komplexen Objekte zeigt *Abbildung 9.1*, wobei die Ähnlichkeiten mit *Abbildung 6.10* deutlich werden. Auch die hier beschriebene Methode beruht auf der Möglichkeit, die an das DBMS gerichteten (SQL-)Anfragen zunächst zu protokollieren. Neben der Protokollierung der eigentlichen Anweisung ist es auch notwendig, die Häufigkeit festzuhalten, mit der eine Anweisung ausgeführt wird. Da für eine korrekte Ermittlung des Operations-Mixes die Protokollierung in einem als repräsentativ anzusehenden Zeitraum erfolgen muss, kann im Prinzip für beide Zwecke als Ausgangspunkt das gleiche Workload-Protokoll verwendet werden.

Aus den protokollierten Anweisungen werden dann in **Schritt I** von Abbildung 9.1 durch den Datenbankadministrator wieder die für die Profilerstellung relevanten Operationen ausgewählt. Die Auswahl erfolgt bei der Bewertung verschiedener physischer Repräsentationen komplexer Objekte i.d.R. bezogen auf das konkrete betrachtete Objekt. Das heißt, es werden die auf das komplexe Objekt und dessen Unterobjekte angewendeten Anweisungen ausgewählt. Die weiteren Betrachtungen erfolgen dann aus Gründen der Übersichtlichkeit i.d.R. ebenfalls lokal (vgl. *Abbildung 6.3*), also nicht auf die Gesamt-Workload bezogen. Es ist aber auch genauso möglich und teilweise nötig, die Auswirkungen der Umstellung bezogen auf die Gesamt-Workload zu betrachten.

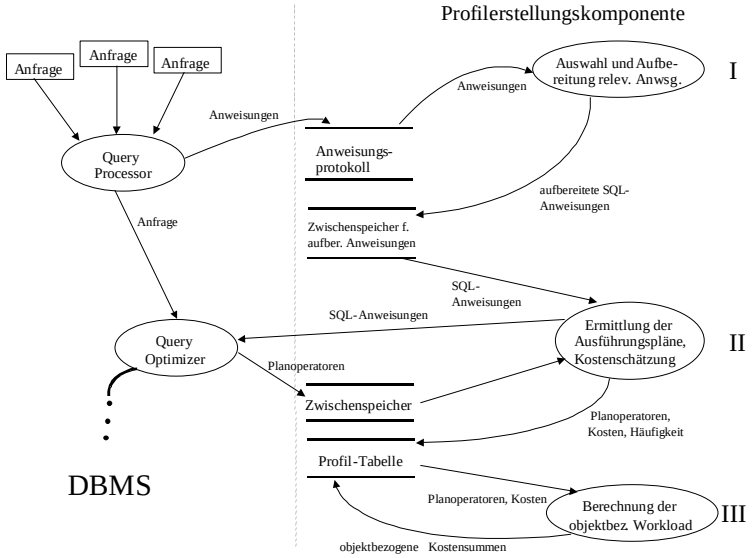


Abbildung 9.1: Methode und Komponenten zur Bewertung alternativer physischer Repräsentationen komplexer Objekte

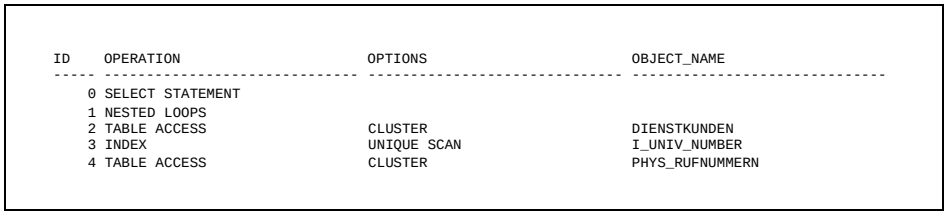
Anschließend werden die ausgewählten Anweisungen wiederum in ein Format gebracht, das später zur Erstellung der jeweiligen Ausführungspläne verwendet werden kann. In Abbildung 9.2 ist ein Auszug aus einem solchen Protokoll dargestellt, das mit einem für Oracle implementierten Prototyp eines Werkzeugs anhand des hier verwendeten Beispiels erzeugt wurde.

```
ST_ID ZNR  SQL_TEXT                                                                    ANZ_AUSF
----- --- ------------------------------------------------------------------------- ---------
   54   1 SELECT * FROM dienstkunden ORDER BY Univ_Number                                    1
   44   1 SELECT dienstkunden.geb_satz,phys_rufnummern.phys_rufnummer  FROM dienstkunden,    25000
   44   2 phys_rufnummern WHERE dienstkunden.Univ_Number=10093 AND                           25000
   44   3 dienstkunden.Univ_Number=phys_rufnummern.Univ_Number                               25000
```

Abbildung 9.2: Protokollierte Beispielanweisungen in relationaler Form im Anweisungsprotokoll gespeichert

Dabei zeigt die Anweisung mit der Identifikationsnummer 44 die Suche der zu einem Dienstkunden gehörenden Daten, die für die Vermittlung eines Anrufs benötigt werden. Hier wurden im Protokollierungszeitraum insgesamt 25 000 Anrufe vermittelt.

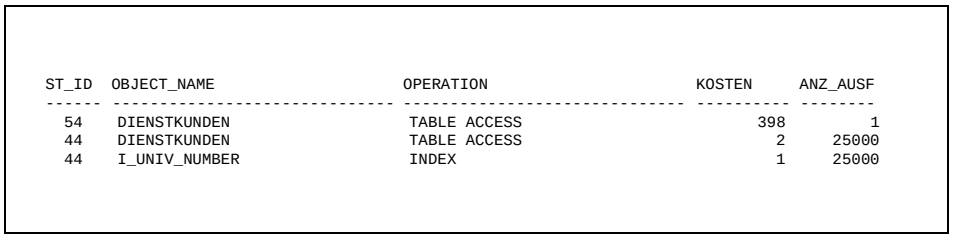
In **Schritt II** erfolgt (ebenfalls unter Nutzung des Anfrageoptimierers des DBMS) die Ermittlung der zu den jeweiligen Anweisungen gehörenden Ausführungspläne, d.h. der zur Ausführung der Anweisung notwendigen Planoperatoren. Mit deren Hilfe können die auf der satzorientierten Datenbankschnittstelle für die jeweiligen Operatoren anfallenden Kosten geschätzt werden. *Abbildung 9.3* zeigt einen Ausschnitt aus dem vom Anfrageoptimierer gelieferten Ausführungsplan der eben betrachteten Beispielanweisung. Die Daten der Dienstkunden und die ihnen zugeordneten physischen Rufnummern liegen in diesem Beispiel objektbezogen geclustert vor (vgl. *Abbildung 3.20*).



ID	OPERATION	OPTIONS	OBJECT_NAME
0	SELECT STATEMENT		
1	NESTED LOOPS		
2	TABLE ACCESS	CLUSTER	DIENSTKUNDEN
3	INDEX	UNIQUE SCAN	I_UNIV_NUMBER
4	TABLE ACCESS	CLUSTER	PHYS_RUFNUMMERN

Abbildung 9.3: Auszug aus dem Ausführungsplan

Die vom Anfrageoptimierer gelieferten Informationen werden vom implementierten Prototyp weiter aufbereitet und in einen Gesamtplan eingetragen, dessen Erstellung sich etwas von der in Abschnitt 6.3.2.4 beschriebenen Vorgehensweise unterscheidet. So werden hier bei der Aufbereitung der vom Anfrageoptimierer gelieferten Daten die einzelnen zu einer Anweisung gehörenden Planoperatoren u.a. mit einer Identifikationsnummer (ST_ID) versehen, um z.B. auch die für Indexzugriffe anfallenden Kosten der gegen eine Tabelle gerichteten Workload zuordnen zu können. Einen Ausschnitt aus dem Gesamtplan zeigt *Abbildung 9.4*. Anschließend können die bei der Ausführung der jeweiligen Planoperatoren anfallenden Kosten berechnet werden.



ST_ID	OBJECT_NAME	OPERATION	KOSTEN	ANZ_AUSF
54	DIENSTKUNDEN	TABLE ACCESS	398	1
44	DIENSTKUNDEN	TABLE ACCESS	2	25000
44	I_UNIV_NUMBER	INDEX	1	25000

Abbildung 9.4: Ausschnitt aus dem Gesamtplan

In Abbildung 9.4 ist ein Ausschnitt der auf die Tabelle `DIENSTKUNDEN` und den zugehörigen Index `I_UNIV_NUMBER` (vgl. *Abbildung 3.17*) im Protokollierungszeitraum angewendeten Planoperatoren samt Ausführungshäufigkeit und Kostenschätzung dargestellt. Dabei zeigt die erste Zeile die Kostenschätzung für eine auf die Tabelle angewendete sequenzielle Suche. Die Kostenschätzung für einen Eintupelzugriff über einen Index zeigen die zweite und dritte Zeile.

In **Schritt III** wird die Summe der durch die Planoperatoren verursachten Kosten, multipliziert mit den jeweiligen Ausführungshäufigkeiten, ermittelt. Diese kann als Maß für den anfallenden Aufwand zur Abarbeitung der gegen das Objekt gerichteten Workload angesehen werden. An dieser Stelle soll nochmals betont werden, dass es

für den Zweck der Ermittlung der Auswirkungen der Umstellung der physischen Speicherungsstruktur von Datenbankobjekten wichtig ist, die *gesamte gegen das Objekt gerichtete Workload* zu betrachten, da Veränderungen in der Struktur der physischen Speicherung von Datenbankobjekten i.d.R. auf Teile dieser Workload positive und auf andere Teile negative Auswirkungen haben.

9.2 Bewertung einer alternativen physischen Repräsentation

Die Auswahl der Ausführungspläne und die Ermittlung von Kostenwerten durch Anfrageoptimierer basiert auf Meta- und Statistikdaten, die für diese Zwecke festgehalten werden. Hier bietet sich ein Ansatzpunkt für die Bewertung alternativer physischer Repräsentationen komplexer Objekte, indem die neue Speicherungsstruktur zunächst nur simuliert wird. Ein evtl. aufwendiges „verdachtsweises" Bewegen von Daten, das mit der Durchführung einer Reorganisation auf der Definitionsebene bei bereits bestehenden Objekten verbunden wäre, kann zunächst vermieden werden. Die Reorganisation braucht erst dann erfolgen, wenn bekannt ist, dass von einer veränderten physischen Struktur tatsächlich Verbesserungen in hinreichendem Maße zu erwarten sind. Voraussetzung dafür ist auch, dass der Anfrageoptimierer in der Lage ist, durch die Veränderungen an den physischen Strukturen kostengünstiger ausführbare Planoperatoren zur Workload-Abarbeitung auszuwählen.

Für diese Simulation von Datenbankobjekten und den zugehörigen physischen Speicherungsstrukturen müssen zunächst die entsprechenden Meta- und Statistikdaten erzeugt werden. Anschließend werden die oben beschriebenen **Schritte II** und **III** zur Ermittlung der Workload-Informationen ausgeführt. Der Vergleich der Summen der Kostenschätzungen für verschiedene physische Repräsentationen eines betrachteten Datenbankobjekts gibt Auskunft über den bei der Verwendung der jeweiligen physischen Struktur zu erwartenden Aufwand zur Workload-Abarbeitung.

Die Erzeugung der für die Simulation benötigten Meta- und Statistikdaten ist allerdings nicht ganz trivial, da neben dem Verständnis der Inhalte genaue Kenntnisse über die Struktur des jeweiligen Datenbankkatalogs und über die Stellen im Katalog vorhanden sein müssen[20], an denen die entsprechenden Einträge erzeugt werden müssen. Problematisch sind hier insbesondere auch die bereits erwähnten starken Unterschiede im Aufbau der Kataloge einzelner DBMS-Produkte und die im Bereich der physischen Beschreibungsinformationen von Datenbankobjekten fehlende Standardisierung.

Um die Probleme beim Erzeugen der für die Simulation benötigten Meta- und Statistikdaten zu verringern, bietet sich die folgende Vorgehensweise an. Zunächst werden auf einem Referenzsystem oder auch auf dem Ausgangssystem für die zu überprüfende physische Repräsentation leere Strukturen mit den entsprechenden

[20] Dabei werden entsprechende Zugriffsrechte auf den Datenbankkatalog zur teilweise notwendige Anpassung von für die Simulation wichtigen Statistikdaten vorausgesetzt.

DDL-Anweisungen erzeugt. Dabei werden somit auch die zugehörigen Metadaten mit erzeugt und in den Datenbankkatalog eingetragen.

Im nächsten Schritt müssen die die Strukturen beschreibenden Statistikdaten erzeugt (geschätzt oder berechnet) und an entsprechender Stelle eingetragen werden. Wichtig sind hier insbesondere für die vorgesehenen Zwecke Informationen über den benötigten Speicherplatz bzw. Informationen über die Anzahl Blätter und die Höhen von Indexbäumen usw. Diese Informationen lassen sich bspw. mit den in *Anhang B* beschriebenen Vorgehensweisen mit ausreichender Genauigkeit abschätzen. Muss die Simulation auf einem Produktionssystem vorgenommen werden, so muss sichergestellt sein, dass die simulierten Strukturen keine negativen Auswirkungen auf den parallel laufenden normalen Datenbankbetrieb (bspw. durch die Erzeugung fehlerhafter oder ungünstiger Ausführungspläne) haben.

Wurden die benötigten Statistikdaten erzeugt, so kann die Workload-Ermittlung wie oben beschrieben erfolgen und die für verschiedene physische Repräsentationen ermittelten Werte können miteinander verglichen werden.

9.3 Überprüfung an einem Beispiel

Die Anwendung des vorgestellten Verfahrens zur Bestimmung möglicher Einsparungspotenziale durch eine Umstellung der physischen Speicherung sowie eine Überprüfung der Genauigkeit der mit dem Verfahren ermittelten Schätzwerte sollen nun am Beispiel des Telekommunikationsdiensts „Universal Number“ (vgl. Abschnitt 3.4.1) gezeigt werden. Dazu wurde auf zwei unterschiedliche physische Repräsentationen des dargestellten Umweltausschnitts eine Beispiel-Workload angewendet. Als Ausgangspunkt wurde die physisch getrennte Speicherung aller drei Tabellen gewählt. Zur Initialisierung wurden zunächst jeweils die Daten von **5000** Dienstkunden mit jeweils durchschnittlich zehn zugehörigen realen Rufnummern künstlich (per Zufallszahlengenerator) erzeugt und in die entsprechenden Tabellen eingetragen.

Danach wird in der eigentlichen Lastsimulation eine Menge an Verbindungsdatensätzen erzeugt. Weiterhin werden einmalig die Gebührenabrechnung für die Dienstkunden sowie die Berechnung des Gesamtumsatzes des Telekommunikationsunternehmens simuliert. Die einzelnen vom Simulationsprogramm ausgeführten Anweisungen sowie die Häufigkeit von deren Anwendung werden für die Workload-Messung protokolliert.

Als erster Fall soll die *objektübergreifende (tabellenbezogene) Cluster-Bildung* (vgl. *Abbildung 3.19*) betrachtet werden, bei der, wie bei relationalen DBMS allgemein üblich, die drei Tabellen auch in drei physisch getrennten Speicherbereichen (also tabellenbezogen geclustert) abgelegt werden. Eine Indexierung über B^*-Baum-Indexe erfolgt über die jeweiligen Primärschlüssel sowie über die Fremdschlüssel (`Univ_Number` der Tabelle `PHYS_RUFNUMMERN` und `Gew_Nr` der Tabelle `VERBINDUNGEN`). Je „Anruf“ werden damit im Beobachtungszeitraum folgende Operationen ausgeführt:

- Ein Zugriff auf die Tabelle `DIENSTKUNDEN` über den Primärschlüsselindex, um den Gebührensatz zu ermitteln.
- Zugriffe auf die realen Rufnummern des jeweiligen Dienstkunden in der Tabelle `PHYS_RUNFNUMMERN` unter Nutzung des Index über dem Fremdschlüssel `Univ_Number`.
- Ein Zugriff auf die Tabelle `VERBINDUNGEN` zum Eintragen des Verbindungssatzes sowie Zugriffe zum Erzeugen der Einträge in die beiden über der Tabelle definierten Indexe.

Zur Abrechnung der Gebühren wird die Tabelle `DIENSTKUNDEN` sequenziell durchlaufen. Zu jedem Dienstkunden werden über den Index auf `Gew_Nr` die entsprechenden Verbindungsdatensätze gelesen. Zur Umsatzberechnung wird die Tabelle `VERBINDUNGEN` einmal sequenziell durchlaufen.

Im zweiten Fall soll zur Gegenüberstellung die *objektbezogene (tabellenübergreifende) Clusterung* betrachtet werden, die mit den von Oracle gebotenen Möglichkeiten zur tabellenübergreifenden Cluster-Bildung (vgl. *Abschnitte 3.3.4* und *3.4.2*) realistisch simuliert werden kann. Dabei werden die Daten der Dienstkunden und die zugehörigen realen Rufnummern auch physisch beieinanderliegend gespeichert. Der gemeinsame Zugriffspfad über die Dienstkundennummer wird dabei über einen B^*-Baum-Index realisiert. Die Speicherungsform und die Indexierung der Tabelle `VERBINDUNGEN` bleibt unverändert. Im Beobachtungszeitraum fallen damit jetzt je Anruf

- ein Zugriff auf die Tabellen des Clusters über den Cluster-Index und
- ein Zugriff auf die Tabelle `VERBINDUNGEN` zum Eintragen des Verbindungssatzes sowie die Zugriffe zum Erzeugen der Einträge in die beiden über der Tabelle definierten Indexe an.

Zum Ermittlen der realen Rufnummern werden durch die Clusterung jetzt im Rahmen der Verbundoperation lediglich zwei weitere logische Blockzugriffe (auf das entsprechende Blatt des Cluster-Index und auf den Datenblock) nötig. Die entsprechenden Blöcke wurden (wenn sie nicht schon im Puffer vorlagen) zum Auffinden der Daten der jeweiligen Dienstkunden gelesen.

Zur Abrechnung der Gebühren wird die Tabelle `DIENSTKUNDEN` wieder sequenziell durchlaufen. Dabei steigt der Aufwand etwas an, weil jetzt der gesamte Cluster durchsucht werden muss. Zu jedem Dienstkunden werden unter Nutzung des über `Gew_Nr` existierenden Index die entsprechenden Verbindungsdatensätze gelesen. Zur Umsatzberechnung wird hier ebenfalls die Tabelle `VERBINDUNGEN` einmal sequenziell durchlaufen.

Zur Verdeutlichung der Wirkungen der Umstellung der physischen Speicherung wurde die beschriebene Beispiel-Workload mehrfach auf die beiden Implementierungen angewendet. Dabei wurde die Anzahl der simulierten Anrufe schrittweise erhöht. Bei der Aufnahme der einzelnen Messreihen wurde die Anzahl erfolgter logischer sowie die Anzahl physischer Blockzugriffe festgehalten. Über die von Oracle zur Verfügung gestellten Schnittstellen wurden allerdings nur Zugriffe

ermittelt, die im Rahmen von Leseoperationen erfolgt sind. Eine genaue Erfassung der Anzahl der reinen Schreibzugriffe (im Rahmen des Einfügens von Datensätzen über die Verbindungen) war wegen der asynchronen Ausführung von Schreiboperationen durch Page Cleaner nicht möglich. Eine Zuordnung zwischen tatsächlich ausgeführten Schreiboperationen und den im Rahmen der Einfügeoperationen vorgenommenen Änderungen an Datenblöcken konnte nicht vorgenommen werden.

Neben der Anzahl jeweils erfolgter Blockzugriffe wurden auch die Ausführungszeiten gemessen, um die Genauigkeit der für das Beispiel ermittelten Werte überprüfen zu können. Die Ergebnisse zeigt *Abbildung 9.5* für die verschiedenen Anrufzahlen.

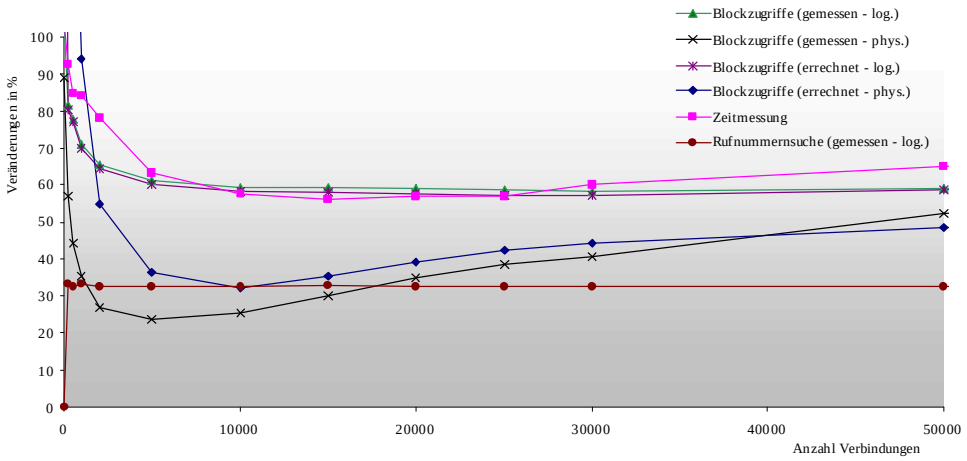


Abbildung 9.5: Kostenreduktion durch tabellenübergreifende Clusterung

Die dargestellten sechs Kurven zeigen das Verhältnis des Aufwands, der zur Abarbeitung der beschriebenen Workload bei der tabellenübergreifenden bzw. bei der tabellenbezogenen Clusterung der Daten von Dienstkunden und zugehörigen realen Rufnummern anfällt. Das heißt, an den Messpunkten wurden bei beiden Speicherungsformen die entsprechenden Aufwandswerte absolut ermittelt. Danach wurden die bei der tabellenübergreifenden Clusterung jeweils ermittelten Werte durch die bei Verwendung der tabellenbezogenen Clusterung ermittelten Werte geteilt. Dabei zeigt sich, dass sich im vorliegenden Beispiel mit zunehmender Zahl der Verbindungen das geschätzte Kostenverhältnis bezüglich der gemessenen *logischen* Blockzugriffe bei etwa **58%** einpegelt. Ähnliche Werte ergaben sich bei der Zeitmessung. Das bedeutet, dass bei der Umstellung von tabellenbezogener auf tabellenübergreifende Clusterung im betrachteten Beispiel eine Kosteneinsparung von etwa **42%** erwartet werden kann. Die mit der Zahl der Verbindungen zunehmende Aufwandsreduzierung lässt sich mit dem größer werdenden Anteil der auf Dienstkunden und reale Rufnummern angewendeten Verbundoperationen begründen, die für das Finden der realen Rufnummern je Anruf nötig sind und die durch die tabellenübergreifende Clusterung besonders gut unterstützt werden.

Zur Vermittlung einer Verbindung müssen zunächst die Daten des über die gewählte Nummer identifizierten Dienstkunden geprüft und die Menge der zugehörigen realen

Rufnummern ermittelt werden. Dabei sind jedem Dienstkunden in der Beispielumgebung ca. 10 solcher Rufnummern zugeordnet. Zur Ausführung der notwendigen Verbundoperation zwischen den Tabellen `DIENSTKUNDEN` und `PHYS_RUFNUMMERN` fallen bei der physisch getrennten Speicherung der Tabellen ca. fünfzehn Blockzugriffe an. Durchschnittlich 12 Zugriffe sind dabei zur Ermittlung der Rufnummern notwendig. Die Zugriffe auf beide Tabellen erfolgen jeweils über Indexe mit zwei Ebenen. Bei der tabellenübergreifenden Clusterung erfolgen die Zugriffe über den Cluster-Index, der ebenfalls zwei Ebenen besitzt. Zum Ermitteln der Daten und zur Ausführung der Verbundoperation fallen hier fünf Blockzugriffe an. Damit ergibt sich rechnerisch eine Aufwandsreduzierung in Bezug auf I/O-Kosten von ca. **67%**. Dies zeigen auch die Messwerte für die Rufnummernsuche (untere Kurve in Abbildung 9.5).

Der Aufwand für die Gebührenabrechnung ist bei der tabellenübergreifenden Clusterung höher als bei der physisch getrennten Speicherung von Dienstkundendaten und Rufnummern. Zur Gebührenabrechnung wird die Tabelle `DIENSTKUNDEN` sequenziell durchsucht. Für jeden Dienstkunden wird unter Nutzung des über der Tabelle `VERBINDUNGEN` definierten Fremdschlüsselindex auf dessen Verbindungsdaten zugegriffen. Da bei der tabellenübergreifenden Clusterung die Rufnummern physisch in die Daten der Dienstkunden eingebettet gespeichert werden, fällt hier beim sequenziellen Durchsuchen des Cluster-Segments ein höherer Aufwand an. In Abbildung 9.5 ist das an den ersten Messwerten zu erkennen. Da diese für eine Workload ermittelt wurden, in der keine Anrufe vermittelt wurden, ergibt sich durch die Umstellung der physischen Speicherungsstrukturen zunächst sogar eine Aufwandserhöhung.

Bei der Vermittlung von Anrufen fällt weiterhin noch Leseaufwand zum Einfügen der Verbindungsdaten (realisiert über einzelne `INSERT`-Anweisungen) an. Über der Tabelle Verbindungen existieren zwei Indexe, die bei 20 000 Verbindungen im Beispiel jeweils eine Höhe von zwei Ebenen haben. Durch die ebenfalls notwendigen Zugriffe auf die Speicherverwaltungsinformationen von Tabellen- und Indexsegmenten, die wegen des hohen Lokalitätsverhaltens im Puffer gehalten werden, werden zum Einfügen eines Verbindungsdatensatzes und der notwendigen Indexeinträge ca. acht logische Blockzugriffe ausgeführt (drei je Index und zwei für den eigentlichen Datensatz). Dieser Aufwand ist von der Umstellung der physischen Speicherungsstruktur nicht betroffen. Gleiches gilt für die Umsatzberechnung, die über sequenzielles Durchsuchen der Tabelle `VERBINDUNGEN` realisiert wird.

Bei der tabellenbezogenen Clusterung fallen zur Vermittlung von 20 000 Anrufen ca. 460 000 logische Blockzugriffe an. Wird die tabellenübergreifende Clusterung genutzt, so sind 260 000 logische Blockzugriffe nötig. Für die Zugriffe auf die Verbindungsdaten zur Gebührenabrechnung (die ebenfalls von der Umstellung der Speicherungsstrukturen unabhängig sind) werden etwa 30 000 logische Zugriffe ausgeführt. Die errechnete Aufwandsreduzierung bezogen auf die Gesamt-Workload liegt damit hier bei ca. **41%**. Die Auswirkungen des je einmalig auftretenden sequenziellen Suchens im Rahmen der Gebührenabrechnung und zur Umsatzberechnung, mit jeweils weniger als 1 000 logischen Blockzugriffen, werden hier dem relativen Anteil entsprechend vernachlässigt.

Die Berechnungen und Messungen bezüglich der *physischen* Blockzugriffe zeigen für das betrachtete Beispiel bei wenigen Verbindungen noch deutlichere Verbesserungen. Dies liegt daran, dass zunächst ca. 95% der Zugriffe im Puffer erfolgt sind. Durch die mit der steigenden Anzahl Verbindungen wachsende Datenmenge sinkt der Pufferungsgrad (besonders beim Einfügen der Verbindungsdaten sowie bei Gebühren- und Umsatzberechnungen), was auch an den Aufwandskurven zu erkennen ist. Lokalitätseffekte, die bspw. durch unterschiedlich intensive Nutzung der Angebote der verschiedenen Dienstkunden auftreten können, beinhaltet das vorliegende Beispiel dabei nicht. Die Abweichungen zwischen errechneten und gemessenen Aufwandswerten sind hier größer als bei den Werten für die logischen Blockzugriffe. Dies kann auf Ungenauigkeiten bei der Ermittlung der Pufferungsgrade zurückgeführt werden. Mit der in Abschnitt 5.4.4 vorgestellten Methode können über längere Zeit auftretende Schwankungen von Pufferungsgraden nicht genau erfasst werden.

Insgesamt gesehen liegen die auf Grund der Kostenschätzungen zu erwartende Aufwandsreduzierung und die tatsächliche Reduzierung eng beieinander. Die stärkeren Abweichungen im linken Teil von Abbildung 9.5 ergeben sich auch aus der in der Test-Durchführung per Zufallszahlengenerator getroffenen Auswahl der Dienstkunden und Rufnummern sowie der anfangs geringen Anzahl getätigter Anrufe.

10 Zusammenfassung und Ausblick

10.1 Ausgangssituation

Durch die Nutzung von relationalen und objektrelationalen Datenbank-Management-Systemen kann auf der Anwendungsebene ein hoher Grad an Datenunabhängigkeit erreicht werden, der es ermöglicht, dass physische Aspekte der Datenspeicherung bei der Anwendungsentwicklung „im Prinzip“ nicht berücksichtigt werden müssen. Dies gilt so allerdings nicht für den *Betrieb und die Administration* von Anwendungssystemen, die auf großen Datenbanken basieren.

Um ein zufriedenstellendes Antwortzeitverhalten sicherzustellen, müssen aus Administratorsicht (und teils aus Anwendersicht) physische Aspekte der Datenspeicherung intensiv betrachtet werden. So sind für große Datenbanken u.a. Konzepte zu entwickeln, wie Daten auf die zur Verfügung stehenden Datenträger so verteilt werden können, dass eine möglichst gute Lastverteilung erreicht wird und keine Engpässe entstehen. Darüber hinaus kann durch eine günstige Abbildung der auf der Anwendungsebene existierenden logischen Datenstrukturen auf physische Speicherungsstrukturen – physischen Datenbankentwurf also – der zur Abarbeitung der Datenbank-Workload notwendige Aufwand (unter den gegebenen Umständen) gering gehalten werden, was sich positiv auf die Systemleistung auswirkt. Das heißt, dass zumindest (aber nicht nur) für Datenbankadministratoren Aspekte der physischen Datenspeicherung eine hohe Relevanz besitzen.

Hier ergibt sich allerdings schnell das Problem, dass die zur Entwicklung eines auf eine konkrete Systemumgebung abgestimmten Speicherungskonzepts nötige Übersicht über die Strukturen der Datenbank und die gegen die Datenbank gerichtete Workload bei großen Systemen mit tausenden Tabellen schnell verloren geht bzw. gar nicht erlangt werden kann. Eine Unterstützung durch Administrations- und Tuning-Werkzeuge ist hier dringend erforderlich. Dieser Bedarf wurde auch von Herstellern von DBMS-Produkten und Drittanbietern erkannt. So sind bspw. Werkzeuge (für DB2 z.B. unter den Namen Index Advisor bzw. Design Advisor) verfügbar, die Administratoren bei der Definition von Zugriffspfaden unterstützen und Vorschläge zum Kreieren oder Löschen von Indexen („und mehr“) unterbreiten. Allerdings werden mit diesen Werkzeugen Probleme meist nur lokal betrachtet. Ob und welche Auswirkungen vorgenommene Änderungen (z.B. das Anlegen eines neuen Index zur Beschleunigung bestimmter Abfragen) auf das Gesamtsystem haben, kann wegen fehlender Informationen oft nicht betrachtet werden.

Ein anderes Problem beim Betreiben von großen Datenbanken sind in den physischen Speicherungsstrukturen auftretende Degenerierungen aller Art, weil die Strukturen bei Änderungsoperationen aus Performance-Gründen nicht bzw. nicht sofort vollständig gepflegt werden. Diese Degenerierungen verursachen meist eine Erhöhung der Verarbeitungskosten, weil z.B. Überlaufbereiche durchsucht werden müssen, Indirektionen in Datenbereichen verfolgt werden müssen oder Sortieroperationen aufwendiger oder überhaupt erst erforderlich werden. Eingestreuter Freiplatz verursacht neben einem erhöhten Verarbeitungsaufwand auch einen Speicherplatzmehrbedarf. Zur Beseitigung vorhandener Degenerierungen

müssen zyklisch Wartungsarbeiten (*Datenbankreorganisationen*) ausgeführt werden. Um gezielt die Bereiche zu reorganisieren, die Degenerierungen enthalten, ist deren genaue Lokalisierung notwendig. Auch hier sind einige Werkzeuge verfügbar, die Datenbankadministratoren diesbezüglich unterstützen.

Diese Werkzeuge erkennen in Datenbankobjekten vorhandene Degenerierungen und empfehlen bei Über- bzw. Unterschreitung bestimmter Schwellwerte die Durchführung von Reorganisationsmaßnahmen, die ebenfalls von angebotenen Werkzeugen unterstützt wird. Abhängig von der verwendeten Reorganisationsmethode ergeben sich bei der Durchführung von Datenbankreorganisationen allerdings Einschränkungen im normalen Datenbankbetrieb. Bei der Durchführung von Reorganisationen im Offline-Betrieb sind die betroffenen Datenbankobjekte während der Reorganisation für Benutzerzugriffe nicht verfügbar, bei Online-Reorganisationen werden zumindest Ressourcen für die Reorganisationsdurchführung gebunden, wodurch sich aus Anwendungssicht eine Verringerung der Systemleistung darstellt. Dies steht oft deutlich im Widerspruch zu existierenden Verfügbarkeitsanforderungen. Durch die Priorisierung von Wartungsarbeiten können solche Einschränkungen verringert werden. Verfügbare Werkzeuge zur Ermittlung eines eventuell vorhandenen Reorganisationsbedarfs führen die Analysen aber i.d.R. „statisch“ aus, d.h. ohne Berücksichtigung der auf die betrachteten Datenbankobjekte angewendeten Workload. Eine Berücksichtigung von Informationen über die Workload ist allerdings zwingend notwendig, wenn die Auswirkungen vorhandener Degenerierungen in einer konkreten Anwendungsumgebung genauer als mit der pauschalen Aussage, dass vorhandene Degenerierungen negative Auswirkungen auf die Systemleistung haben, beurteilt werden sollen.

Die eigentliche Reorganisationsdurchführung wird ebenfalls in der Fachliteratur ausführlich behandelt. Mit den beschriebenen Verfahren wird i.d.R. auch das Ziel verfolgt, den mit der Reorganisationsdurchführung verbundenen Aufwand und die Einschränkungen bezüglich der Verfügbarkeit von Daten möglichst gering zu halten.

Eine mit der Weiterentwicklung und Verbreitung objektrelationaler Datenbank-Management-Systemen einhergehende größere Vielfalt verfügbarer Datenstrukturen und die damit wachsende Vielfalt von Kombinations- und z.B. Schachtelungsmöglichkeiten [Luf05] lässt auch wachsenden Unterstützungsbedarf bei der Abbildung logischer Datenstrukturen auf physische Speicherungsstrukturen erwarten [Ska06].

10.2 Ergebnisse der Arbeit

Das Thema Datenbankreorganisation allgemein (Begriffsbestimmung, Ansatzpunkte usw.) wird in der Literatur teils bereits für vorrelationale System behandelt. In der vorliegenden Arbeit werden *Datenbankreorganisationen* deshalb zunächst *im Kontext relationaler und objektrelationaler Datenbank-Management-Systeme* betrachtet (*Kapitel 2*). Dazu werden Ebenen, auf denen Reorganisationen erfolgen können, verschiedene Reorganisationsmethoden und Granulate zur

Reorganisationsdurchführung im Überblick behandelt. Weiterhin erfolgt eine für die vorliegende Arbeit geltende Abgrenzung und Einordnung des Begriffs Datenbankreorganisation.

Trotz Unterschieden in der Verwendung von Begriffen und Details der Implementierung ähneln die von gängigen DBMS-Produkten verwendeten physischen Speicherungsstrukturen und die darauf angewendeten Algorithmen einander. Als Grundlage für die weiteren Ausführungen wurden deshalb *verbreitet verwendete Speicherungsstrukturen und Speicherungskonzepte im Überblick* betrachtet (*Kapitel 3*). In die Betrachtungen ist auch das Verhalten üblicher über den Speicherungsstrukturen implementierter Operationen eingeschlossen. Die Entstehung von Degenerierungen, deren Auswirkungen und Maßnahmen, mit denen die Entwicklung von Degenerierungen verzögert werden kann, werden dabei ebenfalls mit behandelt. Neben den physischen Speicherungsstrukturen und Degenerierungen werden auch Konzepte zur horizontalen Partitionierung und zur Cluster-Bildung beschrieben, die ebenfalls Aspekte der physischen Speicherung darstellen. Szenarien zur sinnvollen Anwendung dieser Konzepte werden skizziert.

Daran schließt sich ein *Überblick über wissenschaftliche Arbeiten* an, die in Zusammenhang mit der Thematik Reorganisation von Datenbanken bzw. Dateisystemen stehen (*Kapitel 4*). Ein großer Teil dieser Arbeiten behandelt Vorgehensweisen zur Reorganisation bestimmter physischer Speicherungsstrukturen. Dabei werden in den Beiträgen neben Verfahren zur Beseitigung von Degenerierungen auch Verfahren zur Konvertierung von Speicherungsstrukturen behandelt. In die Ausführungen der vorliegenden Arbeit sind auch Beiträge, die die Thematik in Zusammenhang mit konkreten DBMS-Produkten (DB2, Informix, Oracle) betrachten und ein *Überblick über die Leistungsfähigkeit einer Auswahl verfügbarer Werkzeuge* zur Unterstützung von Datenbankadministratoren bei Reorganisationsbedarfsanalysen und Reorganisationsdurchführung eingeschlossen.

Den Schwerpunkt der vorliegenden Arbeit bildet die Entwicklung von Vorgehensweisen zur *Prognostizierung des von Datenbankreorganisationen zu erwartenden Nutzens* und der mit der Reorganisationsdurchführung verbundenen *Kosten.* Im Vordergrund stehen dabei Reorganisationen zur Beseitigung von Degenerierungen. Zur Vermeidung von Systemabhängigkeiten und als einheitliche Basis für diese Betrachtungen wurde in der vorliegenden Arbeit ein *Speicher- und Verhaltensmodell* entwickelt (*Kapitel 5*). Im Rahmen dieses Modells wird ein vereinheitlichtes Schema (das *eInformationsschema*) zur Speicherung von Informationen vorgeschlagen, welche die Eigenschaften und den Zustand von physischen Speicherungsstrukturen beschreiben. Aus den Beschreibungen des Verhaltens von Operationen werden *Kostenfunktionen* zur Ermittlung des bei der Operationsausführung anfallenden I/O-Aufwands abgeleitet. Diese Kostenfunktionen bilden die Basis für die Abschätzung (Vorhersage) des von einer Datenbankreorganisation zu erwartenden Nutzens. Ähnliche Kostenbetrachtungen werden auch im Rahmen der Anfrageoptimierung, als Grundlage der DBMS-Optimizer also, angestellt. Allerdings sind die von den Anfrageoptimierern konkreter DBMS-Produkte verwendeten Kostenmodelle von außen i.d.R. nicht näher dokumentiert bzw. nachvollziehbar. Bei in der Literatur beschriebenen

Kostenmodellen werden Degenerierungen und deren Auswirkungen weitgehend nicht berücksichtigt. Die Kostenmodelle beziehen sich vorwiegend auf den scheinbaren „Normalfall“ nur weniger vorhandener Degenerierungen. Deshalb wurde für die Quantifizierung des Nutzens von Datenbankreorganisationen ein eigenes *I/O-Kostenmodell* entwickelt, mit dem es auch möglich ist, den bei der Ausführung von Operationen durch vorhandene Degenerierungen verursachten Mehraufwand vom normal anfallenden Aufwand zu trennen.

Die *Quantifizierung des von einer Datenbankreorganisation zu erwartenden Nutzens* (*Kapitel 6*) ist, wenn die Auswirkungen auf die Systemleistung betrachtet werden sollen, nur möglich, wenn Informationen über die gegen die Datenbank (oder Teile davon) gerichtete Workload vorliegen. Die manuelle *Erfassung der Workload-Informationen* im hier benötigten Detaillierungsgrad (Datenbankanweisungen) ist bei großen Anwendungssystemen mit vertretbarem Aufwand kaum zu realisieren. Deshalb wird eine Protokollierung der gegen eine Datenbank gerichteten SQL-Anweisungen durch die an der Anfrageverarbeitung beteiligten oder durch aufgesetzt realisierte Komponenten vorgeschlagen. Ausgehend von diesem Anweisungsprotokoll können vom jeweiligen Anfrageoptimierer in einer konkreten Systemumgebung die zur Ausführung der Anweisungen verwendeten Ausführungspläne ermittelt werden. Für die in den Ausführungsplänen enthaltenen Planoperatoren können die Kosten für deren Ausführung vor einer eventuell durchzuführenden Reorganisation abgeschätzt (vorhergesagt) werden. Der in den ermittelten Kosten enthaltene Mehraufwand, der von vorhandenen Degenerierungen verursacht wird, kann dann isoliert und von den Kosten abgezogen werden. Damit können auch die voraussichtlichen (reduzierten) I/O-Kosten zur Workload-Abarbeitung nach einer Reorganisation abgeschätzt werden. Die Differenz der Kostenwerte vor und nach der Reorganisation stellt den von der Datenbankreorganisation zu erwartenden „absoluten“ (Performance-)Nutzen dar.

Neben dem Nutzen müssen bei der Entscheidung über die Durchführung von Datenbankreorganisationen auch die durch sie verursachten Kosten berücksichtigt werden. In die Betrachtungen zur *Ermittlung der Reorganisationskosten* (*Kapitel 7*) wurden ein Offline- und zwei Online-Verfahren einbezogen. Die konkreten Vorgehensweisen der einzelnen Verfahren orientieren sich an den für das DBMS-Produkt Oracle verfügbaren Implementierungen, lassen sich aber zu großen Teilen auch auf andere Systeme übertragen. Beschreibungen von Vorgehensweisen für offline durchzuführende Reorganisationen über Ex- und Import von Daten, Online-Reorganisationen in eine Kopie sowie online durchführbare In-Place-Reorganisationen werden auch in der Literatur ausführlich behandelt. Die Beschreibungen unterscheiden sich zwar teilweise in Details der Implementierung und Umsetzung, die grundlegenden Vorgehensweisen sind aber ähnlich. Neben der Möglichkeit, dem von einer Datenbankreorganisation zu erwartenden Nutzen auch die Kosten für die Reorganisationsdurchführung gegenüberzustellen, konnten durch die Kostenbetrachtungen auch Faktoren identifiziert werden, die den bei der Anwendung der betrachteten Reorganisationsmethoden anfallenden Aufwand u.U. stark beeinflussen können.

Basierend auf der Möglichkeit der näherungsweisen Vorhersage von Kosten und Nutzen von Datenbankreorganisationen wurde ein Ansatz zur *Unterstützung* von Datenbankadministratoren *bei der Auswahl von Reorganisationskandidaten* entwickelt (*Kapitel 8*). Der Ansatz zielt auf die Maximierung des von einer Datenbankreorganisation zu erwartenden Nutzens unter Berücksichtigung einer Kostenobergrenze für die Reorganisation ab.

Prognosen über die Höhe des zu erwartenden Nutzens, besonders bezüglich der Systemleistung, werden von derzeit allgemein verfügbaren Lösungen nicht geboten. Die Quantifizierung von Kosten und Nutzen von Datenbankreorganisationen stellt diesbezüglich eine Erweiterung der Möglichkeiten dar. Die zusätzlichen Informationen dienen zur Unterstützung von Datenbankadministratoren bei der Entscheidung über die Durchführung geplanter Datenbankreorganisationen. Auch zur Unterstützung bei der Auswahl von Reorganisationskandidaten wurde ein Ansatz entwickelt. Mit den entwickelten Konzepten können bspw. Werkzeuge zur Unterstützung der automatisierten Durchführung bestimmter Administrationsaufgaben erweitert und Datenbankadministratoren entlastet werden. Denkbar ist hier eine zyklische Überwachung des Zustands physischer Speicherungsstrukturen und die Durchführung von Nutzen- und Kosten-Analysen. Übersteigt dabei bspw. das Verhältnis von Nutzen und Kosten einen bestimmten Schwellwert, so wird dem DBA ein Vorschlag mit möglichen Reorganisationskandidaten angeboten. Bei der Erstellung dieses Vorschlags kann eine vorab eingestellte obere Grenze für die von der Reorganisation verursachten Kosten berücksichtigt werden.

Eine ähnliche Vorgehensweise wie bei der Quantifizierung des von Datenbankreorganisationen zu erwartenden Nutzens kann auch zur *Bewertung verschiedener physischer Repräsentationen von Datenbankobjekten* angewendet werden (*Kapitel 9*). Vor einer u.U. aufwendigen Konvertierung der die Daten und Zugriffspfadinformationen enthaltenden physischen Speicherungsstrukturen kann damit überprüft werden, ob von der Restrukturierung (Reorganisation auf der Definitionsebene) wirklich Vorteile bezüglich des zur Workload-Abarbeitung anfallenden Aufwands erwartet werden können. Das Verfahren basiert ebenfalls auf einem in der jeweiligen Systemumgebung gewonnenen Anwendungsprotokoll und auf der Ermittlung der bei der Workload-Abarbeitung anfallenden Kosten. Die Betrachtungen für verschiedene Speicherungsstrukturen werden ermöglicht, indem im Datenbankkatalog deren Vorhandensein durch das Erzeugen von entsprechenden Meta- und Statistikdaten simuliert wird.

Ein wesentlicher Vorteil der beschriebenen Methoden zur Quantifizierung des Nutzens von Datenbankreorganisationen und zur Bewertung alternativer physischer Repräsentationen von Datenbankobjekten liegt darin, dass diese *in den konkreten Systemumgebungen durchgeführt* werden können, für die die Betrachtungen angestellt werden. Damit können auch dort vorhandene spezielle Gegebenheiten weitgehend berücksichtigt werden. Anhand von prototypischen Implementierungen und durchgeführten Messreihen wurde die Umsetzbarkeit der im Rahmen der vorliegenden Arbeit entwickelten Vorgehensweisen überprüft.

10.3 Anknüpfungspunkte für weitere Arbeiten

Die in der vorliegenden Arbeit behandelten Themenstellungen bieten einige Anknüpfungspunkte für weiterführende Arbeiten.

Prototypische Implementierungen wurden im Rahmen der Arbeit größtenteils zur Überprüfung der entwickelten Methoden und Vorgehensweisen vorgenommen. Für einen möglichen praktischen Einsatz müssten die entstandenen Teillösungen zusammengeführt werden. Damit eine solche Lösung auch bei verschiedenen DBMS-Produkten angewendet werden kann, müssen für die jeweiligen Produkte Transformationsschichten implementiert werden, die zumindest eine Transformation und Übertragung von Daten aus dem jeweiligen Systemkatalog in das eInformationsschema realisiert.

Die durchgeführten Messreihen zur Ermittlung der Kosten von Datenbankreorganisationen haben gezeigt, dass mit den in Kapitel 7 beschriebenen Vorgehensweisen vorab Kostenschätzungen zur groben Orientierung durchgeführt werden können. Allerdings sind hier noch weitere Untersuchungen zur Verbesserung der Genauigkeit notwendig. Von Interesse sind dabei besonders die genauere Abschätzung des im Rahmen asynchron durchgeführter Schreiboperationen anfallenden Aufwands sowie Verbesserungen bei der Berücksichtigung des Aufwands für die Erzeugung von temporären Log-Einträgen. Weiterhin sind hier Vorgehensweisen zur Sicherung konsistenter Suchergebnisse im Vorfeld von Update-Operationen zu nennen. Die Ergebnisse solcher Untersuchungen können auch für genauere Kostenfunktionen für Änderungsoperationen (wie in *Abschnitt 5.4.2* beschrieben) genutzt werden.

Das Speicher- und das Verhaltensmodell beinhalten derzeit verbreitete Strukturen und Operationen. Allerdings wurden und werden DBMS-Produkte im Zuge der Entwicklung ständig erweitert. So werden bspw. Bitmap-Indexe oder spezielle Strukturen zur Speicherung und Indexierung (raumbezogener) geographischer Daten [Sto06] von immer mehr Produkten unterstützt. Gleiches gilt für die Unterstützung von XML-Daten. Die Erweiterung und Vervollständigung des Speicher- und Verhaltensmodells stellt einen Anknüpfungspunkt für weiterführende Arbeiten dar.

Auch die Unterstützung von komplexen Objekten in objektrelationalen Datenbank-Management-Systemen bietet eine Reihe von Anknüpfungspunkten für weitere Arbeiten. In [Luf05] werden Erweiterungen des SQL-Sprachumfangs zur Unterstützung komplexer Datenstrukturen vorgeschlagen. Physische Speicherungsstrukturen und Indexierungsmöglichkeiten für komplexe Objekte werden in [Ska06] untersucht, und es wird eine Spracherweiterung vorgeschlagen, die es erlaubt, auch die Abbildung der komplexen Objekte auf physische Speicherungsstrukturen festzulegen. Solche Erweiterungen bieten zwar prinzipiell mehr Spielraum bei der Beeinflussung der logischen und physischen Strukturierung von Datenbankobjekten, allerdings wird auch der Prozess der Datenmodellierung durch die wachsende Vielfalt an logischen und physischen Strukturen auf allen Ebenen komplexer und aufwendiger. Besonders bei der physischen Modellierung

komplexer Objekte muss die gegen die Objekte gerichtete Workload berücksichtigt werden. Die vorgestellte Methode zur Bewertung alternativer physischer Repräsentationen komplexer Objekte zeigt einen prinzipiellen Weg auf, wie hier eine Unterstützung geboten werden kann. Die Betrachtungen wurden an einem einfachen Beispiel vorgenommen. Wie die Methode gestaltet werden muss, damit bspw. auch über mehrere Stufen geschachtelte Objekte bewertet werden können, wird in der vorliegenden Arbeit nicht behandelt und könnte Gegenstand zukünftiger Untersuchungen sein.

Verfahren zur Konvertierung physischer Speicherungsstrukturen (also zur Reorganisation auf der Definitionsebene) wurden im Rahmen der vorliegenden Arbeit ebenfalls nicht eingehender untersucht. Inwieweit Prüfungen der Plausibilität angestrebter physischer Repräsentationen komplexer Objekte vor Reorganisationsbeginn möglich und sinnvoll sind, könnte ebenfalls im Rahmen weiterführender Arbeiten untersucht und damit die Unterstützung von Datenbankadministratoren verbessert werden.

Anhang A – Elemente des Speichermodells des eInformations-schemas

Die auf den folgenden Seiten dieses Anhangs angegebenen Datentabellen beschreiben die einzelnen Elemente des Speichermodells des eInformationsschemas. Das Tabellenschema wurde in Referenzimplementierungen für das DBMS-Produkt Oracle [Hel01, Wil04] umgesetzt. Neben Namen, Datentyp und kurzen beschreibenden Informationen zum Inhalt der jeweiligen Attribute, wird bei den Größen, die zur Quantifizierung von Nutzen und Kosten von Datenbankreorganisationen benötigt werden, ein Formelzeichen angegeben.

Teilweise werden im eInformationsschema Daten auch redundant gehalten. So wird beispielsweise die Anzahl Tupel (Sätze) als Eigenschaft von Partitionen aufgefasst, um auch Analysen auf Partitionen bezogen durchführen zu können. Die Anzahl Tupel wird aber auch als Eigenschaft von Tabellen angesehen und dort gespeichert, obwohl sie i.d.R. die Summe der Anzahl Tupel aller Partitionen der Tabelle darstellt. Allerdings können dabei auch scheinbare Unstimmigkeiten auftreten. Werden beispielsweise die Daten von zwei Tabellen in einem Table Cluster gespeichert, der in einer Partition (Segment) untergebracht wird, so enthält die Partition als Eigenschaft die Summe der Anzahlen der Tupel aus beiden Tabellen. Wie viele davon die einzelnen Tabellen enthalten geht daraus nicht hervor. Deshalb werden diese Informationen bei den Tabellen festgehalten.

Die Tabelle DATENBANK enthält Informationen, die systemweit Gültigkeit besitzen.

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
db_id	numerisch	Datenbankidentifikator	-	
db_name	Zeichenkette	Name der Datenbank	-	SQL-Norm: INFORMATION_ SCHEMA_CATALOG_NAME. CATALOG_NAME
db_ps	numerisch	Blockgröße (in Byte)	DB_{PS}	
db_vdp	numerisch	Länge der Verwaltungsinformation je Datenblock (in Byte)	DB_{VDP}	z.B. Block Header
db_vip	numerisch	Länge der Verwaltungsinformation je Indexblock (in Byte)	DB_{VIP}	z.B. Block Header
db_lvt	numerisch	Länge der Verwaltungsinformation je Table Space (in Blöcken)	DB_{LVT}	
db_lvp	numerisch	Länge der Verwaltungsinformation je Partition (in Blöcken)	DB_{LVP}	
db_lvs	numerisch	Länge der Verwaltungsinformation je Satz (in Byte)	DB_{LVS}	Verwaltungsinformationen je Satz, die systemweit gleich sind, wie z.B. Slot-Eintrag
db_lvk	numerisch	Länge der Verwaltungsinformation je Schlüsselwert eines Index (in Byte)	DB_{LVK}	Manche Systeme verwenden auch hier (wg. der variablen Länge der SEinträge) Slot-Einträge.
db_lba	numerisch	Länge einer Blockadresse (in Byte)	DB_{LBA}	
db_ltid	numerisch	Länge eines Tupelidentifikators (in Byte)	DB_{LTID}	Als Tupelidentifikatoren werden hier Verweise auf Speicherorte (z.B. TIDs bzw. ROWIDs zum Verweis auf Sätze) bezeichnet.
db_pre	numerisch	Prefetch-Rate (in Blöcken)	DB_{PRE}	gibt die Anzahl Blöcke an, die bei sequenziellem Lesen mit einer I/O-Operation von Datenträgern gelesen werden
db_bw	numerisch	Blockungsfaktor beim Schreiben	DB_{BW}	gibt die Anzahl Blöcke an, die bspw. von Werkzeugen zum Datenimport mit einer I/O-Operation geschrieben werden können
db_llogd	numerisch	Länge des festen Teils eines Log-Eintrags zu Änderungen an Datenblöcken (in Byte)	DB_{LLOGD}	
db_llogi	numerisch	Länge des festen Teils eines Log-Eintrags zu Änderungen an Indexblöcken (in Byte)	DB_{LLOGI}	

Tabelle A.1 Beschreibungsinformationen von Datenbanken

Die Tabelle RELATIONEN enthält Informationen zu den Tabellen, die unabhängig von deren jeweiliger Organisationsform sind. Die Information, ob es sich bei einer Tabelle bspw. um eine indexorganisierte Tabelle handelt, kann aus dem Typ des Index abgelesen werden, über den die IOT organisiert ist.

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
t_id	numerisch	Identifikator der Tabelle	-	
t_name	Zeichenkette	Name der Tabelle	-	SQL-Norm: TABLES.TABLE_NAME
t_dbid	numerisch	Datenbankidentifikator	-	Zuordnung zu einer Datenbank (Fremdschlüssel) evtl. auch aus SQL-Norm: TABLES.TABLE_CATALOG, TABLES.TABLE_SCHEMA
t_cid	numerisch	Identifikator des Table Clusters	-	Fremdschlüssel
t_len	numerisch	durchschnittliche Satzlänge (in Byte)	T_{LEN}	Information wird bei vielen DBMS über Statistikkomponenten ermittelt und ermöglicht auf einfache Weise genaue Abschätzungen des Speicherbedarfs einer Tabelle
t_anzt	numerisch	Tupelzahl	T_{ANZT}	Information wird bei vielen DBMS über Statistikkomponenten ermittelt und erspart u.U. die Aufsummierung der Werte der einzelnen Partitionen.
t_ovfl	numerisch	aktuelle Anzahl ausgelagerter Sätze	T_{OVFL}	Information wird bei vielen DBMS über Statistikkomponenten ermittelt und erspart u.U. die Aufsummierung der Werte der einzelnen Partitionen.
t_f1	numerisch	Füllungsgrad 1 (in %)	T_{F1}	Bis zu diesem Grad wird ein Block bei Einfügeoperationen gefüllt. Der Rest wird für satzverlängernde Updates reserviert.
t_f2	numerisch	Füllungsgrad 2 (in %)	T_{F2}	Nach Überschreitung von T_{F1} muss der Block bis zum Grad T_{F2} geleert werden, bevor wieder Sätze in den Block eingefügt werden können.
t_anzp	numerisch	Anzahl Partitionen der Tabelle	T_{ANZP}	bei nicht partitionierten Tabellen = 1
t_fext	numerisch	Größe First Extent (in Blöcken)	T_{FEXT}	Größe des ersten Extent beim Anlegen der Tabelle
t_next	numerisch	Größe Next Extents (in Blöcken)	T_{NEXT}	wird bei weiteren zu reservierenden Extents verwendet

Tabelle A.2: Beschreibungsinformationen von Tabellen

Die Tabelle FELDER enthält Informationen über die in Tabellen enthaltenen Spalten (Attribute).

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
a_id	numerisch	Identifikator des Felds	-	
a_name	Zeichenkette	Name des Felds	-	SQL-Norm: COLUMNS.COLUMN_NAME
a_tid	numerisch	Identifikator der zugehörigen Tabelle	-	Fremdschlüssel evtl. auch aus SQL-Norm: COLUMNS.TABLE_CATALOG, COLUMNS.TABLE_SCHEMA, COLUMNS.TABLE_NAME
a_len	numerisch	Speicherbedarf (in Byte)	A_{LEN}	Für Datentypen variabler Länge wird der durchschnittliche Speicherbedarf angegeben. evtl. auch ermittelbar aus SQL-Norm: COLUMNS.DATA_TYPE
a_plen	numerisch	Blockgröße bei LOBs (in Byte)	A_{PLEN}	wird nur verwendet, wenn es sich beim entspr. Feld um ein LOB-Feld handelt
a_lva	numerisch	Länge der Verwaltungsinformation (in Byte)	A_{LVA}	Länge der Verwaltungsinformation, die für das Feld benötigt wird (z.B. für die aktuelle Länge)

Tabelle A.3: Beschreibungsinformationen von Feldern

Eine Sonderrolle spielen Felder (Attribute) mit Datentypen für *Large Objects* (LOB). Für diese Datentypen wird in der Tabelle physisch meist lediglich ein Deskriptor fester Länge abgelegt, der u.a. auch einen Verweis auf den Speicherort des LOB enthält. Weiterhin ist es möglich, für LOB-Felder jeweils individuelle Blockgrößen festzulegen, abweichend von der sonst verwendeten Blockgröße.

In der Tabelle INDEXE werden verschiedene Index-Typen unterschieden.

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
i_id	numerisch	Identifikator des Index	-	
i_name	Zeichenkette	Name des Index	-	
i_type	Zeichen	Index-Typ	-	B- "normaler" B*-Baum (Heap-Tabelle) C- Cluster-Index H- Hash I- indexorganisierte Tabelle S- Sekundärindex IOT
i_levels	numerisch	aktuelle Anzahl Ebenen im Index	I_{LEV}	bei Typ B,C,I und S
i_leafes	numerisch	aktuelle Anzahl Blöcke auf der Blattebene	I_{LEAF}	bei Typ B,C,I und I
i_clust	numerisch	Clustering Factor	I_{CLUST}	normalisiert, bei Typ B und S
i_unique	numerisch	aktuelle Anzahl unterschiedlicher Schlüsselwerte	I_{UNQ}	bei Typ B,C,I und S
i_min	numerisch	kleinster Schlüsselwert	I_{MIN}	
i_max	numerisch	größter Schlüsselwert	I_{MAX}	
i_hkeys	numerisch	Anzahl Hash-Werte |H|	I_{HKEYS}	bei Typ H
i_chains	numerisch	Anzahl Überlaufblöcke	I_{CHAINS}	bei Typ I und H
i_dir_acc	numerisch	Anzahl korrekter Zugriffe über Blockverweis in UROWID (in %)	I_{DIR_ACC}	bei Typ S
i_fill	numerisch	Füllungsgrad (in %)	I_{FILL}	Bis zu diesem Grad wird ein Block beim Bottom-Up-Aufbau gefüllt.
i_anzp	numerisch	Anzahl Partitionen des Index	I_{ANZP}	bei nicht partitionierten Indexen = 1
i_fext	numerisch	Größe First Extent (in Blöcken)	I_{FEXT}	Größe des ersten Extent beim Anlegen der Tabelle
i_next	numerisch	Größe Next Extents (in Blöcken)	I_{NEXT}	wird bei weiteren zu reservierenden Extents verwendet

Tabelle A.4: Beschreibungsinformationen von Indexen

Abhängig vom Typ eines Index werden nur bestimmte Informationen festgehalten. So werden bei indexorganisierten Tabellen die Festlegungen bezüglich der Extent-Größen verwendet, die zur zugehörigen Tabelle in RELATIONEN hinterlegt sind. Die Füllungsgrade, die dort ebenfalls hinterlegt sind, werden ignoriert. Hier wird der bei den Daten des Index gespeicherte Wert verwendet.

Für B-Baum-Indexe wird die Wahrscheinlichkeit, dass sich ein Block, auf den zugegriffen werden soll, im Puffer befindet (Pufferungsgrad), bezogen auf die einzelnen Ebenen des Index in der Tabelle INDEX_PUFFERUNG festgehalten.

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
ip_iid	numerisch	Identifikator des Index	-	Fremdschlüssel
ip_lev	numerisch	Ebene des Index	-	mit 1 beim Wurzelknoten beginnend
ip_br	numerisch	Pufferungsgrad (in %)	I_{BR}	

Tabelle A.5: Speicherung der ebenenbezogenen Pufferungsgrade von B-Baum-Indexen

Informationen über Table Cluster enthält die Tabelle `TAB_CLUSTER`. Im einfachsten Fall wird in einem Table Cluster nur eine Tabelle abgelegt. Eine tabellenübergreifende Clusterung findet nicht statt und ein Cluster-Index ist nicht notwendig. Bei tabellenübergreifender Clusterung muss ein entsprechender Eintrag in der Tabelle `INDEXE` existieren.

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
c_id	numerisch	Identifikator des Table Clusters	-	
c_type	Zeichen	Cluster-Typ	-	`H` - Hash-Cluster `I` - Cluster über B*-Baum `N` - normale Tabelle
c_bsize	numerisch	Bucket-Größe (in Byte)	C_{BSIZE}	Die Information wird nicht bei Typ `N` genutzt.
c_iid	numerisch	Identifikator des Cluster-Index	-	Fremdschlüssel Die Information wird nicht bei Typ `N` benutzt.

Tabelle A.6: Beschreibungsinformationen von Table Clustern

PARTITIONEN enthält Informationen über die physischen Speicherungsstrukturen von Partitionen (Segmente). Derzeit werden keine (logischen) Informationen zum Partitionierungsschema gespeichert.

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
p_id	numerisch	Identifikator der Partition	-	
p_cid	numerisch	Identifikator des zugehörigen Table Clusters	-	Fremdschlüssel
p_tsid	numerisch	Identifikator des Table Space	-	Fremdschlüssel
p_anzt	numerisch	aktuelle Anzahl Sätze in der Partition	P_{ANZT}	zur Überprüfung, ob eine erwartete Verteilung der Sätze eingehalten wurde
p_splt	numerisch	Anzahl der Zeiger, die durch Satzsplitting entstanden sind	P_{SPLT}	Muss ein Satz auf mehrere Blöcke verteilt werden, so werden die Teile miteinander verkettet.
p_ovfl	numerisch	aktuelle Anzahl ausgelagerter Sätze	P_{OVFL}	
p_f1	numerisch	Füllungsgrad 1 (in %)	P_{F1}	Bis zu diesem Grad wird ein Block bei Einfügeoperationen gefüllt. Der Rest wird für satzverlängernde Updates reserviert.
p_f2	numerisch	Füllungsgrad 2 (in %)	P_{F2}	Nach Überschreitung von P_{F1} muss der Block bis zum Grad P_{F2} geleert werden, bevor wieder Sätze eingefügt werden können.
p_used	numerisch	Anzahl der aktuell genutzten Blöcke	P_{USED}	Information wird bei vielen DBMS über Statistikkomponenten ermittelt und erspart u.U. die Aufsummierung der Werte der einzelnen Extents.
p_fext	numerisch	Größe First Extent (in Blöcken)	P_{FEXT}	Verwendung beim Anlegen der Partition
p_next	numerisch	Größe Next Extents (in Blöcken)	P_{NEXT}	wird bei weiteren zu reservierenden Extents verwendet
p_br	numerisch	Pufferungsgrad (in %)	P_{BR}	wird für Partitionen verwendet, die Daten enthalten
p_brs	numerisch	Pufferungsgrad für sequenzielles Suchen (in %)	P_{BRS}	Der Wert unterscheidet sich von P_{BR}, wenn das DBMS unterschiedliche Seitenersetzungsstrategien für seq. Suchen und Zugriffe über Indexe verwendet.

Tabelle A.7: Beschreibungsinformationen von Partitionen

Die Tabelle IND_PART enthält Informationen über die Beziehungen zwischen Indexen und den Partitionen/Segmenten, in denen sie gespeichert werden.

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
ip_pid	numerisch	Identifikator der Partition	-	Fremdschlüssel
ip_iid	numerisch	Identifikator des Index	-	Fremdschlüssel
ip_anzk	numerisch	Anzahl der Schlüsselwerte in dem in dieser Partition gespeicherten (Teil-) Index	IP_{ANZK}	

Tabelle A.8: Beziehung zwischen Indexen und Partitionen

Die Informationen über die (N:M)-Beziehungen zwischen Indexen und den zugehörigen Feldern werden in der Tabelle FEL_IND gespeichert.

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
ai_aid	numerisch	Identifikator des Felds	-	Fremdschlüssel
ai_iid	numerisch	Identifikator des Index	-	Fremdschlüssel
ai_pos	numerisch	Position des Felds im Index	AI_{POS}	

Tabelle A.9: Beziehung zwischen Attributen und Indexen

Die Informationen über Table Spaces werden in der Tabelle TABLESPACES hinterlegt.

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
ts_id	numerisch	Identifikator des Table Space	-	
ts_name	Zeichenkette	Name des Table Space	-	
ts_esize	numerisch	Standardgröße von Extents (in Blöcken)	TS_{ESIZE}	Die Größe wird verwendet, wenn keine tabellen- bzw. parttionsspezifischen Angaben vorliegen.
ts_zbv	numerisch	Speicher für besondere Verwendung (in Blöcken)	TS_{ZBV}	Wenn z.B. im Table Space das Log (oder ein Teil davon) gespeichert wird, wird hier der dafür verwendete Speicherplatz erfasst.

Tabelle A.10: Beschreibungsinformationen von Table Spaces

Dateien (Volumes) sind Speichereinheiten, die dem DBMS durch den DBA zur Verfügung gestellt werden. Beschreibungsinformationen werden in der Tabelle VOLUMES gehalten. Dabei wird davon ausgegangen, dass der Speicher der Datei, z.B. durch eine Reorganisation auf Dateisystemebene, bei Bedarf vorher defragmentiert wurde. Im Modell wird nicht unterschieden, ob zwei (oder mehrere) Dateien auf

einem physischen Datenträger (z.B. in Form von Dateien) abgelegt werden oder ob je Datei ein eigener physischer Datenträger verwendet wird, obwohl dies sicherlich Auswirkungen auf die Systemleistung haben kann. Probleme, die sich aufgrund einer ungünstigen Verteilung der Dateien ergeben, z.B. Leistungseinbußen durch konkurrierende Zugriffe, wenn mehrere Dateien auf einem physischen Datenträger angelegt wurden oder durch (externe) Fragmentierungen der Dateien, liegen außerhalb der Verantwortung des DBMS und stellen somit keine Degenerierung in unserem Sinne dar. Für sehr genaue Abschätzungen wäre hier eventuell noch eine Erweiterung um Leistungsmerkmale (Zugriffszeiten etc.) der physischen Datenträger, auf denen die jeweiligen Dateien abgelegt werden, denkbar.

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
v_id	numerisch	Identifikator der Datei	-	
v_sip	numerisch	Größe der Datei	V_{SIP}	Angabe in Blöcken
v_tsid	numerisch	Identifikator des Table Space	-	Fremdschlüssel

Tabelle A.11: Beschreibungsinformationen von Dateien

Mit dem in der Tabelle EXTENTS gespeicherten Wert E_{PUSE} sind jetzt alle Werte eingeführt, um den Clustering Factor von Indexen in normalisierter Form zu berechnen. Neben E_{PUSE} wird für die Berechnung des Clustering Factor eines Index noch die Zahl der in den Partitionen gespeicherten Sätze (P_{ANZT}), auf die die entsprechenden Indexeinträge verweisen und die Anzahl (N_{BW}), wie oft bei einem Index Scan der Block wechselt, auf den zugegriffen werden muss, benötigt. Der Wert von N_{BW} wird bspw. bei DB2 und Oracle in den Statistikdaten der Indexe als Clustering Factor geführt. *Gleichung A.1* wird zur Berechnung des Clustering Factor verwendet.

$$I_{CLUST} = \left(1 - \frac{N_{BW} - \sum_{\forall Extents} E_{PUSE}}{\sum_{\forall Partitionen} P_{ANZT} - \sum_{\forall Extents} E_{PUSE}} \right) \cdot 100$$

Gleichung A.1

Der Clustering Factor ist eine Prozentzahl, die die Einhaltung einer internen Sortierreihenfolge der Tupel einer Tabelle nach dem Sortierkriterium des jeweiligen Index beschreibt. Im besten Fall (vollständige Einhaltung der Sortierreihenfolge) entspricht die Anzahl notwendiger Blockwechsel der Anzahl Blöcke, die die Tabelle belegt. Dieser unvermeidliche ”Grundaufwand” wird durch den Abzug der Gesamtzahl Blöcke, die von der Tabelle belegt werden, berücksichtigt. Damit wird auch eine durch schlechte Speicherausnutzung erhöhte Anzahl Blockwechsel relativiert. Der Clustering Factor kann zur Bestimmung des Mehraufwands genutzt werden, der durch die Nichteinhaltung interner Sortierreihenfolgen bei der Ausführung von Index Scans auftritt.

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
e_id	numerisch	Identifikator des Extent	-	
e_type	Zeichen	Typ des Extents	-	F → First Extent N → Next Extent
e_pid	numerisch	Identifikator der zugehörigen Partition	-	Fremdschlüssel
e_vid	numerisch	Identifikator des Volume	-	Fremdschlüssel
e_size	numerisch	Größe des Extent (in Blöcken)	E_{SIZE}	
e_puse	numerisch	Anzahl belegter Blöcke	E_{PUSE}	
e_lfdnr	numerisch	lfd. Nummer des Extent innerhalb der Partition	-	Zur Erkennung von Fragmentierungen ist die log. Reihenfolge der Extents interessant.
e_adr	Adresse	Anfangsadresse des Extent	-	Dient zur Erkennung, ob sich der Extent auch physisch an den log. vorherigen Extent anschließt.

Tabelle A.12: Beschreibungsinformationen von Extents

Die Beschreibungsinformationen zu Blöcken werden hier der Vollständigkeit halber aufgeführt. Informationen für die Tabelle BLOECKE werden derzeit im eInformationsschema aus Aufwandsgründen nicht ermittelt und festgehalten.

Attribut	Typ	Information	Formel-zeichen	Bemerkungen
b_id	numerisch	Identifikator des Blocks	-	z.B. Blocknummer
b_eid	numerisch	Identifikator des Extent	-	Fremdschlüssel
b_fsp	numerisch	freier Platz innerhalb des Blocks (in Byte)	B_{FSP}	Freiplatz kann in Fragmenten vorliegen. Diese werden allerdings bei Bedarf zusammengeführt.

Tabelle A.13: Beschreibungsinformationen von Blöcken

Anhang B – Speicherplatzbedarfsabschätzungen

Speicherplatzabschätzung für Datenbereiche von Heap-Tabellen

Bei Tabellen mit Sätzen fester Länge ist es relativ einfach, den (minimal benötigten) Speicherbedarf der Datenbereiche abzuschätzen, wenn die Anzahl der Tupel der Tabelle bekannt ist. Hier ist zu beachten, dass der Speicher für Tabellen in Extents reserviert wird. Das heißt, dass der beim Anlegen einer Tabelle tatsächlich reservierte Speicherbereich größer sein wird, als der geschätzte Wert. Von einer „schlechten“ Speicherausnutzung kann frühestens gesprochen werden, wenn sich die Anzahl der aktuell reservierten Extents um mehr als eins von der errechneten Anzahl notwendiger Extents unterscheidet. Ferner muss berücksichtigt werden, dass verbreitete DBMS-Produkte aus Performance-Gründen eine Verteilung von Tupeln auf mehrere Blöcke nur dann vornehmen, wenn die Satzlänge die Größe des in einem Block für Daten maximal verfügbaren Speichers überschreitet. Die evtl. daraus resultierende schlechtere Speicherplatzauslastung wird toleriert. Sollte allerdings eine Aufteilung der Sätze notwendig sein, so wird i.d.R. versucht, diese auf möglichst wenige Blöcke zu verteilen. Auch hier wird toleriert, dass der letzte Block oftmals nicht vollständig ausgenutzt wird. Bei der Verwendung von Datentypen variabler Länge (z.B. `VARCHAR`, `TEXT`, `BYTE` usw.) muss mit Durchschnittswerten bzw. mit geschätzten Durchschnittswerten für einzelne Feldlängen gearbeitet werden. Dabei hat natürlich die Genauigkeit der Schätzung Einfluss auf die Genauigkeit des Gesamtergebnisses.

Der Speicher, der insgesamt für eine Tabelle benötigt wird, beinhaltet die Bereiche für die eigentlichen Daten und die Bereiche für die Abspeicherung von Indexen. Dabei können Daten- und Indexblöcke gemischt abgespeichert werden, wie z.B. bei Informix [IBM05a] oder getrennt in unterschiedlichen Bereichen (Segmenten), wie z.B. bei Oracle [Ora03b]. Die Form der Speicherung muss berücksichtigt werden, wenn die Ermittlung der benötigten Extents erfolgt. Für Tabellen, bei denen die Länge der Datensätze kleiner als die Blockgröße ist, kann der für den Datenbereich mindestens benötigte Speicher nach dem folgenden Schema näherungsweise berechnet werden.

Zunächst erfolgt die Berechnung der Satzlänge (*LT - Gleichung B.1*) inklusive der Verwaltungsinformationen (A_{LVA}) für die einzelnen Felder.

$$LT = \sum_{\forall Felder\ der\ Sätze} \left(A_{LEN} + A_{LVA} \right) \qquad \text{Gleichung B.1}$$

Anschließend kann die Anzahl Sätze je Block (*TP*) berechnet werden (*Gleichung B.2*).

$$TP = \left\lfloor \frac{\overbrace{\left(DB_{PS} - DB_{VDP} \right)}^{verfügb.\ Teil\ im\ Datenblock} \cdot \overbrace{\left(T_{F1} / 100 \right)}^{Füllungsgrad}}{\underbrace{LT + DB_{LVS}}_{Satzlänge\ inkl.\ Verwaltungsdaten\ je\ Satz}} \right\rfloor \qquad \text{Gleichung B.2}$$

Im eInformationsschema wird die aktuell vorhandene Anzahl Sätze als Eigenschaft von Partitionen aufgefasst. Ist eine Tabelle in mehrere Partitionen unterteilt, so kann die Satzanzahl für die gesamte Tabelle (*T*) entweder aus der Summe der in allen Partitionen gespeicherten Sätze (P_{ANZT}) berechnet (*Gleichung B.3*) oder aus den Beschreibungsinformationen der jeweiligen Tabelle (T_{ANZT}) abgelesen werden.

$$T = \sum_{\forall\ Patritionen} P_{ANZT}$$ Gleichung B.3

Anschließend kann die Anzahl der zur Datenspeicherung benötigten Blöcke (*P*) ermittelt werden (*Gleichung B.4*). Dabei müssen die Verwaltungsblöcke der einzelnen Partitionen der Tabelle berücksichtigt werden.

$$P_{IDEAL} = \left\lceil \frac{T}{TP} \right\rceil + \underbrace{DB_{LVP} \cdot T_{ANZP}}_{Verwaltungsinformationen}$$ Gleichung B.4

Sollen die Standardvorgaben des jeweiligen Table Space (TS_{ESIZE})verwendet werden, in dem die Tabelle abgelegt wird, kann jetzt die Anzahl Extents berechnet werden (*NE - Gleichung B.5*). An dieser Stelle ist zu beachten, dass die Umrechnung in Extents, wenn das DBMS Daten- und Indexblöcke in den Extents gemischt ablegt, erst nach der Berechnung des für Tabellen und Indexe benötigten Speichers erfolgen kann, weil der von Indexen benötigte Speicher in die Berechnung der Anzahl Blöcke (*P*) mit einbezogen werden muss.

$$NE = \left\lceil \frac{P}{TS_{ESIZE}} \right\rceil$$ Gleichung B.5

Werden tabellenspezifische Extent-Größen verwendet, so wird davon ausgegangen, dass beim Anlegen einer Tabelle die jeweiligen First-Extents der Partitionen so groß dimensioniert werden, dass die aktuell vorhandenen Sätze darin untergebracht werden können. Die Anzahl der Extents entspricht dann der Anzahl der Partitionen einer Tabelle. Unter der Annahme, dass die Daten gleichmäßig auf die Partitionen verteilt werden, kann die Extent-Größe über *Gleichung B.6* berechnet werden.

$$T_{FEXT} = \left\lceil \frac{P}{T_{ANZP}} \right\rceil$$ Gleichung B.6

Ist die Satzlänge größer als der in einem Block maximal für Daten zur Verfügung stehende Speicher, so kann die Berechnung der Anzahl der zur Datenspeicherung benötigten Blöcke mit *Gleichung B.7* durchgeführt werden. Der im letzten von einem Satz belegten Block verbleibende Freiplatz wird dabei nicht für Teile anderer Sätze benutzt.

$$P_{IDEAL} = \left\lceil \frac{\overbrace{\left(LT + DB_{LVS}\right)}^{Satzlänge\ inkl.\ Verwaltungsdaten}}{\left(\underbrace{\left(DB_{PS} - DB_{VDP}\right)}_{verfügb.\ Teil\ im\ Datenblock} \cdot \underbrace{\left(T_{F1} / 100\right)}_{Füllfaktor} \right)} \right\rceil \cdot T + \underbrace{DB_{LVP} \cdot T_{ANZP}}_{Vrewaltungsblöcke\ Partition}$$ Gleichung B.7

Speicherplatzabschätzung für B*-Baum-Indexe

Die Berechnung der Anzahl Blöcke, die für einen B-Baum-Index nach einem Bottom-Up-Aufbau benötigt werden, ist mit ausreichender Genauigkeit möglich. Der errechnete Wert kann dann mit der Anzahl tatsächlich belegter Indexblöcke verglichen werden. Zu beachten ist aber, dass die errechnete Blockanzahl bei durch Einfüge- und Löschoperationen „gewachsenen“ B-Bäumen mit Ungenauigkeiten behaftet ist, da sie von der Verteilung der Schlüsselwerte und der Reihenfolge ihres Auftretens abhängig ist [Ham83, Küs83]. B-Baum-Indexe, die im Rahmen von Reorganisationsbedarfsanalysen betrachtet werden, sind i.d.R. „gewachsene“ Bäume. Für diese muss u.U. auch eine Ermittlung des aktuell belegten Speichers erfolgen. Im eInformationsschema wird für eine Partition (ein Segment) die Anzahl aktuell belegter Blöcke gespeichert. Werden Indexe und Daten in getrennten Segmenten abgelegt, so kann die Zahl der von einem Index aktuell belegten Blöcke einfach aus den Statistikdaten des Indexsegments abgelesen werden. Werden Daten- und Indexblöcke gemischt gespeichert, so ist dies nur dann möglich, wenn bei den Statistikdaten der entsprechenden Tabelle die Anzahl der mit Daten belegten Blöcke geführt wird. Sonst kann die Anzahl der von einem Index aktuell belegten Blöcke nur rechnerisch (mit Ungenauigkeiten behaftet) ermittelt werden, da im eInformationsschema (in Anlehnung an gängige DBMS-Produkte) nur Informationen über die Anzahl Ebenen im Baum und die Anzahl Blöcke auf der Blattebene geführt werden.

Bei der Berechnung des Speichers (LI_{ideal}), der nach einem Bottom-Up-Aufbau belegt wird, wird auch ein evtl. vorgegebener Füllungsgrad der Indexblöcke berücksichtigt. Zunächst wird die durchschnittliche Schlüssellänge (*LK* – einschließlich der Längeninformationen für die im Schlüssel enthaltenen Felder) errechnet (*Gleichung B.8*).

$$LK = \sum_{\forall \text{Felder des Schlüssels}} \left(A_{LEN} + A_{LVA} \right)$$

Gleichung B.8

Anschließend kann die durchschnittliche Länge eines Indexeintrags (*LE* - *Gleichung B.9*) berechnet werden. Dabei muss berücksichtigt werden, dass ein Eintrag bei nicht eindeutigen Indexen auf mehrere Sätze verweisen kann. Hinzu kommt u.U. noch die Länge einer Verwaltungsinformation (DB_{LVK}).

$$LE = \underbrace{LK + DB_{LVK}}_{\text{Schlüssel und Verwaltungsdaten}} + \left\lceil \frac{T}{\underbrace{I_{UNQ}}_{\text{Verweise je Schlüssel}}} \cdot \underbrace{DB_{LTID}}_{\text{Länge Verweis (z.B. TID, RID)}} \right\rceil$$

Gleichung B.9

Ist die Länge eines Eintrags auf der Blattebene bekannt, so kann die Anzahl Einträge je Knoten in der Blattebene (*EP*) mit *Gleichung B.10* berechnet werden.

$$EP = \left\lfloor \frac{\overbrace{\left(DB_{PS} - DB_{VIP} - 2 \cdot DB_{LBA} \right)}^{\text{Block abzügl. Verwaltungsdaten und Zeiger für Verkettung}} \cdot \overbrace{\left(I_{FILL} / 100 \right)}^{\text{Füllungsgrad}}}{LE} \right\rfloor$$

Gleichung B.10

Für die Berechnung der Anzahl Blöcke auf der Blattebene (NP_B – *Gleichung B.11*) wird die Anzahl unterschiedlicher Schlüsselwerte (I_{UNQ}) benötigt.

$$NP_B = \left\lceil \frac{I_{UNQ}}{EP} \right\rceil$$ Gleichung B.11

Die Berechnung der Anzahl Blöcke der übrigen Ebenen (NP_i - wobei i die Ebene bezeichnet) einschließlich der Wurzel kann, von der Blattebene ausgehend nach oben, jeweils über die Anzahl Blöcke der nächsttieferen Ebene (NP_{i+1}) und die Anzahl Einträge je Knoten in den inneren Ebenen erfolgen. Die Anzahl Verweise (*NL – Gleichung B.12*) auf Knoten der nächsttieferen Ebene ist dabei (wie bei B-Bäumen üblich) um eins höher als die Anzahl Einträge, die in einem Knoten Platz finden. Die Länge des für den zusätzlichen Verweis benötigten Speicherplatzes wird über die Größe DB_{LBA} berücksichtigt.

$$NL_{IDEAL} = \left\lfloor \frac{\overbrace{\left(DB_{PS} - DB_{VIP}\right)}^{\text{Block abzügl. Verwaltungsdaten}} \cdot \overbrace{\left(I_{FILL} / 100\right)}^{\text{Füllungsgrad}} - DB_{LBA}}{\underbrace{LK + DB_{LVK} + DB_{LBA}}_{\text{Bruttogröße eines Eintrags}}} \right\rfloor + 1$$ Gleichung B.12

Jetzt kann die minimale Anzahl Ebenen des Baums ($I_{LEVideal}$) nach der Reorganisation berechnet werden (*Gleichung B.13*).

$$I_{LEV\,ideal} = \left\lceil \log_{NL_{IDEAL}}\left(NP_B\right) \right\rceil + 1 = \left\lceil \frac{\lg\left(NP_B\right)}{\lg\left(NL_{IDEAL}\right)} \right\rceil + 1$$ Gleichung B.13

Ist die Anzahl der Ebenen bekannt, so lässt sich für jede innere Ebene (NP_i) die Zahl der benötigten Blöcke mit *Gleichung B.14* berechnen.

$$NP_i = \left\lceil \frac{NP_i + 1}{NL_{IDEAL}} \right\rceil$$ Gleichung B.14

Damit kann die Gesamtzahl der für einen Indexbaum mindestens benötigten Blöcke (LI_{IDEAL}), beginnend bei der unmittelbar über der Blattebene gelegenen Ebene, berechnet werden (*Gleichung B.15*). Dabei entspricht der Startwert von NP_{i+1} dem Wert NP_B. Werden Daten- und Indexblöcke nicht gemischt gespeichert, so muss noch die Anzahl der Verwaltungsblöcke der Indexpartitionen addiert werden.

$$LI_{IDEAL} = \underbrace{NP_B}_{\text{Blattebene}} + \sum_{i=I_{LEVideal}-1}^{1} \underbrace{\left\lceil \frac{NP_{i+1}}{NL_{IDEAL}} \right\rceil}_{\text{Blöcke innere Ebene}} + \underbrace{DB_{LVP} \cdot T_{ANZP}}_{\text{Verwaltungsblöcke Partition}}$$ Gleichung B.15

Die u.U. nötige Umrechnung in Extents (NE_{IIDEAL}) erfolgt über Gleichung B.16.

$$NE_{IIDEAL} = \left\lceil \frac{LI_{IDEAL}}{TS_{ESIZE}} \right\rceil$$ Gleichung B.16

Kann die Zahl der Blöcke, die ein „gewachsener“ Index aktuell belegt, im Rahmen einer Reorganisationsbedarfsanalyse nicht aus Statistikdaten abgelesen werden, so muss sie unter Verwendung der Werte für die aktuelle Anzahl Ebenen im Index (I_{LEV})

und die aktuelle Anzahl Blattknoten (I_{LEAF}) geschätzt werden. Hier fließt die Annahme ein, dass sich der Füllungsgrad der Knoten auf den inneren Ebenen durch das Löschen von Einträgen auf der Blattebene nicht wesentlich verschlechtert. Diese Annahme beruht darauf, dass gängige DBMS-Produkte Verschmelzungen von Indexknoten aus Aufwandsgründen weitgehend vermeiden. Die schlechtere Speicherplatzauslastung (gegenüber einem „normal gepflegten“ B-Baum) wird in Kauf genommen.

Werden vom jeweiligen DBMS-Produkt Daten und Indexe gemischt gespeichert, so muss der Speicherbedarf von Daten und Indexen vor der Umrechnung in Extents summiert werden.

Speicherplatzabschätzung für indexorganisierte Tabellen

Zur Abschätzung des Speichers, der für Daten und Hauptindex einer indexorganisierten Tabelle benötigt wird, kann im Wesentlichen wie bei anderen B*-Baum-Indexen auch vorgegangen werden. Bei der Berechnung der Länge der Einträge auf der Blattebene müssen allerdings alle in den Blättern des Hauptindex der IOT gespeicherten Felder (einschließlich Längeninformationen) berücksichtigt werden. Abhängig davon, ob beim Anlegen der IOT Teile der Sätze zur Speicherung in Überlaufbereichen vorgesehen wurden, muss noch ein Verweis (Zeiger) auf den jeweiligen Überlaufsatz gespeichert werden. Zusätzlich ist die Länge für die Verwaltungsinformationen zu berücksichtigen, die zu einem Satz gespeichert werden. Zur Berechnung der Anzahl Datensätze je Blattblock (*EP*) können hier die beiden folgenden Gleichungen verwendet werden.

Die Berechnung der Länge der Sätze auf der Blattebene der IOT kann mit *Gleichung B.17* erfolgen.

$$LS_B = \sum_{\forall Felder\ im\ Blatt} \left(A_{LEN} + A_{LVA}\right) + DB_{LVS} + \underbrace{DB_{LTID}}_{Verweis\ auf\ Überlaufsatz} \qquad \text{Gleichung B.17}$$

Der Wert kann jetzt in *Gleichung B.18* zur Berechnung der Anzahl Sätze je Blattknoten eingesetzt werden.

$$EP = \left\lfloor \frac{\overbrace{\left(DB_{PS} - DB_{VIP} - 2 \cdot DB_{LBA}\right)}^{Block\ abzügl.\ Verwaltungsdaten\ und\ Zeiger\ für\ Verkettung} \cdot \overbrace{\left(I_{FILL} / 100\right)}^{Füllungsgrad}}{LS_B} \right\rfloor \qquad \text{Gleichung B.18}$$

Ab hier können die weiteren Berechnungen wie für „normale“ B*-Baum-Indexe (*Gleichung B.11*) erfolgen. Für die Berechnung der in *Gleichung B.12* benötigten Schlüssellänge (*LK*) kann der über Gleichung B.17 berechnete Wert LS_B verwendet werden.

Da die eventuell zu indexorganisierten Tabellen gehörenden Überlaufbereiche als Heap organisiert sind, kann die Abschätzung des für solche Überlaufbereiche benötigten Speichers wie oben beschrieben (*Gleichung B.1* bis *Gleichung B.7*) erfolgen. Es sind bei der Berechnung der Satzlänge in *Gleichung B.1* allerdings nur

die Längen der Felder zu berücksichtigen, die für die Speicherung im jeweiligen Überlaufbereich vorgesehen sind.

Speicherplatzabschätzung für Hash-Tabellen

Bei gängigen Implementierungen für Hash-Tabellen wird i.d.R. der gesamte Speicher für den primären Hash-Bereich bereits beim Anlegen der Tabelle reserviert. Damit ist dessen Größe zunächst unabhängig von der unmittelbar zu speichernden Datenmenge. Es wird davon ausgegangen, dass der reservierte Speicher sukzessive mit Daten gefüllt wird. Die Größe des Bereichs ergibt sich aus der Anzahl Buckets, multipliziert mit der Bucket-Größe. Zur Behandlung von Kollisionen und Bucket-Überläufen wird häufig das Verfahren des „Seperate Chaining“ verwendet, bei dem Überlaufketten gebildet werden. Solche Überlaufketten sind i.d.R. unerwünscht, da deren Durchsuchen negative Auswirkungen auf die Verarbeitungskosten hat. Durch überlegte Wahl der Bucket-Größe (wenn das DBMS dies zulässt) kann u.U. erreicht werden, dass i.d.R. alle Daten in einem Bucket untergebracht werden können, für die die Hash-Funktion die gleiche Bucket-Adresse ermittelt. Dabei wird allerdings auch vorausgesetzt, dass die Hash-Funktion die Buckets gleichmäßig füllt. Zur Berechnung der Bucket-Größe (C_{BSIZE} - *Tabelle A.6*) müssen die (erwartete) Anzahl Hash-Werte (I_{HKEYS} - *Tabelle A.4*) und die Satzlänge (LT – *Gleichung A.1*) bekannt sein.

$$C_{BSIZE} = \left\lceil \frac{T}{I_{HKEYS}} \cdot \underbrace{\left(LT + DB_{LVS}\right)}_{\text{Satzlänge inkl. Verwaltungsinformation}} \right\rceil \qquad \text{Gleichung B.19}$$

Wenn die Bucket-Größe kleiner als die verfügbare Blockgröße ist, kann mit *Gleichung B.20* die Berechnung der Größe des primären Datenbereichs in Blöcken (P) vorgenommen werden. Dabei wird davon ausgegangen, dass ein Bucket nicht auf mehrere Blöcke verteilt wird.

$$P = \underbrace{\left\lfloor \frac{\overbrace{\left(DB_{PS} - DB_{VDP}\right)}^{\text{verfügb. Teil im Datenblock}}}{C_{BSIZE}} \right\rfloor}_{\text{Buckets je Block}} \cdot I_{HKEYS} + \underbrace{DB_{LVP} \cdot T_{ANZP}}_{\text{Verwaltungsböcke Partitionen}} \qquad \text{Gleichung B.20}$$

Übersteigt die Bucket-Größe die der Blöcke, so kann die Größe des primären Hash-Bereichs mit *Gleichung B.21* berechnet werden.

$$P = \underbrace{\left\lceil \frac{C_{BSIZE}}{\underbrace{\left(DB_{PS} - DB_{VDP}\right)}_{\text{verfügb. Teil im Datenblock}}} \right\rceil}_{\text{Blöcke je Bucket}} \cdot I_{HKEYS} + \underbrace{DB_{LVP} \cdot T_{ANZP}}_{\text{Verwaltungsböcke Partitionen}} \qquad \text{Gleichung B.21}$$

Die abschließende Berechnung der Anzahl der zu reservierenden Extents bzw. der Extent-Größen kann analog zu *Gleichung B.5* bzw. *Gleichung B.6* erfolgen.

Speicherplatz für Hash- und Index-Cluster

Bei Speicherplatzabschätzungen für Cluster müssen mehrere Tabellen berücksichtigt werden. Die Daten der Tabellen werden nach den Werten der Cluster-Schlüssel angeordnet. Dabei wird für das Auftreten der Cluster-Schlüsselwerte wieder eine Gleichverteilung angenommen. Zunächst muss die Länge der Sätze der jeweiligen Tabellen analog zu *Gleichung B.1* bestimmt werden. Zur Berechnung der Bucket-Größe bei Hash-Clustern (*Gleichung B.22*) werden die jeweiligen Tupelanzahlen (T) und die Satzlängen der einzelnen Tabellen (LT) sowie die Anzahl der verschiedenen Hash-Werte (I_{HKEYS}) benötigt.

$$C_{BSIZE} = \sum_{\forall Tabellen des\ Clusters} \left\lceil \left(LT + DB_{LVS}\right) \cdot \frac{T}{I_{HKEYS}} \right\rceil$$

Gleichung B.22

Die Berechnung der für den Datenbereich des Clusters benötigten Anzahl Blöcke kann, abhängig von der Bucket-Größe, mit *Gleichung B.20* bzw. *Gleichung B.21* erfolgen. Zur Berechnung der Anzahl der zu reservierenden Extents bzw. der Extent-Größen wird hier auf *Gleichung B.5* bzw. *Gleichung B.6* verwiesen.

Bei der Oracle-Implementierung von Index-Clustern ist zu beachten, dass der Cluster-Schlüssel in jedem Bucket nur einmal gespeichert wird. Daher muss die Länge des Cluster-Schlüssels von den Längen der Sätze der einzelnen Tabellen abgezogen und separat eingerechnet werden. Zur Berechnung der Bucket-Größe (*Gleichung B.23*) muss neben Satzlängen und Tupelanzahlen die Anzahl der verschiedenen Cluster-Schlüsselwerte (I_{UNQ}) bekannt sein.

$$C_{BSIZE} = \underbrace{A_{LEN} + A_{LVA}}_{Cluster-Schlüssel} + \sum_{\forall Tabellen\ des\ Clusters} \left\lceil \left(LT - \underbrace{A_{LEN} - A_{LVA}}_{Cluster-Schlüssel} + DB_{LVS} \right) \cdot \frac{T}{I_{UNQ}} \right\rceil$$

Gleichung B.23

Die weiteren Berechnungen können wie bei Hash-Clustern erfolgen. In *Gleichung B.20* bzw. *Gleichung B.21* muss lediglich die Anzahl der Hash-Werte *(I_{HKEYS})* durch die Anzahl der Cluster-Schlüsselwerte (I_{UNQ}) ersetzt werden.

Anhang C – Kostenschätzung für weitere Operationen

Weitere Zugriffsoperationen

In diesem Anhang werden noch weitere Kostenfunktionen für Datenbankoperationen aufgeführt. Deren Beschreibung im Hauptteil hätte den Rahmen der Arbeit gesprengt. In [Kli05] wurden am Beispiel von Oracle9i und Oracle 10g auch Untersuchungen zur Überprüfung der Kostenformeln angestellt.

Mit *Gleichung C.1* kann der im eInformationsschema als normalisierter Wert aufgeführte Clustering Factor in die für Kostenschätzungen benötigte Anzahl Blockwechsel (N_{BW}) umgerechnet werden.

$$N_{BW} = \left(P_{ANZT} - P_{USED}\right) \cdot \left(1 - \frac{I_{CLUST}}{100}\right) + P_{USED} \qquad \text{Gleichung C.1}$$

Im Gegensatz zu den Betrachtungen für Zugriffe über eindeutige Indexe in *Abschnitt 5.4.1.2* muss bei der Abschätzung der bei einem *Index Lookup über einem nicht eindeutigen Index* anfallenden Kosten beachtet werden, dass die zu einem Schlüsselwert gehörende Liste der Verweise auf Datensätze u.U. auf mehrere Blätter verteilt sein kann. Dies wird über den Quotient aus der Anzahl Blätter (I_{LEAF}) und der Anzahl unterschiedlicher Schlüsselwerte im Index (I_{UNQ}) berücksichtigt. Ist der Quotient größer als 1, so muss im Mittel auf mehr als ein Blatt zugegriffen werden. Unter der Annahme, dass alle Indexeinträge nahezu gleich lang sind, wäre der Wert aufzurunden. Wird jedoch angenommen, dass Indexeinträge unterschiedliche lang sind, weil für das Auftreten der Werte des Indexschlüssels keine genaue Gleichverteilung angenommen werden kann, so ergibt die Verwendung des genauen Werts des Quotienten auch genauere Ergebnisse. Allerdings muss immer auf mindestens ein Blatt zugegriffen werden. Dies wird über die Funktion „`max(...)`“ ausgedrückt. Die hier angegebenen Gleichungen können auch bei Zugriffen über eindeutige Indexe verwendet werden. Mit *Gleichung C.2* kann die bei einem Index Lookup über einen nicht eindeutigen Index anfallende Anzahl logischer Blockzugriffe näherungsweise bestimmt werden.

$$C_{LILookup} = \underbrace{\left(I_{LEV} - 1\right)}_{\text{innere Ebenen}} + \underbrace{\max\left(\frac{I_{LEAF}}{I_{UNQ}}, 1\right)}_{\text{Blattebene}} + \underbrace{\frac{P_{ANZT} + P_{OVFL}}{I_{UNQ}}}_{\text{Datenzugriffe}} \qquad \text{Gleichung C.2}$$

Die Berücksichtigung der Einflüsse der Pufferung führt zu *Gleichung C.3*.

$$C_{PILookup} = \underbrace{\sum_{i=1}^{I_{LEV}-1}\left(1 - I_{BR_i}\right)}_{\text{innere Ebenen}} + \underbrace{\max\left(\frac{I_{LEAF}}{I_{UNQ}}, 1\right) \cdot \left(1 - I_{BR_h}\right)}_{\text{Blattebene}} + \underbrace{\frac{P_{ANZT} + P_{OVFL}}{I_{UNQ}} \cdot \left(1 - P_{BR}\right)}_{\text{Datenzugriffe}}$$

Gleichung C.3

Bei einem Index Lookup *über den Hauptindex* einer *indexorganisierten Tabelle* ergibt sich die Anzahl notwendiger Blockzugriffe aus der Höhe des Indexbaums.

Hinzu kommen noch Zugriffe auf eventuell vorhandene Überlaufbereiche. Hier kann die Anzahl logischer Blockzugriffe mit *Gleichung C.4* geschätzt werden.

$$C_{LILookupIOT} = \underbrace{I_{LEV} - 1}_{\text{innere Ebenen}} + \underbrace{\max\left(\frac{I_{LEAF}}{P_{ANZT}}, 1\right)}_{\text{Blattebene}} + \underbrace{\frac{P_{OVFL}}{P_{ANZT}}}_{\text{Überlaufbereich}} \qquad \text{Gleichung C.4}$$

In *Gleichung C.5* werden zusätzlich die Einflüsse der Datenbankpufferung berücksichtigt.

$$C_{PILookupIOT} = \underbrace{\sum_{i=1}^{I_{LEV}-1}\left(1 - I_{BR_i}\right)}_{\text{innere Ebenen}} + \underbrace{\max\left(\frac{I_{LEAF}}{P_{ANZT}}, 1\right) \cdot (1 - I_{BRh})}_{\text{Blattebene}} + \underbrace{\frac{P_{OVFL}}{P_{ANZT}} \cdot (1 - P_{BR})}_{\text{Überlaufbereich}} \qquad \text{Gleichung C.5}$$

Etwas aufwendiger ist die Schätzung des Aufwands für Index Lookups *über indexorganisierten Tabellen* zugeordnete *Sekundärindexe*. Hierbei müssen die Eigenheiten der Verweise in Sekundärindexen auf die in den Blattknoten der indexorganisierten Tabelle enthaltenen Sätze berücksichtigt werden. Die von Oracle verwendeten Universal ROWIDs (UROWID) basieren auf den Schlüsselwerten des Hauptindex der IOT und enthalten zusätzlich einen Verweis auf den Datenblock, in dem der zu referenzierende Satz beim Anlegen des Index gespeichert war. Damit kann ohne weiteres Durchsuchen des Hauptindex der IOT auf den Satz zugegriffen werden, solange der Satz nicht auf Grund von Splitting-Operationen oder Änderungen in einen anderen Block verschoben wird. Durch ein solches Verschieben wird die Blockadresse ungültig, und der Zugriff muss über den Hauptindex erfolgen.

Der Statistikwert `PCT_DIRECT_ACCESS` gibt bei Oracle den prozentualen Anteil korrekter Blockadressen an. Die entsprechende Größe wird im eInformationsschema mit I_{DIR_ACC} bezeichnet. *Abbildung C.1* zeigt die unterschiedlichen Zugriffsmöglichkeiten.

Zunächst wird der Sekundärindex durchsucht (Abbildung C.1 - ①). Ist die Blockadresse korrekt, so kann direkt auf die entsprechenden Daten zugegriffen werden (Abbildung C.1 - ②). Wenn die Blockadresse nicht korrekt ist (oder als nicht korrekt angenommen wird), wird über den Hauptindex der IOT (Abbildung C.1 - ③) zugegriffen, der anschließend durchsucht wird (Abbildung C.1 - ④). Wurden die entsprechenden Datensätze gefunden, muss eventuell noch auf vorhandene Überlaufsätze zugegriffen werden (Abbildung C.1 - ⑤).

Davon ausgehend, dass die in den Blattknoten der indexorganisierten Tabelle gespeicherten Sätze nicht über mehrere Knoten verteilt werden und dass die IOT über einem eindeutigen Schlüssel aufgebaut ist, kann hier die Anzahl logischer Blockzugriffe mit *Gleichung C.6* geschätzt werden.

$$C_{LILookupIOTSK} = \underbrace{\left(I_{LEVSK} - 1\right)}_{\text{innere Ebenen Sekundärindex}} + \underbrace{\max\left(\frac{I_{LEAFSK}}{I_{UNQSK}}, 1\right)}_{\text{Blattebene Sekundärindex}} + \underbrace{\frac{P_{ANZT}}{I_{UNQSK}}}_{\text{Verweise je Eintrag}} \cdot$$

$$\cdot \left(\underbrace{1}_{\text{Versuch Zugriff über Blockadresse}} + \underbrace{\left(1 - \frac{I_{DIR\_ACC}}{100}\right) \cdot I_{LEVIOT}}_{\text{Durchsuchen des Hauptindex der IOT}} + \underbrace{\frac{P_{OVFL}}{P_{ANZT}}}_{\text{Überlaufbereich}} \right)$$

Gleichung C.6

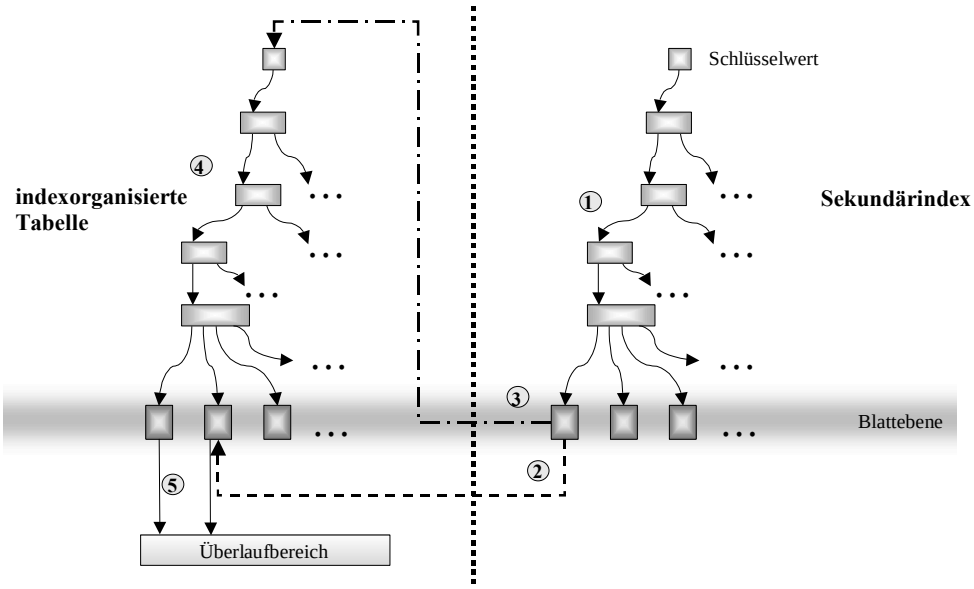


Abbildung C.1: Zugriffspfade bei Sekundärindexen über indexorganisierten Tabellen

Um zwischen den Statistikdaten des Sekundärindex und denen der indexorganisierten Tabelle unterscheiden zu können, sind die Formelzeichen mit dem Zusatz *SK* (Sekundärindex) bzw. *IOT* erweitert worden. Dies gilt ebenso bei der zusätzlichen Berücksichtigung der Pufferungsgrade in *Gleichung C.7*.

$$C_{PILookupIOTSK} = \underbrace{\sum_{i=1}^{I_{LEVSK}-1}\left(1 - I_{BRi}\right)}_{\text{innere Ebenen Sekundärindex}} + \underbrace{\max\left(\frac{I_{LEAFSK}}{I_{UNQSK}}, 1\right) \cdot \left(1 - I_{BRhSK}\right)}_{\text{Blattebene Sekundärindex}} + \underbrace{\frac{P_{ANZT}}{I_{UNQSK}}}_{\text{Verweise je Eintrag}} \cdot$$

$$\cdot \left(\underbrace{\left(1 - I_{BRhIOT}\right)}_{\text{Versuch Zugriff über Blockadresse}} + \underbrace{\left(1 - \frac{I_{DIR\_ACC}}{100}\right) \cdot \sum_{i=1}^{I_{LEVIOT}}\left(1 - I_{BRiIOT}\right)}_{\text{Durchsuchen des Hauptindex der IOT}} + \underbrace{\frac{P_{OVFL}}{P_{ANZT}} \cdot \left(1 - P_{BR}\right)}_{\text{Überlaufbereich}} \right)$$

Gleichung C.7

Die Anzahl logischer Blockzugriffe für eine Bereichssuche über dem Schlüssel des Hauptindex einer indexorganisierten Tabelle kann mit *Gleichung C.8* geschätzt werden.

$$C_{LIScanIOT} = \underbrace{(I_{LEV} - 1)}_{\text{innere Ebenen}} + \underbrace{S \cdot (I_{LEAF} + P_{OVFL})}_{\text{Blattebene und Überlaufbereich}}$$

Gleichung C.8

Die Berücksichtigung der Einflüsse der Datenbankpufferung führt zu *Gleichung C.9*.

$$C_{PIScanIOT} = \underbrace{\sum_{i=1}^{I_{LEV}-1}(I_{LEV} - 1)}_{\text{innere Ebenen}} + \underbrace{S \cdot I_{LEAF} \cdot (1 - I_{BRh})}_{\text{Blattebene}} + \underbrace{S \cdot P_{OVFL} \cdot (1 - P_{BR})}_{\text{Überlaufbereich}}$$

Gleichung C.9

Bei der Abschätzung des Aufwands für eine Bereichssuche über einen zu einer indexorganisierten Tabelle gehörenden Sekundärindex muss ebenfalls das bei solchen Indexen verwendete Adressierungskonzept beachtet werden. Bei der Abschätzung der anfallenden I/O-Kosten mit *Gleichung C.10* wird angenommen, dass die in den Blattknoten der indexorganisierten Tabelle gespeicherten Sätze nicht über mehrere Knoten verteilt sind und dass die IOT über einem eindeutigen Schlüssel aufgebaut ist.

$$C_{PIScanIOTSK} = \underbrace{\underbrace{\sum_{i=1}^{I_{LEVSK}-1}(1 - I_{BRiSK})}_{\text{innere Ebenen}} + \underbrace{\max\left(S \cdot I_{LEAF_{SK}} \cdot (1 - I_{BRhSK}), (1 - I_{BRhSK})\right)}_{\text{Blattebene Sekundärindex}}}_{\text{Zugriff auf Sekundärindex}} + \underbrace{S \cdot P_{ANZT}}_{\text{Anzahl Datenzugriffe}} \cdot$$

$$\cdot \underbrace{\left(\underbrace{(1 - I_{BRhIOT})}_{\substack{\text{Versuch Zugriff über} \\ \text{Blockadresse}}} + \underbrace{\left(1 - \frac{I_{DIR\_ACC}}{100}\right) \cdot \sum_{i=1}^{I_{LEVIOT}}(1 - I_{BRiIOT})}_{\text{Durchsuchen des Hauptindex der IOT}} + \underbrace{\frac{P_{OVFL}}{P_{ANZT}} \cdot (1 - P_{BR})}_{\text{Überlaufbereich}} \right)}_{\text{Aufwand für einen Datenzugriff}}$$

Gleichung C.10

In Gleichung C.10 werden auch die Einflüsse der Datenbankpufferung berücksichtigt. Auch hier sind die Formelzeichen mit dem Zusatz *SK* (Sekundärindex) bzw. *IOT* erweitert worden, um zwischen den Größen, die den Sekundärindex betreffen und denen, die die indexorganisierten Tabelle betreffen, unterscheiden zu können.

Änderungsoperationen bei Index-Clustern

Wird ein neuer Satz in einen Index-Cluster *eingefügt*, so muss, für den Fall dass der Cluster-Schlüsselwert bereits existiert, zunächst über den Cluster-Index auf das Bucket zugegriffen werden, in das der neue Satz einzufügen ist. Muss ein Bucket bzw. die zugehörige Überlaufkette erweitert werden (Anfügen eines Blocks an die Überlaufkette), so muss in jedem Fall auch auf den bisherigen letzten Block zugegriffen werden, da dort der Verweis für die Verkettung zum neuen Block eingetragen werden muss. Der dabei anfallende Log-Aufwand wird mit LOG_{OVFL} angenommen. Die Wahrscheinlichkeit dafür, dass ein neues Bucket reserviert werden muss, ist von der Anzahl Buckets (also der Anzahl eindeutiger Schlüsselwerte des Cluster-Index – I_{UNQ}) und dem Verhältnis von Bucket-Größe (C_{BSIZE}) zur Länge des einzufügenden Satzes (LT) zuzüglich der Verwaltungsinformation für den Satz (DB_{LVS}) abhängig. Beim Reservieren eines neuen Blocks werden zwei Zugriffe

eingerechnet, einer für die Aktualisierung der Daten der Freispeicherverwaltung und einer für das Schreiben des neuen Blocks. Wenn ein neuer Cluster-Schlüsselwert eingefügt werden muss, so muss auch ein neuer Eintrag im Cluster-Index erzeugt und ein neues Bucket reserviert werden. Die Wahrscheinlichkeit, das dieser Fall eintritt wird über die Zahl der Schlüsselwerte im Cluster-Index und die Anzahl der im Cluster gespeicherten Sätze aller Tabellen geschätzt, die bei Clustern der Anzahl der in der Datenpartition des Clusters gespeicherten Sätze (P_{ANZT}) entspricht. Die Berechnung des Aufwands für das Einfügen eines Eintrags in den Cluster-Index ($C_{InsertIndex}$) kann über *Gleichung 5.19* erfolgen.Der Aufwand für das Einfügen eines neuen Satzes kann mit *Gleichung C.11* geschätzt werden. Dabei wird davon ausgegangen, dass immer an deren Ende eingefügt wird, wenn eine Überlaufkette existiert.

$$C_{InsertIClusterDaten} = \underbrace{\frac{I_{UNQ}}{P_{ANZT}} \cdot \left(\underbrace{2}_{Reserv.\ und\ Schreiben\ Bucket} + LOG_{InsertDaten} + \underbrace{C_{InsertIndex} + C_{LogInsertIndex}}_{Eintrag\ in\ Cluster-Index} \right)}_{Einfügen\ eines\ neuen\ Cluster-Schlüssels\ und\ Reservieren\ eines\ neuen\ Buckets\ zum\ Einfügen} +$$

$$+ \left(\underbrace{1 - \frac{I_{UNQ}}{P_{ANZT}}}_{Wahrscheinlichkeit,\ dass\ Cluster-Schlüssel\ vorhanden} \right) \cdot$$

$$\cdot \left(\underbrace{\underbrace{\sum_{i=1}^{I_{LEV}} (1 - I_{BRi})}_{Indexteil} + \underbrace{\left(1 + \frac{I_{CHAINS}}{I_{UNQ}}\right) \cdot (1 - P_{BR})}_{Datenteil} + \underbrace{1}_{Schreibaufwand} + LOG_{InsertDaten}}_{Suche\ nach\ Bucket,\ wenn\ Schlüsselwert\ vorhanden} + \right.$$

$$\left. + \underbrace{2 \cdot I_{UNQ} \cdot \frac{C_{BSIZE}}{LT + DB_{LVS}} + LOG_{OVFL}}_{Aufwand\ für\ Verlängerung\ Überlaufkette} \right)$$

Gleichung C.11

Existieren über der Tabelle des Clusters, in die ein Satz einzufügen ist, noch weitere Indexe, so müssen auch diese Indexe gepflegt werden. Der Gesamtaufwand für das Einfügen eines Satzes in einen Index-Cluster kann über *Gleichung C.12* ermittelt werden.

$$C_{Insert} = C_{InsertIClusterDaten} + \underbrace{\sum \left(C_{InsertIndex} + C_{LogInsertIndex} \right)}_{\forall Indexe\ außer\ Cluster-Index}$$

Gleichung C.12

Für das *Löschen von Sätzen aus einem Index-Cluster* wird angenommen, dass einmal reservierte Buckets im normalen Betrieb nicht wieder freigegeben werden. Ausnahmen, wie bspw. die Verwendung der Anweisung

```
ALTER CLUSTER ... DEALLOCATE UNUSED
```

Werden in [Sin00] beschrieben.

Damit ist das Verhalten beim *Löschen* ähnlich dem Verhalten bei als Heap organisierten Tabellen und es können die Gleichungen aus *Abschnitt 5.4.2.2* angewendet werden. Allerdings darf hier gemäß den Annahmen beim Berechnen des

Aufwands für die Pflege der Indexe in *Gleichung 5.24* der Cluster-Index nicht berücksichtigt werden.

Zur Abschätzung des anfallenden Aufwands beim *Ändern* von in Index-Clustern gespeicherten Sätzen kann auf die Ausführungen in *Abschnitt 5.4.2.3* zurückgegriffen werden. Migrierte Tupel können bei Clustern allerdings nicht nur bei der Verlängerung von Sätzen entstehen. Ändern sich bei Tupeln die Werte von Attributen, über denen der Cluster-Schlüssel definiert ist, so müssen die zugehörigen Sätze physisch verschoben werden. Um auch hier das Nachpflegen eventuell vorhandener zusätzlicher Indexe zu vermeiden, wird ebenfalls mit Auslagerungszeigern gearbeitet. Da aber die Wahrscheinlichkeiten, dass ein Satz wegen einer verlängernden Operation ausgelagert oder dass ein Satz wegen einer Änderung eines im Cluster-Schlüssel enthaltenen Werts verschoben wird, nicht einzeln bekannt sind, wird auch hier (wie in *Gleichung 5.25*) die Wahrscheinlichkeit für das Auslagern von Sätzen über das Verhältnis von vorhandenen ausgelagerten Sätzen zur Gesamtzahl der im Cluster gespeicherten Sätze geschätzt.

Änderungsoperationen bei Hash-Clustern

Da bereits beim Anlegen von Hash-Clustern für alle möglichen Hash-Werte die entsprechenden Buckets erzeugt werden, ist die Ermittlung des Aufwands für *Einfügen neuer* Sätze einfacher als bei Index-Clustern (*Gleichung C.13*).

$$C_{Insert_{HClusterDaten}} = \left(\underbrace{(1 - P_{BR})}_{Leseaufwand} + \underbrace{1}_{Schreibaufwand} + \underbrace{2 \cdot I_{HKEYS} \cdot \frac{C_{BSIZE}}{LT + DB_{LVS}} + LOG_{Insert_{Daten}}}_{Aufwand\ für\ Verlängerung\ Überlaufkette} \right)$$

Gleichung C.13

Die Bestimmung des bei einer Einfügeoperation anfallenden Gesamtaufwands kann mit *Gleichung C.14* erfolgen, wenn über der Tabelle, in die ein Satz einzufügen ist, noch weitere Indexe existieren.

$$C_{Insert} = C_{Insert_{HClusterDaten}} + \sum_{\forall Indexe\ der\ Tabelle} \left(C_{Insert_{Index}} + C_{Log_{InsertIndex}} \right)$$

Gleichung C.14

Zur Abschätzung des Aufwands für das *Löschen* und *Ändern* von in Hash-Clustern gespeicherten Daten gelten die Ausführungen in den *Abschnitten 5.4.2.2* und *5.4.2.3*, da sich in Hash-Clustern gespeicherte Tabellen bezüglich dieser Operationen so verhalten, wie normale Tabellen mit als Heap organisierten Datensegmenten. Lediglich bei der Ermittlung der Suchkosten muss berücksichtigt werden, dass der Zugriff auf die zu löschenden bzw. zu ändernden Sätze evtl. über eine Hash-Funktion erfolgen kann.

Änderungsoperationen bei indexorganisierten Tabellen

Das *Einfügen* von Daten in eine indexorganisierte Tabelle ähnelt dem in *Abschnitt 5.4.2.1* beschriebenen Einfügen eines neuen Eintrags in einen „normalen“ B*-Baum-

Index. Zur Berechnung der Wahrscheinlichkeit, dass es auf der Blattebene des Hauptindex der IOT zum Aufspalten von Knoten kommt, muss auf den in *Gleichung B.18* errechneten Wert für die Anzahl Einträge je Knoten zurückgegriffen werden. Der Aufwand für das Einfügen eines Überlaufsatzes muss nur berücksichtigt werden, wenn Teile der Sätze zur Speicherung in Überlaufbereichen vorgesehen sind. Der Aufwand für das Einfügen eines Datensatzes kann mit *Gleichung C.15* erfolgen.

$$C_{InsertDatenIOT} = \underbrace{\underbrace{\sum_{i=1}^{I_{LEV}} \left(1 - I_{BRi}\right)}_{Durchsuchen} + \underbrace{1}_{Schreiben\,Blatt} + \underbrace{3 \cdot \left(W_{BSplit} \cdot \sum_{i=0}^{I_{LEV}-1} W_{ISplit}^{\;i} \right)}_{Zusatzaufwand\,durch\,Blocksplitting} + \underbrace{W_{BSplit} \cdot W_{ISplit}^{\;I_{LEV}-1}}_{Aufwand\,neueWurzel}}_{Hauptindex} + \underbrace{\left(1 - P_{BR}\right) + \frac{1}{TP} + 1}_{Überlaufbereich}$$

Gleichung C.15

Bei der Berechnung der Länge der Einträge auf der Blattebene (*EP*) von zu indexorganisierten Tabellen gehörenden Sekundärindexen ist zu beachten, dass die Verweise auf die Datensätze der IOT neben der Adresse des Blocks, in dem der jeweilige Satz vermutet wird, auch den zum Satz gehörenden Wert des Schlüssels enthalten, über dem die IOT aufgebaut ist. Der Wert wird zur Bestimmung des Aufwands für die Aktualisierung des Sekundärindexe ($C_{InsertIndex}$ - *Gleichung 5.19*) benötigt. Der gesamte I/O-Aufwand beim Einfügen eines Satzes in eine IOT kann mit *Gleichung C.16* geschätzt werden. Wird neben dem Satz auf der Blattebene der IOT noch ein Überlaufsatz eingefügt, so müssen zwei Log-Einträge für das Einfügen von Datensätzen geschrieben werden.

$$C_{Insert} = C_{InsertDatenIOT} + \underbrace{2}_{Zugr.f.\,Schreiben\,Log-Einträge} + \sum_{\forall Sekundärindexe} \left(C_{InsertIndex} + C_{LogInsertIndex} \right)$$

Gleichung C.16

Wird bei indexorganisierten Tabellen davon ausgegangen, dass beim *Löschen* von Sätzen keine Verschmelzungen von Knoten durchgeführt werden, kann der Aufwand für das Löschen eines Satzes (und eines evtl. vorhandenen Überlaufsatzes) mit *Gleichung C.17* bestimmt werden.

$$C_{DeleteDatenIOT} = \underbrace{\left(1 - I_{BRh}\right) + 1}_{Aufwand\,Hauptindex} + \underbrace{\left(1 - P_{BR}\right) + 1}_{Aufwand\,Überlaufbereich}$$

Gleichung C.17

Der Aufwand für das Pflegen der Sekundärindexe ($C_{DeleteIndex}$) kann über *Gleichung 5.23* ermittelt werden. Der gesamte I/O-Aufwand beim Löschen eines Satzes aus einer IOT kann mit *Gleichung C.18* geschätzt werden. Neben dem reinen Löschaufwand ist auch der Aufwand für das Suchen der zu ändernden Datensätze zu berücksichtigen. Muss auch noch ein Überlaufsatz gelöscht werden, so müssen zwei Log-Einträge für das Löschen von Datensätzen geschrieben werden.

$$C_{Delete} = C_{Suche} + \underbrace{S \cdot P_{ANZT}}_{Anz.\,Löschungen} \cdot \left(\begin{array}{l} C_{DeleteDatenIOT} + 2 \cdot LOG_{delD} + \\ + \sum_{\forall Sekundärindexe} \left(C_{DeleteIndex} + C_{LogDeleteIndex} \right) \end{array} \right)$$

Gleichung C.18

Beim Ändern von Datensätzen fällt ebenfalls der Aufwand für das Suchen der Datensätze an. Die Änderungen können an Werten vorgenommen werden, die in der Blattebene des Hauptindex der IOT oder im Überlaufbereich gespeichert sind. Da i.d.R. aus den Informationen in Datenbankkatalogen und auch aus dem eInformationsschema keine Informationen darüber abgeleitet werden können, mit welcher Wahrscheinlichkeit bestimmte Attribute geändert werden, müssen Annahmen getroffen werden. Vereinfachend wird hier zunächst angenommen, dass die Änderungen dabei jeweils am ursprünglichen Speicherort durchgeführt werden können. Dabei steht:

- $W_{UpdateLeaf}$ für die Wahrscheinlichkeit, dass nur auf der Blattebene gespeicherte Sätze geändert werden,
- $W_{UpdateOvfl}$ für die Wahrscheinlichkeit, dass nur in Überlaufbereichen gespeicherte Sätze geändert werden müssen und
- $W_{UpdateAll}$ für die Wahrscheinlichkeit, dass Änderungen an beiden Sätzen notwendig sind.

Wird für ein Tupel der Wert mindestens eines im Schlüssel (über dem die indexorganisierte Tabelle organisiert ist) enthaltenen Attributs geändert, so muss der entsprechende Datensatz in ein anderes Blatt verschoben werden. Das Verschieben wird über das Löschen des alten Satzes und das anschließende Einfügen des geänderten Satzes in ein neues Blatt durchgeführt. Eventuell existierende Überlaufsätze sind von der Verschiebeoperation nicht betroffen. Der beim Verschieben eines Satzes anfallende Aufwand kann mit *Gleichung C.19* abgeschätzt werden.

$$C_V = \underbrace{\left(1 - I_{BR_h}\right) + 1 + LOG_{Delete_{Daten}}}_{\text{Löschen alter Satz}} + \underbrace{\sum_{i=1}^{I_{LEV}} \left(1 - I_{BR_i}\right)}_{\text{Suchen Einfügestelle}} + \underbrace{\frac{1}{\breve{}}}_{\text{Schreiben}} + \underbrace{3 \cdot \left(W_{BSplit} \cdot \sum_{i=0}^{I_{LEV}-1} W_{ISplit}^{\ i} \right)}_{\text{Zusatzaufwand durch Blocksplitting}} + \underbrace{W_{BSplit} \cdot W_{ISplit}^{\ I_{LEV}-1}}_{\text{Aufwand neueWurzel}} + LOG_{Insert_{Daten}}$$

$$\underbrace{\qquad}_{\text{Einfügen geänderter Satz}}$$

Gleichung C.19

Für die Wahrscheinlichkeit, mit der solche Verschiebeoperationen auftreten, müssen ebenfalls Annahmen getroffen werden. Dafür steht die Größe W_V. Mit Gleichung C.20 kann der Aufwand für das Ändern von Datensätzen und die notwendige Pflege des Hauptindex abgeschätzt werden. Darin eingeschlossen ist auch der Aufwand für das Aufzeichnen der Änderungen im Log.

$$C_{Update_{Daten}} = \underbrace{W_V \cdot C_V}_{\text{Verschieben von Sätzen}} + \underbrace{\left(W_{Update_{Leaf}} + W_{upd_{All}}\right) \cdot 2 +}_{\text{Schreiben geändertes Blatt und Log-Aufwand}} + \underbrace{\left(1 - P_{BR}\right) + \left(W_{Update_{Ovfl}} + W_{Update_{All}}\right) \cdot 2}_{\text{Lesen und Schreiben Ueberlaufblock und Log-Aufwand}}$$

Gleichung C.20

Neben dem Aufwand für das Ändern der Datensätze müssen u.U. auch Einträge in Sekundärindexen gepflegt werden. Hier kann bei der Abschätzung des Aufwands für

die Pflege eines Index wie in *Abschnitt 5.4.2* verfahren werden, wenn die Eigenheiten des Aufbaus der Verweise auf Datensätze in Sekundärindexen von indexorganisierten Tabellen berücksichtig werden. Gepflegt werden müssen dabei Indexe, deren Schlüssel Attribute enthalten, deren Werte geändert wurden. Eine Abschätzung des gesamten mit einer Änderungsoperation an in einer indexorganisierten Tabelle gespeicherten Daten verbundenen I/O-Aufwands kann mit *Gleichung C.21* erfolgen.

$$C_{Update} = C_{Suche} + \underbrace{S \cdot P_{ANZT}}_{Anz.\,Änderungen} \cdot \left(C_{Update_{Daten}} + \sum_{\forall\ betroffenen\ Indexe} \left(C_{Update_{Index}} + C_{Log_{UpdateIndex}} \right) \right)$$

Gleichung C.21

Anhang D – Weitere Kostenfunktionen zur Mehraufwandsabschätzung

Auswirkungen migrierter Tupel

In diesem Anhang sind weitere Gleichungen zur Abschätzung des bei Zugriffsoperationen durch Degenerierungen hervorgerufenen Mehraufwands aufgeführt. Dabei werden die Abschätzungen für Partitionen vorgenommen. Sind die im Rahmen der Analysen betrachteten Tabellen bzw. Indexe nicht partitioniert, so können die Gleichungen so angewendet werden. Werden die Analysen für partitionierte Tabellen und Indexe vorgenommen, so müssen, wenn die Analysen für ganze Tabellen bzw. Indexe durchgeführt werden sollen, zunächst entsprechende Summen (bspw. über die Anzahl migrierter Tupel) über alle Partitionen gebildet werden.

Für die Zugriffsmethode *Index Lookup* kann der durch migrierte Tupel (Satzauslagerungen) verursachte I/O-Mehraufwand allgemein (für eindeutige und nicht eindeutige Indexe) mit *Gleichung D.1* abgeschätzt werden.

$$M_{LIlookup} = \frac{P_{OVFL}}{\underbrace{\left(\underbrace{I_{LEV}-1}_{\text{innere Ebenen}} + \underbrace{\max\left(I_{LEAF}/I_{UNQ},1\right)}_{\text{Blätter}}\right)\cdot I_{UNQ} + P_{ANZT}}_{\text{Indexteil}}} \cdot 100$$

Gleichung D.1

Die Berücksichtigung der Einflüsse der Datenbankpufferung führt zu *Gleichung D.2*.

$$M_{PILookup} = \frac{P_{OVFL}\cdot\left(1-P_{BR}\right)}{\underbrace{\left(\sum_{i=1}^{I_{LEV}-1}\left(1-I_{BRi}\right) + \max\left(I_{LEAF}/I_{UNQ},1\right)\cdot\left(1\text{-}I_{BRh}\right)\right)\cdot I_{UNQ}}_{\text{Indexteil}} + \underbrace{P_{ANZT}\cdot\left(1-P_{BR}\right)}_{\text{Datenteil}}} \cdot 100$$

Gleichung D.2

Werden vorhandene Auslagerungszeiger beim sequenziellen Suchen (*Table Scan*) verfolgt, so sind auch diese direkt, durch das „Springen“ zum neuen Speicherort des Satzes, von Auswirkungen der Satzauslagerungen betroffen. Im Auslagerungsblock wird der entsprechende Satz gelesen und danach wird zum Ausgangsblock zurückgesprungen (*Abbildung 5.6*). Für einen Table Scan mit Verfolgung vorhandener Auslagerungszeiger kann der entstehende Mehraufwand mit *Gleichung D.3* abgeschätzt werden.

$$M_{ScanMT} = \frac{P_{OVFL}\cdot 2}{P_{USED}} \cdot 100$$

Gleichung D.3

Dabei kann unter der Annahme, dass sich Pufferungsgrade von Datenbankobjekten durch eine Reorganisation nicht signifikant ändern, die Berücksichtigung der Pufferungsgrade entfallen, weil nur Zugriffe auf Datenblöcke erfolgen. Beim Verfolgen eines Auslagerungszeigers kann allerdings davon ausgegangen werden,

dass der Block, in dem der Ausgangspunkt des Zeigers liegt und zu dem zurückgesprungen werden muss, sich beim „Rücksprung“ mit hoher Sicherheit noch im Datenbankblockpuffer befindet. Für die Abschätzung des Mehraufwands ergibt sich *Gleichung D.4*.

$$M_{ScanMT} = \frac{P_{OVFL}}{P_{USED}} \cdot 100 \qquad \text{Gleichung D.4}$$

Werden die Auslagerungszeiger beim sequenziellen Suchen nicht verfolgt, so ergibt sich i.d.R.trotzdem ein Mehraufwand, da an den ursprünglichen Speicherorten der Tupel Lücken entstehen, die oft nicht wieder vollständig geschlossen werden können. Dadurch erhöht sich die Zahl der zu lesenden Blöcke. Die Abschätzung des Mehraufwands kann mit *Gleichung 6.6* erfolgen.

Auswirkungen von Überlaufbereichen

Ein übliches Verfahren zur Behandlung der bei der Nutzung von Hash-Verfahren auftretenden Kollisionen ist das *Seperate Chaining*. Dabei wird im Falle des „Überlaufens“ eines Buckets für dieses eine Überlaufkette angelegt. Der Idealwert, jedes Tupel mit einem Blockzugriff zu erreichen, kann nicht eingehalten werden. In der Literatur finden sich unterschiedliche Berechnungsformeln, mit denen die Auswirkungen solcher Überlaufketten abgeschätzt werden können. Diese unterscheiden sich hauptsächlich in den Basiswerten, auf denen die jeweilige Berechnung beruht und in der Genauigkeit, mit der eine Abschätzung des Mehraufwands möglich ist. Ein relativ einfaches Verfahren findet sich z.B. in [Oll92]. Dabei werden als Basisgrößen lediglich die Anzahl der Hash-Werte und die Anzahl der Überlaufblöcke benötigt. Mit diesen Größen, die auch im eInformationsschema enthalten sind, kann der entstehende Mehraufwand mit der von [Oll92] abgeleiteten *Gleichung D.5* abgeschätzt werden. Wenn angenommen wird, dass sich die Pufferungsgrade der Blöcke im Primärbereich und in den Überlaufketten nicht unterscheiden, kann auf deren Berücksichtigung verzichtet werden.

$$M_{HC_{LookupP}} = \frac{I_{CHAINS}}{I_{HKEYS} \cdot 2} \cdot 100 \qquad \text{Gleichung D.5}$$

Bei diesem Berechnungsverfahren wird implizit davon ausgegangen, dass die Hash-Funktion die zu speichernden Tupel gleichmäßig auf die Buckets verteilt und dass auch die Zugriffe gleichmäßig verteilt erfolgen. Weiterhin wird vorausgesetzt, dass

- nur nach im Datenbereich vorhandenen Schlüsseln gesucht wird,
- jeder Schlüsselwert nur einmal auftritt, das Hash-Verfahren also auf den Primärschlüssel angewendet wird und
- dies auch vom DBMS berücksichtigt wird.

Solche Operationen werden in [Küs82] auch als *erfolgreiche Suche* bezeichnet. Dort werden die Auswirkungen von Überlaufketten sehr detailliert untersucht und Näherungsverfahren zur Bestimmung des entstehenden Aufwands angegeben. Zur

näherungsweisen Bestimmung des Aufwands (*s*) beim erfolgreichen Suchen wird dort die folgende Gleichung (*Gleichung D.6*) verwendet.

$$s = 1 + 0{,}5 \cdot \left(\frac{n-1}{m \cdot b} - 1 + \frac{1}{b} \right) - \frac{m}{n} \cdot \left(\frac{(b-1) \cdot (2 \cdot b - 1)}{(12 \cdot b) - (b-1)/4} \right)$$ Gleichung D.6

Dabei wird vorausgesetzt, dass die Tupelzahl (n), die Anzahl Buckets im primären Datenbereich (*m*) und die Anzahl Tupel, die in einem Bucket untergebracht werden können (*b*), bekannt sind. Unter der Voraussetzung, dass in jedem Bucket genau ein Tupel untergebracht wird, vereinfacht sich die Gleichung nach [Küs82] wie in *Gleichung D.7* dargestellt.

$$s = 1 + \frac{n-1}{2 \cdot m}$$ Gleichung D.7

Diese Gleichung entspricht größtenteils Gleichung D.5. Allerdings wird in Gleichung D.7 der Gesamtaufwand und in Gleichung D.5 nur der Mehraufwand bestimmt.

Werden Hash-Verfahren auf Sekundärschlüssel angewendet, so muss in jedem Fall (wenn vorhanden) die gesamte Überlaufkette durchsucht werden. Dies gilt auch, wenn die Schlüsseltransformation auf den Primärschlüssel angewendet wurde und nach einem nicht im Datenbereich vorhandenen Schlüssel gesucht wird. Der bei solchem *erfolglosen Suchen* [Küs82] durch die Überlaufketten entstehende Mehraufwand kann unter der Voraussetzung gleichmäßiger Auslastung des primären Hash-Bereichs und gleichverteilter Zugriffe folgendermaßen mit *Gleichung D.8* abgeschätzt werden.

$$M_{HCLookupS} = \frac{I_{CHAINS}}{I_{HKEYS}} \cdot 100$$ Gleichung D.8

Diese Formel basiert auf der in [Küs82] angegebenen Gleichung zur Bestimmung der mittleren Zahl der Überlauf-Buckets (*a – Gleichung D.9*).

$$a = m * (u - 1)$$ Gleichung D.9

Diese Gleichung wurde lediglich nach dem bei einer „erfolglosen Suche“ anfallenden Aufwand (*u*) umgestellt, da die Anzahl Überlauf-Buckets (I_{CHAINS}) und die Anzahl Buckets im primären Datenbereich (I_{HKEYS}) im eInformationsschema bekannt sind.

Abbildung D.1 zeigt die Auswirkungen von Überlaufbereichen auf die Anzahl notwendiger Blockzugriffe, wenn das Hash-Verfahren auf einen Sekundärschlüssel angewendet wird. Die Vorgehensweise zur Aufnahme der Messreihe entsprach im Wesentlichen der bei Index-Clustern beschriebenen. Dies gilt insbesondere auch für die Gleichverteilungen der Schlüsselwerte und der Datenzugriffe während der Aufnahme der Messreihe.

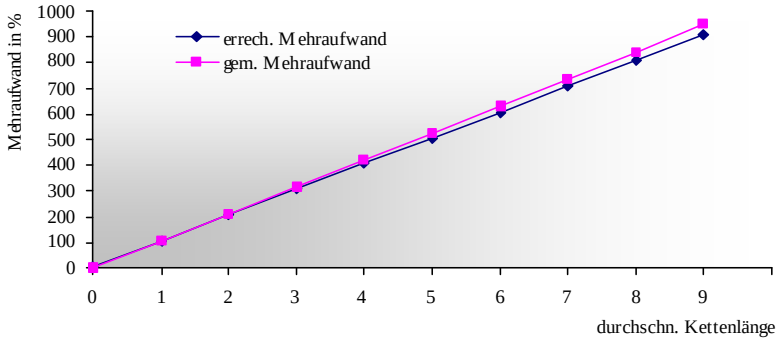


Abbildung D.1: Auswirkungen von Überlaufbereichen auf Zugriffsoperationen mittels Hash-Verfahren über einem Sekundärschlüssel, basierend auf gemessenen Blockzugriffen

Auswirkungen der Nichteinhaltung von internen Sortierreihenfolgen

Bei Tabellen, deren Daten häufig nach einem bestimmten Kriterium sortiert angefordert werden oder wenn die Schlüsselwerte angeforderter Tupel in bestimmten Werteintervallen liegen, ist es sinnvoll, die Sätze auch physisch nach diesem Kriterium sortiert abzuspeichern (wertebasiert zu clustern). Das erspart oder vereinfacht Sortierläufe, wenn die Daten entsprechend geordnet benötigt werden. Bei Datenbank-Management-Systemen wird eine interne Sortierung meist über Indexe (oft Clusterd Index genannt) erreicht. Werden die Daten in als Heap organisierten Bereichen gespeichert, so wird die Sortierung w aus Aufwandsgründen bei Einfüge- und Änderungsoperationen i.d.R. nicht weiter beachtet. Damit geht die Sortierung sukzessive verloren. Da eine physische Sortierung nur nach einer Dimension erfolgen kann, gilt natürlich auch bei indexorganisierten Tabellen für alle weiteren Indexe eine mehr oder weniger vorhandene Übereinstimmung der Ordnung von Index und Daten. Auskunft über den Grad der Übereinstimmung der Ordnung eines Index und der Sätze, auf die die Index-Einträge verweisen, gibt der im eInformationsschema in normalisierter Form gespeicherte Clustering Factor. Zur Abschätzung des bei einem *Index Scan* durch die Nichteinhaltung interner Sortierreihenfolgen entstehenden Mehraufwands (*Gleichung D.10*) wird aber die Anzahl auftretender Blockwechsel (N_{BW}) benötigt. Diese kann aus dem Clustering Factor mit Gleichung C.1 erfolgen.

$$M_{P_{IScanSort}} = \left(\frac{N_{BW}}{P_{USED}} - 1 \right) \cdot (1 - P_{BR}) \cdot 100 \qquad \text{Gleichung D.10}$$

Diese Gleichung wurde wiederum anhand einer Messreihe überprüft. Die Ergebnisse zeigt *Abbildung D.2.* Zunächst wurde eine Tabelle mit einem Index angelegt und gezielt Tupel so eingefügt, dass die interne Sortierreihenfolge der Sätze dem Ordnungskriterium des Index vollständig entspricht. Die anschließend bei der Ausführung eines Index Scan ermittelten Aufwandswerte dienen wieder als Basiswerte. An allen weiteren Messpunkten wurden Tabelle und Index jeweils neu angelegt. Allerdings wurden die einzufügenden Testdaten jetzt so erzeugt, das sich von Messpunkt zu Messpunkt ein schlechterer Clustering Factor ergibt. Nach dem

Einfügen der jeweiligen Daten wurden wieder Index Scans ausgeführt und der jeweils angefallene I/O-Aufwand festgehalten.

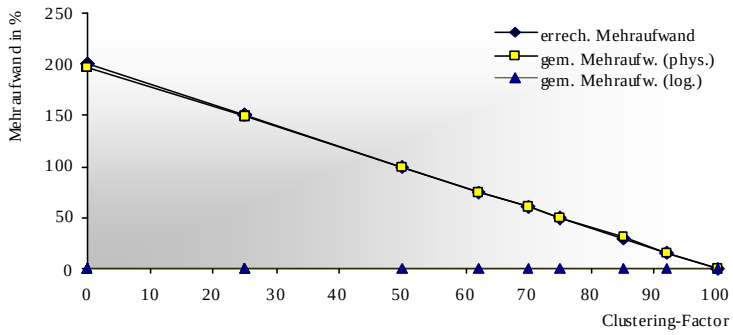


Abbildung D.2: Auswirkungen der Veränderung des Clustering Factors auf Index Scans, basierend auf gemessenen physischen und logischen Blockzugriffen

Aus Abbildung D.2 ist auch ersichtlich, dass eine Veränderung des Clustering Factors keinerlei Einfluss auf die Anzahl logischer Blockzugriffe hat, da diese ja von der Anzahl zu lesender Sätze abhängig ist. Eine Erhöhung des I/O-Aufwands ergibt sich hier dadurch, dass die Blockwechsel eine Erhöhung der physischen Blockzugriffe bewirken.

Bei verbesserten Implementierungen von Index Scans (wie in neueren Versionen des DBMS-Produkts Oracle) werden aufeinanderfolgende Sätze mit einem Zugriff gelesen, wenn sie sich in einem Datenblock befinden. Hier tritt auch ein Mehraufwand bezüglich der Anzahl logischer Blockzugriffe auf. Dieser kann ebenfalls mit Gleichung D.10 näherungsweise bestimmt werden, wenn der Wert für P_{BR} auf 1 gesetzt wird.

An dieser Stelle soll nochmals darauf hingewiesen werden, dass die eben betrachteten Möglichkeiten zur Mehraufwandsabschätzung verwendet werden können, wenn zur Erzeugung der sortierten Treffermenge Index (Range) Scans eingesetzt werden. Wird die Tabelle sequenziell gelesen und werden anschließend die gelesenen Tupel sortiert, so ist der durch eine schlechte Vorsortierung anfallende Mehraufwand von der konkreten Implementierung des Sortieralgorithmus abhängig. Die dabei wirkende große Menge an Einflussgrößen ist aber allgemein gültig kaum zu erfassen und wird hier deshalb nicht berücksichtigt.

Auch wenn über einen Sekundärschlüsselindex, insbesondere wenn dieser eine niedrige Selektivität besitzt, auf alle Sätze mit einem bestimmten Schlüsselwert bzw. Schlüsselwerten aus einer Werteliste zugegriffen werden soll, ist eine intern sortierte Speicherung vorteilhaft. Auch hier erhöht sich die Zahl der anfallenden physischen Blockzugriffe, wenn Sätze mit gleichem Schlüsselwert verstreut abgespeichert werden. Eine strenge Sortierung ist in diesem Falle nicht einmal erforderlich, wichtig ist nur, dass Sätze mit gleichem Schlüsselwert physisch beieinander gespeichert werden. Da Zugriffe auf Sätze mit einem bestimmten Schlüsselwert (über *Index Lookup*) im Prinzip einen Ausschnitt eines vollständigen Index Scan darstellen, kann die Mehraufwandsabschätzung mit Gleichung D.10 erfolgen.

Auswirkungen der Indexhöhe

Auch die Ebenenzahl eines Index hat Einfluss auf die Anzahl der notwendigen Blockzugriffe bei der Suche nach Tupeln unter Verwendung des Index. Eine Verringerung der Anzahl der Ebenen bringt auch eine Verringerung der notwendigen Blockzugriffe mit sich. In Anlehnung an [IBM02] kann, ausgehend von der derzeit aktuellen Anzahl Ebenen des Index ($H=I_{LEV}$) geprüft werden, ob eine Ebene eingespart werden kann. Dazu werden weiterhin die Anzahl möglicher Verweise auf Sohnknoten im Idealfall (NL_{ideal} - *Gleichung B.12*) in den inneren Ebenen und die Anzahl der mindestens notwendigen Blätter (NP_B – *Gleichung B.11*) benötigt. Dabei wird über die Anzahl Verweise in der Ebene *H*-2 (NL^{H-2}) die in der Ebene *H*-1 maximal unterzubringende Anzahl an Blöcken berechnet. Diese wird, wie in (*Un*)*Gleichung D.11* gezeigt, ins Verhältnis zur in der Blattebene mindestens benötigten Blockanzahl gesetzt.

$$\frac{NL^{H-2}}{NP_B} \geq 1 \qquad \text{Gleichung D.11}$$

Ergibt sich dabei ein Wert größer oder gleich 1, so kann im Idealfall durch eine Reorganisation die Höhe des Index um eine Ebene verringert werden, denn auf der Ebene *H*-1 können genügend Indexblöcke für die zukünftige Blattebene zur Verfügung gestellt werden. Der Mehraufwand, der durch die eigentlich nicht benötigte Indexebene entsteht, ist von der Höhe des Indexbaums abhängig und kann mit *Gleichung D.12* bestimmt werden.

$$M_{LIHöhe} = \left(\frac{I_{LEV}}{I_{LEV} - 1} \text{-} 1 \right) \cdot 100 \qquad \text{Gleichung D.12}$$

Vor allem nach Massenlöschungen sollte geprüft werden, ob eine Reduzierung der Höhe der betroffenen Indexbäume möglich ist. Dies kann u.U. erhebliche Einsparungen an I/O-Operationen bewirken. Die Einbeziehung der Einflüsse der Datenbankpufferung in diese Gleichung ist schwierig, da in diesem Fall davon ausgegangen werden muss, dass sich die Pufferungsgrade der einzelnen Indexebenen nach der Reorganisation ändern.

Auswirkungen ungültiger Blockadressen in Sekundärindexen indexorganisierter Tabellen

Durch das Verschieben von Sätzen auf der Blattebene von indexorganisierten Tabellen werden die jeweiligen *Blockadressen ungültig,* die in eventuell existierenden Sekundärindexen gespeichert werden. Damit kann bei Suchen über Sekundärindexe nur durch zusätzliches Durchmustern des Hauptindex der IOT auf die verschobenen Datensätze zugegriffen werden. Der dabei anfallende Mehraufwand entspricht dem Aufwand für das Durchsuchen des Hauptindex, abzüglich des Zugriffs auf die die Daten enthaltenden Blattknoten, da diese auch bei Zugriffen über die Blockadresse anfallen. Die Anzahl der ungültigen Blockadressen kann über den in

I_{DIR_ACC} gespeicherten Anteil gültiger Adressen ermittelt werden. Die Abschätzung des bei *Index Lookups* anfallenden Mehraufwands kann mit *Gleichung D.13* erfolgen.

$$M_{LILookupIOTSK} = \frac{\overbrace{\underbrace{\left(1-\frac{I_{DIR\_ACC}}{100}\right)}_{\text{ungült. Blockadressen}} . P_{ANZT} / I_{UNQSK} \cdot \left(I_{LEVIOT}-1\right)}^{\text{Durchsuchen des Hauptindex der IOT}}}{\underbrace{I_{UNQSK} \cdot \left(\underbrace{\left(I_{LEVSK}-1\right)}_{\text{innere Ebenen}} + \underbrace{\max\left(I_{LEAFSK} / I_{UNQSK}, 1\right)}_{\text{Blattebene}} \right)}_{\text{Sekundärindex}} + \underbrace{P_{ANZT}}_{\text{direkte Datenzugriffe}} + \underbrace{P_{OVFL}}_{\text{Überlaufsätze}}}$$

Gleichung D.13

Die Erweiterung zur Berücksichtigung der Einflüsse der Datenbankpufferung führt zu *Gleichung D.14.*

$$M_{PILookupIOTSK} =$$

$$\frac{\overbrace{\underbrace{\left(1-\frac{I_{DIR\_ACC}}{100}\right)}_{\text{ungült. Blockadressen}} . P_{ANZT} / I_{UNQSK} \cdot \sum_{i=1}^{ILEVIOT-1}\left(1-I_{BRiIOT}\right)}^{\text{Durchsuchen des Hauptindex der IOT}}}{\underbrace{I_{UNQSK} \cdot \left(\sum_{i=1}^{ILEVSK-1}\left(1-I_{BRiSK}\right) + \underbrace{\max\left(I_{LEAFSK} / I_{UNQSK}, 1\right)}_{\text{Blattebene}} \cdot \left(1-I_{BRhSK}\right) \right)}_{\text{Sekundärindex}} + \underbrace{P_{ANZT} \cdot \left(1-I_{BRhIOT}\right)}_{\text{direkte Datenzugriffe}} + \underbrace{P_{OVFL} \cdot \left(1-P_{BR}\right)}_{\text{Überlaufsätze}}}$$

Gleichung D.14

Ungültige Blockadressen verursachen auch bei Bereichssuchen eine Aufwandserhöhung, die mit *Gleichung D.15* quantifiziert werden kann.

$$M_{LILookupIOTSK} = \frac{\overbrace{\underbrace{\left(1-\frac{I_{DIR\_ACC}}{100}\right)}_{\text{ungült .Blockadressen}} . P_{ANZT} / I_{UNQSK} \cdot \left(I_{LEVIOT}-1\right)}^{\text{Durchsuchen des Hauptindex der IOT}}}{\underbrace{\left(I_{LEVSK}-1\right)+I_{LEAFSK}}_{\text{Sekundärindex}} + \underbrace{P_{ANZT}}_{\text{direkte Datenzugriffe}} + \underbrace{P_{OVFL}}_{\text{Überlaufsätze}}}$$

Gleichung D.15

Durch die Erweiterung um die Pufferungsgrade der betrachteten Datenbankobjekte ergibt sich *Gleichung D.16*.

Werden die betroffenen Sekundärindexe neu aufgebaut, so werden in alle Verweise wieder gültige Blockadressen eingetragen und der Mehraufwand entfällt für einen gewissen Zeitraum. Oracle bietet mit `ALTER INDEX ... UPDATE BLOCK REFERENCES` auch eine Anweisung zur Aktualisierung der Blockverweise im Online-Betrieb.

$$M_{PIScanIOTSK} = \frac{\overbrace{\overbrace{\left(1-\frac{I_{DIR\_ACC}}{100}\right)}^{ungült.\ Blockadressen} . P_{ANZT} / I_{UNQSK} \cdot \sum_{i=1}^{I_{LEVIOT}-1}\left(1-I_{BRiIOT}\right)}^{Durchsuchen\ des\ Hauptindex\ der\ IOT}}{\underbrace{\sum_{i=1}^{I_{LEVSK}-1}\left(1-I_{BRiSK}\right)+I_{LEAFSK}\cdot\left(1-I_{BRhSK}\right)}_{Sekundärindex}+\underbrace{P_{ANZT}\cdot\left(1-I_{BRhIOT}\right)}_{direkte\ Datenzugriffe}+\underbrace{P_{OVFL}\cdot\left(1-P_{BR}\right)}_{Überlaufsätze}}$$

Gleichung D.16

Auswirkungen von Extent Interleaving

Wenn mehrere Tabellen innerhalb eines Table Space abgespeichert werden ist es durchaus wahrscheinlich, dass die jeweils zu einer Tabelle gehörenden Extents nicht mehr physisch fortlaufend reserviert werden können. Das heißt, dass die Extents mehrerer Tabellen physisch gemischt abgespeichert werden. Damit kommt es zu einer Fragmentierung (Extent Interleaving), ohne dass Freiplatz eingestreut sein muss. Beim sequenziellen Durchsuchen der Tabelle ist jedes Mal beim Übergang zu einem neuen Extent eine Positionierungsoperation der Lese-/Schreibköpfe nötig. Diese können (zumindest teilweise) bei einer physisch fortlaufenden Speicherung der Extents einer Tabelle eingespart werden. Der durch diese Fragmentierungsart entstehende Mehraufwand kann abgeschätzt werden, indem man zum Aufwand für das sequenzielle Durchmustern den Positionierungsaufwand addiert und den Wert mit dem Aufwand ohne die Positionierungsoperationen ins Verhältnis setzt. Allerdings ist hier die genaue Kenntnis der physischen Eigenschaften der Speichermedien notwendig. Informationen darüber werden derzeit nicht im eInformationsschema erfasst. Aufgrund der starken Hardwareabhängigkeit sind allgemein gültige Aussagen hier nur sehr ungenau. Deshalb wird an dieser Stelle auf die Angabe einer Berechnungsformel verzichtet. Hier ist es eher sinnvoll, dem Extent Interleaving durch eine besonders überlegte Wahl der Extent-Größen vorzubeugen.

Anhang E – Quelltext Reorganisation in eine Kopie

```
-- Kreieren der Interim-Tabelle

CREATE TABLE interim(
       MSISDN        NUMBER,
       name          VARCHAR2(35),
       vorname       VARCHAR2(35),
       gebdat        DATE,
       strasse       VARCHAR2(50),
       plz           CHAR(5),
       ort           VARCHAR2(50),
       MSRN          VARCHAR2(20),
       LA            NUMBER);


-- Kreieren der Log-Tabelle

CREATE MATERIALIZED VIEW LOG ON Teilnehmer;


-- Kopieren der Daten in die Interim-Tabelle

INSERT /*+ APPEND */ INTO interim SELECT /*+ FULL */ * FROM Teilnehmer;


-- Erzeugen der materialisierten Sicht über den Strukturen der Interim-Tabelle

CREATE MATERIALIZED VIEW interim ON PREBUILT TABLE WITH REDUCED PRECISION
REFRESH FAST ON DEMAND WITH PRIMARY KEY AS SELECT * FROM Teilnehmer;


-- Aufbau des Primaerschluesselindexes

CREATE UNIQUE INDEX interim_pk ON interim (MSISDN);


-- Erzeugen weiterer Indexe und Hinzufügen von Constraints

CREATE INDEX I_Name_interim ON interim (Name);


-- Anwendung von Änderungsoperationen auf die Originaltabelle zur
-- „Simulation“ eines parallel laufenden Datenbankbetriebs

-- UPDATE Teilnehmer SET name='albert' WHERE MSISDN=1;
-- DELETE FROM Teilnehmer WHERE MSISDN=12345;
-- ...


-- Synchronisieren von Original- und Interim-Tabelle

EXECUTE DBMS_MVIEW.REFRESH('INTERIM','F');


-- Abschluss der Reorganisation

LOCK TABLE Teilnehmer IN EXCLUSIVE MODE;

LOCK TABLE interim IN EXCLUSIVE MODE;


-- Abschließendes Synchronisieren der Tabellen

EXECUTE DBMS_MVIEW.REFRESH('INTERIM','F');


-- Loschen der Log-Tabelle und der materialisierten Sicht

DROP MATERIALIZED VIEW LOG ON Teilnehmer;

DROP MATERIALIZED VIEW interim;
```

```
-- Tausch der Namen

-- Intern wird dies über die Manipulation von Katalogeinträgen realisiert.
-- Damit ist auch ein Umbenennen von Indexen und Constraints möglich, was mit
-- der Anweisung RENAME nicht möglich ist. Die Anweisungen sind aufgeführt,
-- um das Umbenennen anzudeuten. Natürlich könnten Indexe, Constraints etc.
-- auch erst nach dem Löschen der alte Tabelle und dem Umbenennen der
-- Interim-Tabelle kreiert und aufgebaut werden, aber dann wäre die Phase zum
-- Abschluss der Reorganisation, in der kein Zugriff auf die Daten möglich
-- ist, deutlich länger.


--RENAME I_Name_Teilnehmer TO I_tmp_Name_Teilnehmer;
--RENAME I_pk_tn TO tmp_Teilnehmer_pk;
RENAME Teilnehmer TO tmp_Teilnehmer;

--RENAME I_Name_interim TO I_Name_Teilnehmer;
--RENAME interim_pk TO i_pk_tn;
RENAME interim TO Teilnehmer;

LOCK TABLE Teilnehmer IN SHARE UPDATE MODE;

--RENAME I_tmp_Name_Teilnehmer TO I_Name_interim;
--RENAME tmp_Teilnehmer_pk TO interim_pk;

RENAME tmp_Teilnehmer TO interim;

-- Loeschen der alten Tabelle

DROP TABLE interim;
```

Anhang F – Messreihen zur Schätzung der Reorganisationskosten

Dieser Anhang enthält weitere Informationen zu den in *Abschnitt 7.3* angestellten Beispielrechnungen zur Schätzung der mit der Durchführung von Datenbankreorganisationen verbundenen Kosten sowie Ergebnisse durchgeführter Messreihen.

Basis der Messreihen

Grundlage für die Beispielrechnungen und die unter Nutzung von Oracle 10g durchgeführten Aufwandsmessungen ist die Tabelle `TEILNEHMER` des Beispielschemas aus *Abschnitt 6.3.2.6*. Das Attribut `MSISDN` stellt den Primärschlüssel der Tabelle dar. Um das Verhalten der von Oracle implementierten Vorgehensweisen bei In-Place-Reorganisationen und bei der Synchronisation von Original- und Interim-Tabelle im Rahmen von Reorganisationen in eine Kopie nachvollziehen zu können, wurden in verschiedenen Messreihen noch weitere Indexe über den Attributen `Name` und `Ort` aufgebaut.

Jede Messreihe basiert auf einer neu angelegten Tabelle, die mit per Zufallszahlengenerator erzeugten Beispieldaten geladen wurde. Im Rahmen des Ladens erfolgt auch ein Neuaufbau der Indexe und eine Aktualisierung der entsprechenden Statistikdaten. Für jede Messreihe wurden die gleichen Beispieldaten verwendet. Nach dem Laden enthielt die Tabelle jeweils **104341** Tupel mit einer durchschnittlichen Satzlänge von jeweils **149** Byte. Bei der verwendeten Blockgröße der Datenbank von **4096** Byte stehen für die Speicherung von Datensätzen und deren Verwaltungsinformationen ca. **4000** Byte je Block zur Verfügung. Hauptziel der im Rahmen der Messreihen durchgeführten Reorganisationen war die Beseitigung von eingestreutem Freiplatz. Der Parameter `PCTFREE` wurde vor jeder Reorganisation auf den Wert **0** gesetzt. Das heißt, dass bei der Reorganisation kein Speicher für tupelverlängernde Änderungsoperationen freigehalten wird. Damit können in einem Datenblock ca. **26** Sätze untergebracht werden. Die geschätzte Anzahl Datenblöcke, die die Tabelle nach der Reorganisation belegt (P_{IDEAL}), liegt bei **4013**.

Durch den Neuaufbau der Indexe zu Beginn jeder Messreihe können diese auch schon zu Beginn der Reorganisation als degenerierungsfrei angesehen werden. Indexhöhen sowie die Anzahl der jeweils belegten Blöcke sind daher vor und nach der Reorganisationsdurchführung gleich. *Tabelle F.1* enthält die zu den Indexen gehörenden Statistikdaten.

Um eine Vergleichbarkeit der Ergebnisse der Beispielrechnungen in *Abschnitt 7.3* zu ermöglichen, müssen teilweise die Einflüsse von Datenbankpufferung und der asynchronen Ausführung von Schreiboperationen berücksichtigt werden. Vergleiche ausgeführter logischer Blockzugriffe führen hier nicht zu realistischen Ergebnissen. Die mit der in Abschnitt 5.4.4 vorgestellten Methode ermittelten Pufferungsgrade gelten für den normalen Datenbankbetrieb. Dieser wird aber durch die Reorganisation, zumindest für die in Reorganisation befindlichen Datenbankobjekte,

erheblich gestört. Deshalb fließen in die Beispielrechnungen Annahmen ein, die sich aus den in der Beispielumgebung aufgenommenen Messreihen ergeben. Die entsprechenden Werte sind bei den folgenden Ausführungen zu den einzelnen Verfahren angegeben. Genauere Zusammenhänge, die es evtl. erlauben würden, diese Einflüsse (mit entsprechenden Werten versehen) vorab einzuschätzen, ohne dass für die jeweilige Systemumgebung Erfahrungswerte vorliegen, konnten aus den Messreihen nicht abgeleitet werden. Hier fehlen uns detailliertere Kenntnisse über die genaue Implementierung der wirkenden Mechanismen (bspw. die genauen Auslöser der asynchron ausgeführten Schreiboperationen). Hier könnten sich Anknüpfungspunkte für weitere Untersuchungen ergeben.

Index über Attribut	Anzahl Ebenen	Anzahl Blätter	Anzahl belegter Blöcke
MSISDN	3	447	ca. 450
Name	3	1009	ca. 1020
Ort	3	1236	ca. 1252

Tabelle F.1: Statistikdaten der Indexe

Da die Indexe oberhalb der Blattebene nur über wenige Knoten verfügen, wird in den Kostenschätzungen generell angenommen, dass diese Knoten im Puffer gehalten werden. Die Ungenauigkeiten (Mehrkosten), die durch das erstmalige Laden der entsprechenden Blöcke auftreten, werden vernachlässigt.

Messreihen zur Reorganisation in eine Kopie

Im Rahmen der Messreihen wurden zunächst die Schätzungen für die Kosten zum *Kopieren der Daten* in die Interim-Tabelle und zum *Bottom-Up-Aufbau der* zur Interim-Tabelle gehörenden *Indexe* überprüft.

Dazu wurde in den Datenblöcken der Tabelle `TEILNEHMER` eingestreuter Freiplatz erzeugt. Da Oracle beim Kopieren der Daten die normale Pufferschnittstelle des DBMS teilweise umgeht [Hei05], wurde hier über die von Oracle zur Verfügung gestellte Sicht `V$FILESTAT` der an der Dateischnittstelle angefallene physische I/O-Aufwand (in gelesenen bzw. geschriebenen Blöcken) ermittelt. Die Prefetch-Rate wurde in der Beispielumgebung so eingestellt, dass mit jeder I/O-Operation ein Block gelesen wird. Somit entspricht die gemessene Anzahl Blockzugriffe (zumindest aus Sicht des DBMS) auch der Anzahl I/O-Operationen. Die Zahl der erfolgten logischen Zugriffe auf Blöcke und die Zahl der Blockänderungen an der Pufferschnittstelle wurden über die Sicht `V$SESS_IO` ermittelt.

Eine Zusammenfassung der geschätzten und der gemessenen Werte für das Kopieren der Daten in die Interim-Tabelle zeigt *Tabelle F.2*. Die Anzahl der Blöcke vor der Reorganisation (und damit die erwartete Anzahl Blockzugriffe zum Lesen) wurde den Statistikdaten im Oracle-Katalog entnommen. Der erwartete Schreibaufwand entspricht dem Wert von P_{IDEAL}. Die Messreihen wurden mehrfach wiederholt. Die Messergebnisse waren dabei jeweils nahezu gleich (zu vernachlässigende

Schwankung zwischen **3947** und **3951** traten auf), was sicherlich auch darauf zurückzuführen ist, dass jeweils die gleichen Ausgangsdaten zum Laden der Tabelle verwendet wurden.

PCTFREE	Blöcke vor Reorganisation	Blöcke nach Reorganisation	geschätzte Lesekosten	durchschn. gemessene Lesekosten	geschätzte Schreibkosten	durchschn. gemessene Schreibkosten
50%	8050	4024	8050	8022	4013	4028
70%	13718	4024	13718	13719	4013	4028
90%	49517	4024	49517	49523	4013	4028

Tabelle F.2: Kosten für das Kopieren von Daten

Zum Indexaufbau muss die Interim-Tabelle jeweils einmal gelesen werden und je Indexblock fällt eine Schreiboperation an. Die geschätzten und gemessenen Kostenwerte für den Indexaufbau enthält *Tabelle F.3*.

Indexierungs-schlüssel	belegte Blöcke im Indexsegment	geschätzte Lesekosten	durchschn. gemessene Lesekosten	geschätzte Schreibkosten	durchschn. gemessene Schreibkosten
MSISDN	ca. 450	4013	3946	450	474
Name	ca. 1020	4013	3946	1020	1051
Ort	ca. 1252	4013	3946	1252	1277

Tabelle F.3: Kosten zum Aufbau der Indexe

Zur Überprüfung der Kostenschätzungen für die Anwendung der in der Log-Tabelle aufgezeichneten Änderungen wurden ebenfalls Messreihen aufgenommen. Dazu wurden wieder die Ausgangsdaten in die Tabelle `TEILNEHMER` geladen und die Reorganisation gestartet. Nach dem Aufbau der Indexe wurde zunächst eine sich mit jedem Durchlauf ändernde Anzahl Tupel (10, 100, 1000) in die Originaltabelle nacheinander eingefügt, gelöscht oder geändert. Der Zugriff auf die zu löschenden bzw. zu ändernden Tupel erfolgte jeweils über den Primärschlüssel. Die entsprechenden Primärschlüsselwerte wurden per Zufallszahlengenerator ausgewählt. Anschließend erfolgte die Synchronisation von Originaltabelle und Interim-Tabelle. Dies erfolgt bei Oracle mit der Prozedur `DBMS_SNAPSHOT.REFRESH(...)`. Die gemessenen Aufwandswerte gelten daher jeweils für eine vollständige Synchronisation. Damit kann zwar ermittelt werden, wie viele Blöcke im Rahmen der Abarbeitung der gesamten Prozedur gelesen oder geändert wurden, allerdings stehen keine direkten Informationen über die während der Ausführung von Einzeloperationen anfallenden Blockzugriffe zur Verfügung. Zunächst erfolgte aus Gründen der Nachvollziehbarkeit eine Messung der angefallenen logischen Blockzugriffe. Die Messreihen wurden jeweils mit ein, zwei und drei Indexen durchgeführt. Die Indexierung erfolgte dabei zunächst nur über dem Primärschlüssel (`MSISDN`), danach über dem Primärschlüssel und dem Attribut `Name` und

abschließend wurde über dem Primärschlüssel, dem Attribut `Name` und dem Attribut `Ort` jeweils ein Index erzeugt.

Die ermittelten Messwerte für das Ausführen von nur 10 Operation im Rahmen der Synchronisation (`INSERT, UPDATE, DELETE`) weichen erwartungsgemäß stark von denen der übrigen Durchläufe ab. Als Ursache für diese Ungenauigkeit wird der im Vergleich zur ausgeführten Operation sehr hohe Overhead der Prozedur `DBMS_SNAPSHOT.REFRESH(...)`angenommen. Um dessen Einfluss zu verringern, wurden bei den nachfolgenden Durchläufen jeweils die Differenz zu den im vorherigen Lauf ermittelten Werten gebildet und durch die Differenz der Anzahl Operationen im aktuellen und vorherigen Lauf geteilt. Zur besseren Nachvollziehbarkeit der Messergebnisse erfolgt zunächst eine getrennte Betrachtung von Einfüge-, Lösch- und Änderungsoperationen. Die Ergebnisse der Messreihen und die erwartete Anzahl logischer Blockzugriffe sind in *Tabelle F.4* dargestellt.

Operation/ Zahl der Indexe	Lesezugriffe (Consistent Mode)	Lesezugriffe (Current Mode)	Änderungen an Blöcken	erwartete Zahl log. Blockzugriffe	erwartete Zahl an Änderungen
INSERT					
1 Index	6,1	5,4	6,4	10	6
2 Indexe	6,1	8,8	8,8	13	8
3 Indexe	6,2	12,25	11,1	16	10
DELETE					
1 Index	6,1	5,3	6,3	10	6
2 Indexe	6,1	8,4	8,3	13	8
3 Indexe	6,1	11,4	10,5	16	10
UPDATE					
1 Index	9,1	2,3	4,2	11	4
2 Indexe	9,1	2,3	4,2	11	4
3 Indexe	9,1	2,2	4,2	11	4

Tabelle F.4: Aufwand zur Synchronisation von Original- und Interim-Tabelle

Datenblöcke können von Oracle in zwei verschiedenen Modi gelesen werden [Ora03b]. Im sog. *Consistent Mode* werden die Inhalte von Datenblöcken in dem zu Beginn einer Suchoperation aktuellen Zustand gelesen. Dazu wird eine von Oracle intern geführte System Change Number (*SCN*) verwendet. Im Rahmen einer Suchoperation wird auf die Datenblöcke zugegriffen, die die zum Start der Suchoperation aktuelle SCN enthalten. Die Inhalte von Blöcken, die mittlerweile geändert wurden und damit eine „jüngere“ SCN enthalten, werden aus den temporären Log-Bereichen rekonstruiert. Die Zusammenstellung der Treffermenge, auf die bspw. eine Löschanweisung angewendet wird, erfolgt im Consistent Mode. Damit werden nur die Tupel gelöscht, die die Suchbedingung der Löschanweisung zu deren Start erfüllen. Eventuell später (während der Abarbeitung der Löschanweisung) eingefügte Tupel sind nicht betroffen, auch wenn sie die Suchbedingung erfüllen

würden. Zum eigentlichen Löschen der Daten werden dann die betroffenen Datenblöcke im aktuellen Zustand (*Current Mode*) gelesen und verändert.

Der Aufwand beim Nacharbeiten der auf die Originaltabelle angewendeten `UPDATE`-Anweisungen ist unabhängig von der Anzahl der Indexe. Dies liegt daran, dass in den durchgeführten Messreihen die Werte indexierter Attribute nicht geändert wurden. Die relativ hohe Anzahl Lesezugriffe im Consistent Mode lässt sich damit erklären, dass neben dem Zugriff auf die Log-Tabelle auch auf die Originaltabelle zugegriffen werden muss, um den Datensatz im aktuellen Zustand (quasi als Image) zu lesen. Dieser Zugriff erfolgt über den Primärschlüsselindex der Originaltabelle. Der Index besaß eine Höhe von drei Ebenen. Hinzu kommt der Zugriff auf den entsprechenden Datenblock. Anschließend muss auf den Satz in der Interim-Tabelle zugegriffen werden. Der Zugriff erfolgt ebenfalls über den vorhandenen Primärschlüsselindex und (wie bei der Zusammenstellung von Treffermengen üblich) im Consistent Mode. Damit ergeben sich **9** Zugriffe. Anschließend wird der entsprechende Block im Current Mode gelesen und geändert. Der zweite Zugriff im Current Mode erfolgt vermutlich auf den den aktuellen Eintrag enthaltenden Block der Log-Tabelle, damit der Eintrag gelöscht werden kann. Somit ergeben sich insgesamt **11** Lesezugriffe. Die vier Blockänderungen fallen zum Ändern des Satzes in der Interim-Tabelle, zum Löschen des Eintrags aus der Log-Tabelle und zum Erzeugen der beiden Einträge im temporären Log-Bereich des DBMS an.

Die Annahme, dass je geändertem Daten- bzw. Indexblock auch ein Zugriff auf den temporären Log-Bereich erfolgt, liegt darin begründet, dass Oracle vor der Ausführung von Änderungen den alten Zustand (*Before Images*) von Datensätzen (und Indexeinträgen) bis zum Abschluss der jeweiligen Transaktion sichert. Diese Before Images werden im Falle des Abbruchs von Transaktionen verwendet, um den alten Zustand von Daten- und Indexblöcken wiederherstellen zu können. Dabei wird je erzeugtem Eintrag ein logischer Blockzugriff gezählt.

Zum Nachvollziehen einer `INSERT`-Anweisung muss ebenfalls der entsprechende Satz der Originaltabelle gelesen werden. Einschließlich des Zugriffs auf die Log-Tabelle sind hier **5** Zugriffe im Consistent Mode zu erwarten. Zum Einfügen in die Interim-Tabelle und zum Löschen des aktuellen Eintrags aus der Log-Tabelle müssen **2** Blöcke im Current Mode gelesen werden. Hinzu kommen im Rahmen des Einfügens bzw. Aktualisierens von Indexeinträgen **3** Zugriffe pro Index zum Auffinden des jeweiligen Blatts. Neben den Blöcken der Log-Tabelle und der Interim-Tabelle wird je Index i.d.R. das entsprechende Blatt geändert. Für jeden geänderten Block erfolgt noch das Erzeugen der Einträge im temporären Log-Bereich.

Zum Nacharbeiten in der Log-Tabelle vermerkter `DELETE`-Anweisungen wird zunächst im Consistent Mode die zu löschende Treffermenge ermittelt. Da die Synchronisation zwischen Originaltabelle und Interim-Tabelle über den Primärschlüssel erfolgt, muss der Primärschlüsselindex der Interim-Tabelle von der Wurzel bis zum jeweiligen Blatt durchsucht werden. Anschließend wird auf den entsprechenden Datenblock zugegriffen. Dafür fallen insgesamt **4** Blockzugriffe an. Hinzu kommt ein Zugriff im Consistent Mode auf den aktuellen Eintrag in der Log-Tabelle. Anschließend wird auf den Datenblock der Interim-Tabelle und den den

aktuellen Eintrag enthaltenden Block der Log-Tabelle noch einmal im Current Mode zugegriffen, damit deren Inhalte geändert werden können. Da auch die Indexeinträge, die auf den zu löschenden Satz verwiesen haben, gelöscht bzw. aktualisiert werden müssen, erfolgen je Index **3** weitere Zugriffe. Anschließend werden die Datenblöcke sowie die Blätter der Indexe geändert und jeweils die Einträge im temporären Log-Bereich des DBMS vermerkt.

Als Ursache für die vorhandenen Differenzen zwischen den gemessenen und den erwarteten Aufwandswerten in Tabelle F.4 werden Katalogzugriffe, Zugriffe für Änderungen an Segmentverwaltungsblöcken etc. vermutet. Informationen zu weiteren Details der Implementierung liegen uns hier von Herstellerseite nicht vor. Hier wären noch weitere Untersuchungen zum besseren Verständnis bzw. zur Absicherung der Vermutungen nötig. Für die Hersteller der DBMS-Produkte sollte es allerdings unproblematisch sein, solches Detailwissen noch einfließen zu lassen und somit für das eigene Produkt zu genaueren Abschätzungen zu gelangen.

Mit der implementierten Vorgehensweise werden während der Ausführung von Änderungsanweisungen mehr logische Blockzugriffe vorgenommen, als in den Kostenfunktionen berücksichtigt. Ziel der präsentierten Kostenfunktionen ist es aber, unter Berücksichtigung der Einflüsse der Datenbankpufferung (soweit möglich), die Anzahl der während der Ausführung eines Planoperators anfallenden I/O-Operationen zu ermitteln, d.h. die Anzahl der von Datenträgern lesenden und auf Datenträger schreibenden Blockzugriffe näherungsweise zu ermitteln. Die Zahl dieser Operationen dürfte sich durch die bei Oracle verwendete Vorgehensweise allerdings kaum erhöhen. Die Basis dafür bildet die Annahme, dass Datenblöcke nach einem evtl. notwendigen Lesen im Consistent Mode für die Zugriffe im Current Mode nahezu sicher im Puffer vorliegen, da die Zugriffe innerhalb von kurzen Zeiträumen erfolgen. Blöcke, deren Inhalte im Consistent Mode aus dem temporären Log rekonstruiert wurden, müssen für die Zugriffe im Current Mode evtl. von den entsprechenden Datenträgern gelesen werden. Der Anteil solcher Blöcke wird hier aber als eher gering angenommen und bei den Aufwandsschätzungen vernachlässigt.

Zur Abschätzung der Anzahl physischer I/O-Operationen fließen im Rahmen der in der Beispielumgebung aufgenommenen Messreihen ermittelte durchschnittliche Werte für den jeweiligen Pufferungsgrad der Daten- und Indexblöcke von Original- und Interim-Tabelle in die Beispielrechnungen in Abschnitt 7.3 und in die Berechnungen ein, die die Grundlage der Werte in Tabelle F.6 bilden. Auch bezüglich des Einflusses asynchroner Schreiboperationen wurden im Rahmen der Messreihen entsprechende Durchschnittswerte ermittelt. Zur Bestimmung der Werte wurden **5200** `INSERT`-, **3400** `UPDATE`- und **1800** `DELETE`-Anweisungen ausgeführt, die anschließend an der Interim-Tabelle nachvollzogen werden mussten. Bezogen auf die Zahl der Tupel der Tabelle bedeutet dies einen Anteil von **10%**. Dieser im Vergleich zu den Ausführungen in Abschnitt 7.3 höhere Anteil wurde gewählt, um die relative Schwankungsbreite der Messwerte zu verringern. In den Beispielrechnungen wird ein Anteil von **1%** Änderungsoperationen (bezogen auf die Zahl der Tupel der Tabelle) angenommen, was nach unserer Auffassung eher der Realität entsprechen dürfte.

In *Tabelle F.5* ist angegeben, mit welcher Wahrscheinlichkeit ein im Rahmen einer Leseoperation benötigter Block im Puffer gefunden wird. Weiterhin ist der Anteil tatsächlich in die entsprechenden Dateien geschriebener Blöcke an der Zahl der an Blöcken vorgenommenen Änderungen angegeben. Bei der Aufnahme der Messreihen waren alle Indexe in der gleichen Datei gespeichert. Deshalb gelten hier für alle drei Indexe die gleichen Werte.

	Originaltabelle	Interim-Tabelle
Pufferungsgrad der Indexe	0,93	0,83
Pufferungsgrad der Datenblöcke	0,61	0,66
Anteil geschriebener Indexblöcke	0	0,23
Anteil geschriebener Datenblöcke	0	0,39

Tabelle F.5: Messwerte zu Pufferungsgraden und den Auswirkungen asynchronen Schreibens bei Reorganisationen in eine Kopie in der Beispielumgebung

Der während der Messreihen erwartete Aufwand wurde vorab, wie in Abschnitt 7.2.2 beschrieben, berechnet und mit dem auf der Dateischnittstelle der die Daten- und Indexsegmente enthaltenden Table Spaces gemessenen Schreib- und Leseaufwand verglichen. *Tabelle F.6* zeigt eine Gegenüberstellung der errechneten und der gemessenen Werte für Synchronisation von Original- und Interim-Tabelle. Während der Messreihen war jeweils den Attributen `MSISDN`, `Name` und `Ort` ein Index definiert.

	Originaltabelle (gemessen)	Originaltabelle (errechnet)	Interim-Tabelle (gemessen)	Interim-Tabelle (errechnet)
gelesene Datenblöcke	3360	3354	3565	3536
gelesene Indexblöcke	588	602	3940	4148
geschriebene Datenblöcke	0	0	4066	4056
geschriebene Indexblöcke	0	0	4790	4830

Tabelle F.6: Vergleich errechneter und gemessener Aufwandswerte für die Synchronisation von Original- und Interim-Tabelle

Datenbanksystembasierte Reorganisationsmethode

Die im Rahmen der datenbanksystembasierten Reorganisationsmethode genutzten Werkzeuge verwenden beim Datenexport und -import sowie beim Aufbau von Indexen die gleichen Vorgehensweisen [Hei05], wie sie auch bei der Reorganisation in eine Kopie im Rahmen des Kopierens von Daten in die Interim-Tabelle bzw. beim Aufbau der zur Interim-Tabelle gehörenden Indexe angewendet werden. Die in Tabelle F.2 und Tabelle F.3 angegebenen Werte sind daher im Wesentlichen übertragbar. Die tatsächliche Größe der Export-Datei lag bei ca. **15,3** Megabyte, die errechnete Größe bei ca. **15,7** Megabyte. Auch bei der Erstellung von Backups wird

davon ausgegangen, dass die durchzuführenden sequenziellen Leseoperationen auf die gleiche Weise durchgeführt werden, wie beim Export der Daten bzw. beim Kopieren in die Interim-Tabelle.

Messreihen zu In-Place-Reorganisationen

Zur Überprüfung des Berechnungsschemas zur Schätzung der bei In-Place-Reorganisationen anfallenden Kosten wurden (ebenfalls unter Nutzung von Oracle 10g) Untersuchungen durchgeführt und Messreihen aufgenommen.

Die Rahmenbedingungen (Tupelzahlen, Blockgröße etc.) entsprachen dabei im Wesentlichen den oben beschriebenen. Ausgangspunkt war wieder die Tabelle `TEILNEHMER`. Vor jeder Messreihe erfolgte ebenfalls das Laden der Beispieldaten in die Tabelle. Dabei wurde für den Parameter `PCTFREE` mit einem Wert von **0** gearbeitet. Wird beim Laden der Daten Speicher (durch entsprechendes Setzen von `PCTFREE`) freigehalten, so wird der ursprüngliche Parameterwert (selbst nach einer entsprechenden Änderung vor Beginn der Reorganisation) für die bereits reservierten Speicherbereiche auch während der In-Place-Reorganisation berücksichtigt. Zum Erzeugen von eingestreutem Freiplatz wurden in den verschiedenen Messreihen nacheinander die Längen der Werte der Zeichenkettenattribute (Straße, Vorname, MSRN) jeweils auf ein Zeichen verkürzt. Die sich dadurch für die verschiedenen Messreihen ergebenden unterschiedlich hohen Anteile eingestreuten Freiplatzes haben Einfluss darauf, wie viele Datensätze im Rahmen der Reorganisation verschoben werden können. Auf den zum Verschieben eines Satzes zu betreibenden Aufwand hat der Freiplatzanteil keinen direkten Einfluss. Dies zeigte sich auch in den durchgeführten Messreihen.

Oracle greift zur Umsetzung der Reorganisationsfunktionalität zu großen Teilen auf die vom System auch für `DELETE`- und `INSERT`-Anweisungen verwendeten Implementierungen zurück. Das hat den Vorteil, dass (soweit nötig) auch vorhandene Mechanismen zur Integritätssicherung verwendet werden können. Gegenüber den einzeln ausführbaren `DELETE`- und `INSERT`-Anweisungen wurden aber anscheinend gewisse Optimierungen vorgenommen. So deuten die Ergebnisse der aufgenommenen Messreihen darauf hin, dass die Segmentverwaltungsdaten bspw. nicht nach jeder Lösch- bzw. Einfügeoperation aktualisiert werden.

Unter Oracle wird eine In-Place-Reorganisation mit der Anweisung

```
ALTER TABLE <Tabellenname> SHRINK SPACE
```

durchgeführt. Auch hier stehen, ähnlich wie bei den für die Synchronisation von Original- und Interim-Tabelle durchgeführten Messreihen, keine Informationen über die während der Ausführung von Einzeloperationen angefallenen Blockzugriffe zur Verfügung. Es kann zunächst lediglich ermittelt werden, wie viele Blöcke im Rahmen der gesamten Reorganisation gelesen oder geändert wurden.Die Messreihen wurden jeweils mit ein, zwei und drei Indexen durchgeführt, um das Verhalten der von Oracle verwendeten Implementierung besser nachvollziehen zu können. Bei einer weiteren Messreihe wurden vor der Durchführung der Reorganisation alle Indexe gelöscht. Durch die Ermittlung der während der Reorganisation durchgeführten

logischen Blockzugriffe wird die implementierte Vorgehensweise in Teilen nachvollziehbar. *Tabelle F.7* zeigt die Ergebnisse der aufgenommenen Messreihen bezogen auf die zum Verschieben eines Datensatzes erfolgten logischen Blockzugriffe.

Zahl der Indexe	gelesene Blöcke	geänderte Blöcke
0	2,9	4,4
1	9,1	8,7
2	15,2	13,7
3	21,5	18,8

Tabelle F.7: Ergebnisse der zu In-Place-Reorganisationen durchgeführten Messreihen

Die Ergebnisse der Messreihe deuten ebenfalls darauf hin, dass die Reorganisation bei Oracle nicht über eigens für Reorganisationszwecke implementierte Verschiebeoperationen von Datensätzen realisiert wird. Bei solchen Verschiebeoperationen, die in der Literatur auch häufig zur Implementierung der Reorganisationsfunktionalität vorgeschlagen werden (vgl. auch *Kapitel 4*), wird ein Datensatz aus einem Datenblock entfernt und in einen anderen Block eingefügt. Anschließend werden die in vorhandenen Indexeinträgen enthaltenen Verweise auf den verschobenen Datensatz aktualisiert. Dazu muss auf die betroffenen Indexe je Eintrag nur einmal zugegriffen werden. Die Durchführung von Kostenschätzungen hierfür ist in *Abschnitt 7.2.3* beschrieben, ebenso wie für die von Oracle verwendete Implementierung. Wie die Messreihen ergaben, erfolgten lesende Zugriffe auf Daten- und Indexblöcke nahezu ausschließlich im Current Mode. Aufgeteilt auf die zu verschiebenden Sätze, erfolgten bei drei vorhandenen Indexen durchschnittlich lediglich 0,38 Zugriffe je Satz im Consistent Mode.

Die Oracle-Implementierung, bei der das Verschieben eines Satzes über das Ausführen einer nahezu vollständigen `DELETE`- und einer `INSERT`-Operation realisiert wird, erscheint da zunächst aufwendiger. Dies äußert sich auch in der hohen Anzahl gemessener logischer Blockzugriffe. *Tabelle F.8* zeigt eine vermutliche Verteilung der logischen Blockzugriffe auf einzelne (Teil-)Operationen, die sich aus der Auswertung der durchgeführten Messreihen ergibt.

Auch hier wird für jeden im temporären Log-Bereich erzeugten Eintrag ein logischer Blockzugriff gezählt. Messungen der auf der Dateischnittstelle des die Rollback-Segmente enthaltenden Table Space lassen vermuten, dass im Rahmen der durchgeführten Messreihen für jeden vollständig bearbeiteten Datensatz bei Vorhandensein

- eines Indexes **0,12**,
- beim Vorhandensein von zwei Indexen **0,18** und
- beim Vorhandensein von drei Indexen **0,25**

Blöcke geschrieben wurden.

	zu lesende Daten-blöcke	je Index zu lesende Indexblöcke	zu ändernde Datenblöcke	je Index zu ändernde Indexblöcke	Zugriffe auf temp. Log-Bereich	Summe Lesezu-griffe	Summe Schreib-zugriffe
DELETE	1	3	1	1	2		
INSERT	1	3	1	1	2		
Summe ohne Ind.	2	-	2	-	2	**2**	**4**
Summe 1 Index	2	6	2	2	4	**8**	**8**
Summe 2 Indexe	2	12	2	4	6	**14**	**12**
Summe 3 Indexe	2	18	2	6	8	**20**	**16**

Tabelle F.8: Verteilung der Blockzugriffe

Als Ursache für die vorhandenen Differenzen zwischen den gemessenen Werten aus Tabelle F.7 und den rechnerisch ermittelten Aufwandswerten in Tabelle F.8 werden ebenfalls Katalogzugriffe, Zugriffe für Änderungen an Segmentverwaltungsblöcken etc. vermutet.

Trotz der höheren Anzahl an logischen Blockzugriffen, dürfte der anfallende physische I/O-Aufwand bei der von Oracle verwendeten Implementierung, gegenüber einer „reinen“ Verschiebefunktionalität, nur wenig höher sein. Wenn davon ausgegangen wird, dass die Lösch- und Einfügeoperation zeitnah aufeinanderfolgend ausgeführt werden, befinden sich die benötigten Indexblöcke nach Ausführung der Löschoperation mit sehr hoher Wahrscheinlichkeit im Puffer. Somit fallen für die Indexzugriffe beim Einfügen nur logische Blockzugriffe an.

Zur Abschätzung der Anzahl physischer I/O-Operationen fließen in die Berechnungen (Abschnitt 7.3 und Tabelle F.10) ebenfalls im Rahmen der aufgenommenen Messreihen ermittelte Werte für den jeweiligen Pufferungsgrad von Daten- und Indexblöcken ein. Gleiches gilt bezüglich des Einflusses asynchroner Schreiboperationen. Dabei wurden in einer Messreihe ca. **28400** Sätze verschoben und im Rahmen einer weiteren Messreihe ca. **42300** Sätze. Die jeweiligen Messungen wurden mehrfach wiederholt und anschließend wurden Durchschnittswerte errechnet, die in *Tabelle F.9* aufgeführt sind.

	TEILNEHMER
Pufferungsgrad der Indexe	0,98
Pufferungsgrad der Datenblöcke	0,82
Anteil geschriebener Indexblöcke	0,15
Anteil geschriebener Datenblöcke	0,23

Tabelle F.9: Messwerte zu Pufferungsgraden und den Auswirkungen asynchronen Schreibens bei In-Place-Reorganisationen in der Beispielumgebung

Der während der Messreihen anfallende erwartete Aufwand wurde vorab, wie in Abschnitt 7.2.3 beschrieben, berechnet und mit dem auf der Dateischnittstelle der die

Daten- und Indexsegmente enthaltenden Table Spaces gemessenen Schreib- und Leseaufwand verglichen. *Tabelle F.10* zeigt eine Gegenüberstellung der errechneten und der gemessenen Werte.

	ca. 28 400 zu verschiebende Sätze (Freiplatzanteil ca. 27%)		**ca. 42 300 zu verschiebende Sätze (Freiplatzanteil ca. 41%)**	
	errechnete Werte	**gemessene Werte**	**errechnete Werte**	**gemessene Werte**
gelesene Datenblöcke	5112	6438	7614	6868
geschriebene Datenblöcke	6532	7734	9729	7919
gelesene Indexblöcke	1704	2521	2538	1459
geschriebene Indexblöcke	12695	12886	18909	18634
Summen	26043	29579	38790	34880

Tabelle F.10: Gegenüberstellung von errechnetem und gemessenem I/O-Aufwand

Die gemessenen Aufwandswerte liegen bei der ersten Messreihe etwas über und bei der zweiten Messreihe etwas unter den errechneten Werten. Dies ist darauf zurückzuführen, dass in den Berechnungen für die Pufferungsgrade und die Anteile tatsächlich erfolgter Schreiboperationen die Durchschnittswerte beider Messreihen eingeflossen sind. Durch die größere Anzahl zu verschiebender Sätze im Rahmen der zweiten Messreihe ergibt sich da auch eine höhere Lokalität der Zugriffe als bei der ersten Messreihe. Durch die höhere Lokalität können anteilig mehr Zugriffe im Puffer erfolgen. Bei der ersten Messreihe liegt die Lokalität unter der, die sich aus den Durchschnittswerten für Pufferungsgrade und Schreiboperationen ergibt. Dies zeigt nochmals, dass möglichst genaue Einschätzungen der Einflüsse der Pufferung von Datenzugriffen wichtig ist, besonders für die Abschätzung der Kosten von online durchführbaren (Teil-)Reorganisationen. Das lässt aus unserer Sicht weiterführende Untersuchungen empfehlenswert erscheinen.

In den Beispielrechnungen in Abschnitt 7.3 wurden die Durchschnittswerte aus Tabelle F.9 verwendet. Allerdings wurde mit **52100** zu verschiebenden Sätzen gerechnet. Dies würde, wie in den Beispielen für die übrigen Verfahren auch, einem Freiplatzanteil von ca. **50%** entsprechen. Damit wären die errechneten Werte allerdings auch etwas höher, als sie in der Beispielumgebung tatsächlich anfallen würden.

Anhang G – Pseudocode-Darstellung Branch and Bound

```
{ Datentyp zur Speicherung der Daten eines Reorganisationskandidaten }

candidate = record of
              N: real;              { Nutzen }
              C: integer;           { Reorganisationskosten }
              D: real;              { relativer Nutzen }
              k: double;            { Zielfunktionsvariable }
              fixed: boolean;       { Wert fixiert ?}
              Name String;   { Name des Reorganisationskandidaten }
              besteLoesung: integer; { Wert von k bei der bish. best. Lösg. }
       end;

{ Definition globaler Variablen }
besterZielwert: real;
input_v vector of candidate;        { Vektor mit Daten der Kandidaten }


{ Der Funktion greedy wird wiederum ein Vektor mit den Daten der Reorganisations-}
{ kandidaten, deren Anzahl sowie der Wert für die maximal aufwendbaren Kosten als}
{ Parameter übergeben. Die Funktion wird für die einzelnen Problemstellungen     }
{ aufgerufen. Ihre Aufgabe ist es, für die jeweils gegebene Problemstellung eine }
{ Lösung zu suchen. Dazu werden zunächst die von schon fest eingeplanten         }
{ (fixierten) Kandidaten verbrauchten Kosten berechnet. Anschließend werden alle }
{ Kandidaten von der Reorganisation ausgeschlossen, deren Reorganisationskosten  }
{ den Wert für den Rest an aufwendbaren Kosten übersteigen. Danach erfolgt das   }
{ Einplanen der noch „freien" Reorganisationskandidaten, bis das Limit der       }
{ verfügbaren Kosten erreicht oder überschritten ist. Wurden die maximal         }
{ aufwendbaren Kosten beim Einplanen eines Kandidaten überschritten, so wird der }
{ Kandidat zunächst aus- und anschließend anteilig eingeplant. Die Indexnummer   }
{ des Datensatzes innerhalb des Vektors wird als Nummer der Branching-Variablen  }
{ gemerkt. Anschließend wird die obere Grenze für den erreichbaren Zielfunktions-}
{ wert berechnet. Ist der Wert kleiner als der Zielfunktionswert der besten      }
{ bisher bekannten Lösung oder wurde eine gültige Lösung gefunden, so liefert    }
{ die Funktion greedy einen ungültigen Indexwert an die Funktion baum, die       }
{ daraufhin das Teilproblem nicht weiter verzweigt. Anderenfalls wird die Index- }
{ nummer der Branching-Variablen an die Funktion baum geliefert und von dieser   }
{ wird das Teilproblem weiter verzweigt und untersucht.                          }

function greedy(cv, Cmax, anzahl)
begin
       inhalt:=0;
       zielwert:=0;
       schranke:=0;

       { Berechnung von verbrauchten Kosten und Zielfunktionswert }
       { für fixierte Variablen }
       for i in 1..anzahl
       do
              if(b[i].Fixed == true)
              then
                     inhalt:=inhalt + cv[i].C * cv[i].k; { verbrauchte Kosten }
                     zielwert:=zielwert + cv[i].N * cv[i].k; { Zielfunktionswert }
              fi;
       od;

       { Einplanen der freien Kandidaten }

       i:=0;

       do
              if(cv[i].k = „unbelegt“)      { akt. Gegenstand einplanbar }
              then
                     cv[i].k:=1;                  { Gegenstand einplanen }
                     inhalt:=inhalt + b[i].C; { erhöhe verbrauchte Kosten }
                     if (inhalt <= Cmax)   { nicht alle Kosten verbraucht }
                     then
                     zielwert:=zielwert + cv[i].N; { erhöhe akt. Zielwert }
```

```
                        if(zielwert > besterZielwert)        { Lösung besser }
                        then                                 { Lösung merken }
                                for j in 1..anzahl           { Kopiere Lösungsvektor }
                                do
                                        if (cv[j].k = „unbelegt“)
                                        then
                                                input_v[j].besteLoesung:= 0;
                                        else
                                                input_v[j]:=cv[j].k;
                                        fi;
                                od;
                                besterZielwert=zielwert;
                        fi;
                fi;
                i:=i+1;
        while (i <= anzahl and inhalt < Cmax);

        if(inhalt > Cmax) { letzten Kandidat nur anteilig einplanen }
        then
                i:=i-1;{ Indexnr. des letzten eingeplanten Kandidaten }
                inhalt:=inhalt - cv[i].C;     { Ausplanen }
                cv[i].k:=(Cmax - inhalt) / cv[i].C;  { Zuweisg. gebrochener Wert }
                branch=i;                            { Merken Branching-Variable }
        fi;

        for i in 1..anzahl    { obere Schranke berechnen }
        do
                if(cv[i].k <> „unbelegt“)
                then
                        schranke:=schranke + cv[i].N * cv[i].k;
                fi;
        od;

        if(schranke < besterZielwert or branch = „unbelegt“)
        then
                return 0; { Problem ausgelotet }
        else
                return branch; { Problem weiter verzweigen }
        fi;
end;


{ Der Funktion baum werden der Vektor mit den Daten der Reorganisations-         }
{ kandidaten, deren Anzahl, die maximal aufwendbaren Kosten und der Indexwert    }
{ der Branching-Variablen übergeben. Mit der Funktion wird das Aufteilen der     }
{ Problemstellungen und das Betrachten der Teilprobleme durchgeführt. Zunächst   }
{ wird für jedes Teilproblem eine Kopie des Datenvektors angelegt und            }
{ anschließend werden die Branching-Variablen belegt und fixiert. Danach wird    }
{ für jedes Teilproblem mit der Funktion greedy eine Lösung ermittelt. Liefert   }
{ greedy für ein Teilproblem den Indexwert einer Branching-Variablen, so wird    }
{ das Problem weiter verzweigt. Dies erfolgt durch einen rekursiven Aufruf der   }
{ Funktion baum.                                                                 }

function baum(cv, branch, Cmax, anzahl)
begin

        cvL,cvR vector of candidate;  { lokale Vektoren für Teilprobleme }

        for i in 1..anzahl            { Kopiere Daten in lokale Vektoren }

        do
                cvL[i]:=cv[i];
                cvR[i]:=cv[i];
        od;

        { Fixieren der Branching-Variable für PnL }
        cvL[branch].k:=0;
        cvL[branch].fixed:=true;

        branchPnL:=greedy(cvL,Cmax,anzahl); { Anwenden Greedy-Heuristik auf PnL }

        { Fixieren Branching-Variable für PnR }
        cvR[branch].k=1;
        cvR[branch].fixed=true;
```

```
        { Berechnen der bereits verbrauchten Kosten }
        inhalt:=0;
        for i in 1..anzahl
        do
                if(cvR[i].k <> „unbelegt“)
                then
                        inhalt:=inhalt + cvR[i].C * cvR[i].k;
                fi;
        od;

        { Fixieren der Variablen für PnR, bei denen die Kosten die }
        { restlichen verfügbaren übersteigen }
        for i in 1..anzahl
        do
                if ((cvR[i].C > Cmax - inhalt) and „noch nicht fixiert“)
                then
                        cvR[i].k=0;
                        cvR[i].fixed=true;
                fi;
        od;

        branchPnR=greedy(cvR,Cmax,anzahl);   { Anwenden Greedy-Heuristik auf PnR }

        if(branchPnL <> 0)                   { Branching-Variable für PnL vorhanden }
        then                                 { Problem weiter verzweigen }
                baum(cvL,branchPnL,Cmax,anzahl);
        fi;

        if(branchPnR <> 0)                   { Branching-Variable für PnR vorhanden }
        then                                 { Problem weiter verzweigen }
                baum(cvR,branchPnR,Cmax,anzahl);
        fi;
end;



{ Der Funktion branch_and_bound werden der Vektor mit den Daten der              }
{ Reorganisationskandidaten und deren Anzahl sowie die maximal aufwendbaren      }
{ Kosten übergeben. Von der Funktion wird der Vektor zunächst absteigend nach dem}
{ relativen Nutzen sortiert. Anschließend werden alle Variablen fixiert, die die }
{ maximal aufwendbaren Kosten übersteigen. Von der Funktion greedy wird dann eine}
{ Lösung für das Originalproblem ermittelt. Liefert die Funktion keinen gültigen }
{ Indexwert für eine Branching-Variable, so wurde eine zulässige Lösung gefunden }
{ und das Verfahren kann beendet werden. Wird von der Funktion greedy ein        }
{ gültiger Indexwert für eine Branching-Variable geliefert, so wird die Funktion }
{ baum aufgerufen.                                                               }


function branch_and_bound(cv, Cmax, anzahl)
begin

        cv:= sort(cv);        { sortiere Vektor absteigend nach relativem Nutzen }
        besterZielwert:=0;


        for i in 1..anzahl
        do
                if (cv[i].C > Cmax)   { Reorg.-Kosten übersteigen max. Gesamtkosten }
                then
                        cv[i].k:=0;                  { Kandidat ausschließen }
                        cv[i].Fixed:=true;           { Kandidat fixieren }
                fi;
        od;

        branch:=greedy(cv,Cmax,anzahl);      { Anwenden Greedy-Heuristik }
                                             { auf das Originalproblem }

        if(branch <> 0)                              { Branching-Variable vorhanden }
        then                                         { Problem weiter verzweigen }
                baum(cv,branch,cmax,anzahl);
        fi;
end;
```

Literaturverzeichnis

[AF06] J. Albrecht, M. Fiedler. Datenbank-Tuning – einige Aspekte am Beispiel von Oracle 10g. In *Datenbank-Spektrum, Heft 16/2006.* dpunkt.verlag, Heidelberg, Februar 2006

[AON96] K. J. Achyutuni, E. Omiecinski, S. B. Navathe. Two Techniques for On-Line Index Modification in Shared Nothing Parallel Databases. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data.* Montreal, Kanada, Juni 1996

[Ash00] G. Asherie. *Tuning a Database Reorganization for Maximum Speed*. White Paper. Quest Software Inc., 2000

[Ast02] I. Astrova. Transforming Relational Database Schema with Multi-Valued Dependencies into Object-Oriented Database Schema. In *Proc. of the Baltic Conference, BalticDB & IS 2002*. Tallinn, Estland, Juni 2002

[Aue04] K. Auerbach. *2003 TopTen Program Summary: Select Findings from the TopTen Program*. White Paper. Winter Corporation, Waltham, USA, August 2004

[Bat82] D. S. Batory. Optimal File Designs and Reorganization Points. In *ACM Transactions on Database Systems (TODS), Vol. 7, No. 4*. ACM Press, New York, Dezember 1982

[Bau06] J. Baumeister. SQL Server 2005. In *Datenbank-Spektrum, Heft 16/2006.* dpunkt.verlag, Heidelberg, Februar 2006

[BCS75] BCS/CODASYL Data Description Language Committe Data Base Administration Working Group. Report. Juni 1975

[Bel96] S. Bell. Semantische Anfrageoptimierung. In *Informatik-Spektrum, Bd. 19, Nr. 5.* Springer-Verlag, Berlin/Heidelberg, Oktober 1996

[BF77] B. T. Bennett, P. A. Franaszek. Permutation Clustering: An Approach to On-Line Storage Reorganization. In *IBM Journal of Research and Development, Vol. 21, No. 6.* International Business Machines Corporation, November 1977

[BG82] D. S. Batory, C. C. Gotlieb. A Unifying Model of Physical Databases. In *ACM Transactions on Database Systems (TODS), Vol. 7, No. 4*. ACM Press, New York, Dezember 1982

[BHL+02] D. Beulke, M. Hubel, L. Lyon, P. Nelson. Unveiling 8.1: The Next Generation. In *IDUG Solutions Journal, Vol. 9, No. 2.* International DB2 Users Group, August 2002

[BMC04] *SmartDBA Performance Management Solutions for Oracle, Version 2.6.* White Paper. BMC Software Inc., 2004

[BR02] D. Beeler, I. Rodriguez. *Optimizing Your Database Performance ... the Easy Way.* White Paper. BMC Software Inc., 2002

[Bur98] D. Burleson. *Oracle 8 Tuning.* Sybex-Verlag, Düsseldorf, 1998

[BVR01] *Jahresbericht des Bundesverbands der Deutschen Volksbanken und Raiffeisenbanken 2000.* Bundesverband der Deutschen Volksbanken und Raiffeisenbanken – BVR, Berlin, 2001

[BVR03] *Jahresbericht des Bundesverbands der Deutschen Volksbanken und Raiffeisenbanken 2002.* Bundesverband der Deutschen Volksbanken und Raiffeisenbanken – BVR, Berlin, 2003

[CCN+99] M. J. Carey, D. D. Chamberlin, S. Narayanan, B. Vance, D. Doole, S. Rielau, R. Swagerman, N. M. Mattos. O-O, What Have They Done to DB2? In *Proc. of the 25th International Conference on Very Large Data Bases.* Edinburgh, Großbritannien, September 1999

[CGN02] S. Chaudhuri, A. Gupta, V. Narasayya. Compressing SQL Workloads. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data.* Madison, USA, 2002

[CN97] S. Chaudhuri, V. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proc. of the 23rd International Conference on Very Large Data Bases*. Athen, Griechenland, August 1997

[CN98] S. Chaudhuri, V. Narasayya. AutoAdmin 'What-if' Index Analysis Utility. In *Proc. of the 1998 ACM SIGMOD International Conference on Management of Data.* Seattle, USA, Juni 1998

[DK98] S. Dorendorf, K. Küspert. *Datenbankreorganisation bei relationalen Datenbank-Management-Systemen.* Forschungsergebnisse der Fakultät für Mathematik und Informatik, Math/Inf/98/28. Friedrich-Schiller-Universität Jena, November 1998

[DK00] S. Dorendorf, K. Küspert. Datenbankreorganisation bei relationalen Datenbank-Management-Systemen. In *it+ti – Informationstechnik und Technische Informatik, Heft 3/2000.* Oldenbourg Wissenschaftsverlag, München, Juni 2000

[Dor97] S. Dorendorf. *Intelligent Network: SMP-Operating.* course paper. Siemens AG, 1997

[Dor99a] S. Dorendorf. Die fünf großen W der Datenbankreorganisation. In F. Hüsemann, K. Küspert, F. Mäurer (Hrsg), *Jenaer Schriften zur Mathematik und Informatik*, 11. Workshop „Grundlagen von Datenbanken“. Luisenthal, Mai 1999

[Dor99b] S. Dorendorf. Fragmentierung von Datenbankinhalten –Facetten eines scheinbar klaren Begriffs–. In *Datenbank Rundbrief, Nr. 24.* GI-Fachgruppe 2.5.1, November 1999

[Dor00] S. Dorendorf. *Beschreibung eines Speicher- und Verhaltensmodells als Grundlage zur Bedarfsanalyse einer Datenreorganisation bei relationalen Datenbank-Management-Systemen.* Jenaer Schriften zur Mathematik und Informatik, Mat/Inf/00/05. Friedrich-Schiller-Universität Jena, März 2000

[Dor03] S. Dorendorf. Reorganisationsbedarfsanalysen bei relationalen Datenbankmanagementsystemen unter Beachtung der Workload. In *Datenbank-Spektrum, Heft 5/2003.* dpunkt.verlag, Heidelberg, Februar 2003

[Dor05a] S. Dorendorf. Ermittlung des Nutzens von Datenbankreorganisationen: Notwendigkeit, Werkzeuge, Herangehensweisen. In *it – Information Technology, Heft 3/2005.* Oldenbourg Wissenschaftsverlag, München, Juni 2005

[Dor05b] S. Dorendorf. Quantifizierung des zu erwartenden Nutzens von Datenbankreorganisationen. In *Informatik-Forschung und Entwicklung, Bd. 20, Nr. 1-2.* Springer-Verlag, Berlin/Heidelberg, Oktober 2005

[Dun02] O. Dunemann. *Anfrageoptimierung für OLAP-Anwendungen in virtuellen Data Warehouses.* Dissertation. Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, September 2002

[DPW05] *ZEBRA-Zentrale Briefdatenbank, Architekturüberblick*. Projektunterlagen. Deutsche Post World Net, 2003-2005

[DW00] C. Dialeris, G. Wood. *Performance Tuning with Statspack, Part II.* White Paper. Oracle Corporation, Juli 2000

[EBL98] T. Ellinger, G. Beuermann, R. Leisten. *Operations Research: Eine Einführung,* 4. Auflage. Springer-Verlag, Berlin/Heidelberg, 1998

[ECS93] H. Engesser (Hrsg.), C. Volker, A. Schwill (Bearb.). *Duden Informatik*, 2. überarbeitete und erweiterte Auflage. Duden-Verlag, Mannheim/ Leipzig/Wien/Zürich, 1993

[EMB03] *DBArtisan Online-Dokumentation*. Dokumentation zu DBArtisan Version 7.3.1 Workbench inkl. DBArtisan Analyst Series. embarcadero Technologies, 2003

[EN02] R. Elmasri, S. B. Navathe. *Grundlagen von Datenbanksystemen,* 3. überarb. Auflage. Pearson Education Deutschland, München, 2002

[Gan05] K. Ganskow. *Kostenfunktionen für Join-Operationen und Update-Operationen.* Studienarbeit. Institut für Informatik, Friedrich-Schiller-Universität Jena, September 2005

[Gol98] C. Gollmick. *Analyse, Visualisierung und Auswertung von DB2-Log-Einträgen.* Studienarbeit. Institut für Informatik, Friedrich-Schiller-Universität Jena, September 1998

[GBG04] P. Ganesan, M. Bawa, H. Garcia-Molina. Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems. In *Proc. of the 30th International Conference on Very Large Data Bases.* Toronto, Kanada, September 2004

[GE98] R. Graf, G. Etz. Optimierungsstrategien in 1:n-Relationen – Durchsatzsteigerung durch Clustern. In *Datenbankfokus, Heft 03/98.* IT Verlag für innovative Technologien, Höhenkirchen, März 1998

[Gen02] J. Gennick. Redefine Tables Online. In *Oracle Magazine*. Oracle Corporation, Juli/August 2002

[GK03] A. Ganesh, S. Kumar. *The Self-Managing Database: Proactive Space & Schema Object Management*. White Paper. Oracle Corporation, November 2003

[GKM96] C. A. Gerlhof, A. Kemper, G. Moerkotte. On the Cost of Monitoring and Reorganization of Object Bases for Clustering. In *ACM SIGMOD Record, Vol. 25, No. 3*. ACM Press, New York, September 1996

[Ham83] E. Hamm. *Simulation von B*-Bäumen mit verallgemeinerter Überlauftechnik.* Projektarbeit. Universität Kaiserslautern, Fachbereich Informatik, Kaiserslautern, April 1983

[Här05] T. Härder. DBMS Architecture – the Layer Model and its Evolution. In *Datenbank-Spektrum, Hefte 13/2005 und 14/2005.* dpunkt.verlag, Heidelberg, Mai/August 2005

[Her97] A. Herbst: *Anwendungsorientiertes DB-Archivieren. Neue Konzepte zur Archivierung in Datenbanksystemen.* Springer-Verlag, Berlin/ Heidelberg, 1997

[Hei04] A. Heisrath. *Management Tools für gängige Datenbank-Management-Systeme.* Studienarbeit. Institut für Informatik, Friedrich-Schiller-Universität Jena, September 2004

[Hei05] A. Heisrath. *Untersuchung der Optimierungspotenziale für Lade- und Transformationsprozesse in SAP NetWeaver BI unter Ausnutzung von DBMS-spezifischer Funktionalität.* Diplomarbeit. Institut für Informatik, Friedrich-Schiller-Universität Jena, November 2005

[Hel01] T. Helm. *Dokumentation des Katalogs des DBMS ORACLE und Transformation ausgewählter Katalogdaten in ein einheitliches Informationsschema*. Studienarbeit. Studienrichtung Wirtschafts-informatik, Berufsakademie Thüringen, Staatliche Studienakademie Gera, Februar 2001

[HL03] B. Himatsingka, J. Loaiza. *How to Stop Defragmenting and Start Living: The Definitive Word on Fragmentation.* Oracle Paper #711. Oracle Corporation, 2003

[HR01] T. Härder, E. Rahm. *Datenbanksysteme - Konzepte und Techniken der Implementierung*, 2. überarb. Auflage. Springer-Verlag, Berlin/ Heidelberg/New York, 2001

[HT05] L. Hobbs, P. Tsien. *Oracle Database 10g Release 2 Online Data Reorganization and Redefinition*. White Paper. Oracle Corporation, Mai 2005

[HWW04] T. Holey, G. Welter, A. Wiedemann. *Wirtschaftsinformatik – Kompendium der praktischen Betriebswirtschaft*. Friedrich Kiehl Verlag, Ludwigshafen, 2004

[IBM02] *IBM DB2 Universal Database Command Reference Version 8*. International Business Machines Corporation, 2002

[IBM04] *DB2 UDB V8 and WebSphere V5 Performance Tuning and Operations Guide,* First Edition. International Business Machines Corporation, März 2004

[IBM05a] *IBM Informix Dynamic Server Administrator's Guide, Version 10.0*, Third Edition. International Business Machines Corporation, Dezember 2005

[IBM05b] *IBM Informix Dynamic Server Administrator's Reference, Version 10.0*, Second Edition. International Business Machines Corporation, Dezember 2005

[Ioa96] Y. E. Ioannidis. Query Optimization. In *ACM Computing Surveys, symposium issue on the 50th Anniversary of ACM, Vol. 28, No. 1*. ACM Press, New York, März 1996

[ISO03a] International Organization for Standardization. *Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework).* ISO/IEC 9075-1:2003. Genf, Dezember 2003

[ISO03b] International Organization for Standardization. *Information technology – Database languages – SQL – Part 11: Information and Definition Schemas (SQL/Schemata).* ISO/IEC 9075-11:2003. Genf, Dezember 2003

[Joc05] C. Jochum. Versinkt das IT-Management in der Bedeutungslosigkeit? In *Informatik-Spektrum, Bd. 28, Nr. 4.* Springer-Verlag, Berlin/Heidelberg, August 2005

[KE04] A. Kemper, A. Eickler. *Datenbanksysteme - Eine Einführung*, 5. aktualisierte und erweiterte Auflage. Oldenbourg Wissenschaftsverlag, München, 2004

[Keß95] U. Keßler. *Flexible Speicherungsstrukturen und Sekundärindexe in Datenbanksystemen für komplexe Objekte.* Dissertation. Fakultät für Informatik, Universität Ulm, Mai 1995

[Kis02] F. Kissel. *Physische Speicherung komplexer Objekte mit kollektionswertigen Attributen in ORDBMS.* Studienarbeit. Institut für Informatik, Friedrich-Schiller-Universität Jena, Juli 2002

[Kli05] O. Klinger. *Abschätzung von I/O-Kosten für Zugriffsoperationen bei RDBMS*. Diplomarbeit. Institut für Informatik, Friedrich-Schiller-Universität Jena, September 2005

[KLS02] C. Kauhaus, J. Lufter, S. Skatulla. Eine Transformationsschicht zur Realisierung objektrelationaler Datenbankkonzepte mit erweiterter Kollektionsunterstützung. In *Datenbank-Spektrum, Heft 4/2002*. dpunkt.verlag, Heidelberg, November 2002

[KR04] A. Koeller, E. A. Rundensteiner. Incremental Maintenance of Schema-Restructuring Views in SchemaSQL. In *IEEE Transactions on Knowledge and data Engineering, Vol. 16, No. 9.* IEEE Computer Society, September 2004

[Küs83] K. Küspert. Storage Utilization in B*-Trees with a Generalized Overflow Technique. In *Acta Informatica, Vol. 19, No. 1.* Springer-Verlag, Berlin/Heidelberg, April 1983

[Küs82] K. Küspert. Modelle für die Leistungsanalyse von Hashtabellen mit "Separate Chaining". In *Angewandte Informatik, Bd. 24, Nr. 9.* Friedrich Vieweg & Sohn Verlagsgesellschaft, Wiesbaden, September 1982

[LKO+00] M.-L. Lee, M. Kitsuregawa, B. C. Ooi, K.-L. Tan, A. Mondal. Towards Self-Tuning Data Placement in Parallel Database Systems. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data.* Dallas, USA, 2000

[LS96] D. B. Lomet, B. Salzberg (Hrsg.). *IEEE Bulletin of the Technical Committee on Data Engineering, Vol. 19, No. 2*. IEEE Computer Society, Juni 1996

[LS02] C. Lawson, R. Schrag. *Don't Shut Down That Database! Use Oracle 9i Online Object Redefinition Instead*. White Paper. Oracle Corporation, März 2002

[Luf05] J. Lufter. *Unterstützung komplexer Datenstrukturen in SQL-Norm und objektrelationalen Datenbanksystemen.* Dissertation. Institut für Informatik, Friedrich-Schiller-Universität Jena, Juli 2005

[Mak03] M. E. Makoui. *Heuristische Anfrageoptimierung in Relationalen Datenbanken.* Diplomarbeit. Fachbereich für Informatik, Institut für Informationssysteme, Universität Hannover, Januar 2003

[MDL87] H. C. Mayr, K. R. Dittrich, P. C. Lockemann. Datenbankentwurf. In P. C. Lockemann, J. W. Schmidt (Hrsg). *Datenbank-Handbuch*, unveränderter Nachdruck 1993. Springer-Verlag, Berlin/Heidelberg/ New York/London/Paris/Tokio, 1987

[Mea97] A. L. Meads. *Clustering Strategies for Object Databases*. Dissertation. Aston University, Birmingham, Großbritannien, 1997

[MHR96] H. Mucksch, J. Holthuis, M. Reiser. Das Data Warehouse Konzept - ein Überblick. In *Wirtschaftsinformatik, Heft 4/1996*. Friedrich Vieweg & Sohn Verlagsgesellschaft, Wiesbaden, Juli 1996

[MK94] W. J. McIver, R. King. Self-Adaptive, On-Line Reclustering of Complex Object Data. In *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data.* Minneapolis, USA, Mai 1994

[MM87] V. M. Markowitz, J. A. Makowsky. Incremental Reorganization of Relational Databases. In *Proc. of the 13rd International Conference on Very Large Data Bases*. Brighton, Großbritannien, September 1987

[MRB99] V. Markl, F. Ramsak, R. Bayer. Improving OLAP Performance by Multidimensional Hierarchical Clustering. In *Proc. of International Database Engineering & Applications Symposium – IDEAS '99.* Montreal, Kanada, August 1999

[Nav80] S. B. Navathe. Schema Analysis for Database Restructuring. In *ACM Transactions on Database Systems (TODS), Vol. 5, No. 2*. ACM - Association for Computing Machinery, Juni 1980

[NF76] S. B. Navathe, J. P. Fry. Restructuring for Large Data Bases: Three Levels of Abstraction. In *ACM Transactions on Database Systems (TODS), Vol. 1, No. 2.* ACM Press, New York, Juni 1976

[Nob95] N. Noble. Techniques for Fast Database Reorganisation. In *Proc. of the European Oracle Users Group Conference, EOUG-95.* Florenz, Italien, 1995

[Now01a] J. Nowitzky. Partitionierungstechniken in Datenbanksystemen: Motivation und Überblick. In *Informatik-Spektrum, Bd. 24, Nr. 6.* Springer-Verlag, Berlin/Heidelberg, Dezember 2001

[Now01b] J. Nowitzky. *Analytische Bestimmung einer Tabellenpartitionierung für ein Data Warehouse.* Jenaer Schriften zur Mathematik und Informatik, Math/Inf/01/17. Friedrich-Schiller-Universität Jena, August 2001

[Nuß97] R. Nußdorfer. Reorganisationsfreiheit in Datenbanksystemen: Bewertung ist anwendungsabhängig. In *Datenbank Fokus, Januar 1997.* IT Verlag für innovative Technologien, Höhenkirchen, Januar 1997

[OLA91] E. Omiecinski, W. Liu, I. Akyildiz. Analysis of a Deferred and Incremental Update Strategy for Secondary Indexes. In *Information Systems, Vol. 16, No. 3*. Pergamon Press, Juli 1991

[Oll92] H. J. Ollmert. *Datenstrukturen und Datenorganisation,* 2. korrigierte Auflage. R. Oldenbourg Verlag, München/Wien, 1992

[OLS94] E. Omiecinski, L. Lee, P. Scheuermann. Performance Analysis of a Concurrent File Reorganization Algorithm for Record Clustering. In *IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No. 2*. IEEE Computer Society, April 1994

[Omi85] E. Omiecinski. Incremantal File Reorganization Schemes. In *Proc. of the 11th International Conference on Very Large Data Bases*. Stockholm, Schweden, August 1985

[Omi88] E. Omiecinski. Concurrent Storage Structure Conversion: from B+-Tree to Linear Hash File. In *Proc. of the 4th International Conference on Data Engineering*. Los Angeles, USA, Februar 1988

[Omi89] E. Omiecinski. Concurrent File Conversion between B+-Tree and Linear Hash Files. In *Information Systems, Vol. 14, No. 5*. Pergamon Press, November 1989

[Ora02] *Oracle Enterprise Manager Database Tuning with the Oracle Tuning Pack Release 9.0.1.* White Paper. Oracle Corporation, 2002

[Ora03a] *Oracle Database Administrator's Guide 10g Release 1 (10.1)*. Oracle Corporation, Dezember 2003

[Ora03b] *Oracle Database Concepts 10g Release 1 (10.1).* Oracle Corporation, Dezember 2003

[Ora03c] *Oracle Database Performance Tuning Guide 10g Release 1 (10.1).* Oracle Corporation, Dezember 2003

[Ora03d] *Oracle Database 10g: The Self-Managing Database.* White Paper. Oracle Corporation, November 2003

[Ora03e] *PL/SQL Packages and Types Reference 10g Release 1 (10.1).* Oracle Corporation, Dezember 2003

[Ora04] *Oracle 10g Online Data Reorganization & Redefinition.* White Paper. Oracle Corporation, April 2004

[Ora05] *The Self-Managing Database: Proactive Space & Schema Object Management with Oracle Database 10g Release 2.* White Paper. Oracle Corporation, Mai 2005

[Que04] *Space Management with LiveReorg.* Online-Dokumentation. Quest Software Inc., 2004

[Ric03] T. Richter. *Application of Informix Dynamic Server with regard to high Availability at AMD Saxony*. Vortrag. 7th East European Conference, ADBIS 2003, Dresden, September 2003

[RO92] M. Rosenblum, J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *ACM Transaction on Computer Systems, Vol. 10, No. 1*. ACM Press, New York, Februar 1992

[Rab06] G. Rabinovitch. Technologien und Konzepte zur autonomen Verwaltung von IT-Systemen. In *Tagungsunterlagen des 18. Workshop über Grundlagen von Datenbanken.* Wittenberg, Juni 2006

[Ruf05] T. Ruf. *Datenbanken heute: Was hat/braucht die betriebliche Praxis seit/in 25 Jahren – und was nicht?*. Vortrag. Informatik-Kolloquium der Friedrich-Schiller-Universität Jena, der Regionalgruppe Ostthüringen der Gesellschaft für Informatik (GI) und der Fachhochschule Jena, Jena, 25. April 2005

[SA76] M. E. Senko, E. B. Altman. DIAM II and Levels of Abstraction - The Physical Device Level: A General Model for Access Methods. In P. C. Lockemann, E. J. Neuhold (Hrsg.), *Systems for Large Data Bases*, Proc. of the 2nd International Conference on Very Large Data Bases. Brüssel, Belgien, September 1976

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the 1979 ACM SIGMOD International Conference on Management of Data.* Boston, USA, Mai 1979

[SAG97] *ADABAS D Schattenspeicherkonzept – Reorganisationsfreie Datenhaltung ohne I/O-Engpässe.* White Paper. SQL GmbH - Software AG, 1997

[SBC97] G. H. Sockut, T. A. Beavin, C.-C. Chang. A method for on-line reorganization of a database. In *IBM Systems Journal, Vol. 36, No. 3.* International Business Machines Corporation, 1997

[SC99] I. Schmitt, S. Conrad. Restructuring Object-Oriented Database Schemata by Concept Analysis. In T. Polle, T. Ripke, K.-D. Schewe (Hrsg.), *Fundamentals of Information Systems, Papers from the Seventh Workshop on Foundations of Models and Languages for Data and Objects*, Timmel (Ostfriesland), Oktober 1998

[Sch99] R. Schaarschmidt. *Konzept und Sprache für die Archivierung in Datenbanksystemen.* Dissertation. Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, Dezember 1999

[Sch00] R. Schumacher. *Eliminating Space Reorganizations in Oracle8i*. White Paper. embarcadero Technologies Inc., 2000

[Sch04] R. Schumacher. Why You Need Capacity Planning. In *Database Trends And Applications, Vol. 18, No. 7*. www.dbta.com, Juli 2004

[SD92] B. Salzberg, A. Dimock. Principles of Transaction-Based On-Line Reorganization. In *Proc. of the 18th International Conference on Very Large Data Bases.* Vancouver, Kanada, August 1992

[SD03] S. Skatulla, S. Dorendorf. Optimization of Storage Structures of Complex Types in Object-Relational Database Systems. In L. A. Kalinichenko, R. Manthey, B. Thalheim, U. Wloka (Hrsg.), *Advances in Databases and Information Systems, 7th East European Conference, ADBIS 2003*. Dresden, September 2003. Springer-Verlag, Berlin/ Heidelberg, 2003

[SG79] G. H. Sockut, R. P. Goldberg. Database Reorganization - Principles and Practice. In *ACM Computing Surveys*, *Vol. 11, No. 4.* ACM Press, New York, Dezember 1979

[SHS05] G. Saake, A. Heuer, K.-U. Sattler. *Datenbanken: Implementierungstechniken* (2., aktualisierte und erweiterte Auflage). mitp-Verlag, Bonn, 2005

[Sha95] C. A. Shallahamer. Avoiding a Database Reorganization. In *Proc. of the European Oracle Users Group Conference, EOUG-95.* Florenz Italien, 1995

[Sin00] A. Singer. *Analyse und Bewertung des Cluster-Konzepts des DBMS Oracle.* Studienarbeit. Institut für Informatik, Friedrich-Schiller-Universität Jena, August 2000

[Ska02] S. Skatulla. Storage of Complex Types with Collection-Valued Attributes in Object-Relational Database Systems. In *Tagungsband zum 14. GI-Workshop „Grundlagen von Datenbanken“.* Fischland/Darß, Mai 2002

[Ska06] S. Skatulla. *Speicherung und Indexierung komplexer Objekte in objektrelationalen Datenbank-Management-Systemen.* Dissertation. Institut für Informatik, Friedrich-Schiller-Universität Jena, April 2006

[SKK+97] A. Scholl, G. Krispin, R. Klein, W. Domschke. Branch and Bound - Optimieren auf Bäumen: je beschränkter, desto besser. In *c't - Magazin für Computer Technik, Heft 10/1997*. Heise Zeitschriften Verlag, Hannover, Oktober 1997

[SM95] M. Stonebraker, D. Moore. *Object Relational DBMSs: The Next Great Wave.* Morgan Kaufmann Publishers, San Francisco, 1995

[Soc77] G. H. Sockut. *Data Base Performance Under Concurrent Reorganization and Usage*. Dissertation. Div. of Applied Sciences, Harvard University, Cambridge, USA, 1977

[Soe80] L. Söderlund. *A Study on Concurrent Data Base Reorganization.* Dissertation. Royal Inst. of Technology and Univ. of Stockholm, Schweden, Mai 1980

[Soe81] L. Söderlund. Concurrent Data Base Reorganization – Assessment of a Powerful Technique through Modeling. In *Proc. of the 7th International Conference on Very Large Data Bases*. Cannes, Frankreich, September 1981

[SPO89] P. Scheuermann, Y. C. Park, E. Omiecinski. A Heuristic File Reorganization Algorithm based on Record Clustering. In *BIT Computer Science and Numerical Mathematics, Vol. 29, No. 3*. Springer-Verlag Niederlande, September 1989

[SSP+00] S. Skatulla, R. Schaarschmidt, P. Pistor, K. Küspert. Entwurf und Implementierung eines SQL-normkonformen Datenbankkatalogs für ein relationales Datenbankmanagementsystem. In *Informatik-Forschung und Entwicklung, Bd. 15, Nr. 3*. Springer-Verlag, Berlin/Heidelberg, September 2000

[SST97] G. Saake, I. Schmitt, C. Türker. *Objektdatenbanken - Konzepte, Sprachen, Architekturen*. International Thomson Publishing, Bonn, 1997

[Ste02] H. Stefani (Hrsg.). *Datenarchivierung mit SAP*. SAP Press, 2002

[Sto06] K. Stolze. *Integration of Spatial Vector Data in Relational Database Environments of Enterprises.* Dissertation. Institut für Informatik, Friedrich-Schiller-Universität Jena, 2006 (in Vorbereitung)

[Stö01] U. Störl. *Backup und Recovery in Datenbanksystemen,* Teubner Texte zur Informatik Band 33. B. G. Teubner-Verlag, Stuttgart/Leipzig/ Wiesbaden, 2001

[SWS+05] X. Sun, R. Wang, B. Salzberg, C. Zou. Online B-Tree Merging. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data.* Baltimore, USA, Juni 2005

[TK78] D. Tsichritzis, A. Klug. The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Dabatase Management Systems. In *Information Systems, Vol. 3, No. 3*. Pergamon Press, 1978

[TL91] R. Trautloft, U. Lindner. *Datenbanken - Entwurf und Anwendung.* Verlag Technik, Berlin, 1991

[Tue78] W. G. Tuel. Optimum Reorganization Points for Linearly Growing Files. In *ACM Transactions on Database Systems (TODS), Vol. 3, No. 1*. ACM Press, New York, März 1978

[Wan00] C.-M. Wang. *Dynamic Online Data Clustering for Object-oriented Databases*. Dissertation. University of Illinois at Urbana-Champaign, Urbana, USA, 2000

[Wie05] F. Wieczorek. *Datenbankreorganisationen: Verfahren und Kostenvorhersagen.* Diplomarbeit. Institut für Informatik, Friedrich-Schiller-Universität Jena, Dezember 2005

[Wil04] P. Williams. *Erstellung eines Programms zur Reorganisationsbedarfsanalyse*. Studienarbeit. Institut für Informatik, Friedrich-Schiller-Universität Jena, Mai 2004

[Win05] M. Winer. *Reorganization in DB2 UDB for LUW "Why, How, and What to Expect"*. Vortragsunterlagen. 2005.

[YDT76] S. B. Yao, K. S. Das, T. J. Teorey. A Dynamic Database Reorganization Algorithm. In *ACM Transactions on Database Systems (TODS), Vol. 1, No. 2*. ACM Press, New York, Juni 1976

[Zou96] C. Zou. *Dynamic Hierarchical Data Clustering and Efficient On-line Database Reorganization.* Dissertation. Northeastern University, College of Computer Science, Boston, USA, 1996

[ZS96a] C. Zou, B. Salzberg. On-line Reorganization of Sparsely-populated B+-trees. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data.* Montreal, Kanada, Juni 1996

[ZS96b] C. Zou, B. Salzberg. *Efficiently Updating References During On-line Reorganization.* Technical Report NU-CCS-96-08. Northeastern University, College of Computer Science, Boston, USA, 1996

[ZS96c] C. Zou, B. Salzberg. Towards Efficient Online Database Reorganization. In *IEEE Bulletin of the Technical Committee on Data Engineering, Vol. 19, No. 2.* IEEE Computer Society, Juni 1996

[ZS98] C. Zou, B. Salzberg. Safely and Efficiently Updating References During On-line Reorganization. In *Proc. of the 24rd International Conference on Very Large Data Bases*. New York, USA, August 1998

[ZSL98] C. Zou and B. Salzberg and R. Ladin. Back to the Future: Dynamic Hierarchical Clustering. In *Proc. of the 14th International Conference on Data Engineering.* Orlando, USA, Februar 1998

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel und Literatur angefertigt habe.

Jena, den 13. Juni 2006

Lebenslauf

Name	:	Stefan Dorendorf
geboren am	:	23.03.1966 in Glauchau/Sachsen
Familienstand	:	verheiratet mit Frau Konstanze Dorendorf, Diplom-Informatikerin
Kinder	:	Annegret Dorendorf, geb. am 07.02.1996
		Johannes Dorendorf, geb. am 03.01.2000
Vater	:	Hans-Joachim Dorendorf, Dipl.-Bauingenieur
Mutter	:	Christa Dorendorf, Zahntechnikerin
Geschwister	:	Dr. Heike Kühn, Zahnärztin
		Astrid Thiel, Diplom-Ingenieur für Bauwesen
		Katrin Körbel, Stomatologische Schwester

1972 - 1980	Besuch der Allgemeinbildenden Oberschule in Glauchau
1980 - 1984	Besuch der Erweiterten Oberschule „Georgius Agricola“ in Glauchau und Erwerb der allgemeinen Hochschulreife
November 1984 - April 1986	Wehrdienst in der NVA
Mai 1986 - August 1986	Tätigkeit als Operator im Rechenzentrum des VEB Textilwerke "Palla", Glauchau
September 1986 - Februar 1991	Informatikstudium an der Technischen Universität Karl-Marx-Stadt/Chemnitz Studienschwerpunkt: Angewandte Informatik
September 1989 - März 1990	Ingenieurpraktikum im Forschungszentrum für Werkzeugmaschinenbau in Karl-Marx-Stadt

Februar 1991	Verteidigung der Diplomarbeit und Abschluss des Studiums als Diplom-Informatiker
März 1991 - Mai 1991	Tätigkeit als Vertriebs- und Systemingenieur in der Firma Sigma GmbH, Chemnitz
Juni 1991 - September 1993	Sachbearbeiter im gehobenen Dienst in der Abteilung für dezentrale Datenverarbeitung beim Zentralamt der Bundesanstalt für Arbeit, Nürnberg
September 1993 - August 1999	freiberufliche Tätigkeit im Bereich IT-Schulung, Entwicklung und Anpassung von Software, Planung von IT-Lösungen; Lehrtätigkeit an der Staatlichen Studienakademie Glauchau der Berufsakademie Sachsen auf nebenberuflicher Basis
seit August 1999	Leiter der Studienrichtung Wirtschaftsinformatik an der Staatlichen Studienakademie der Berufsakademie Thüringen, Studienabteilung Gera; externer Doktorand am Lehrstuhl für Datenbanken und Informationssysteme am Institut für Informatik der Friedrich-Schiller-Universität Jena

Jena, den 13. Juni 2006