Data Warehousing

Seminar im Sommersemester 1996

Christoph Breitner, Uwe Herzog, Jutta Mülle, Jörg Schlösser (Hrsg.)

Institut für
Programmstrukturen & Datenorganisation
Fakultät für Informatik
Universität Karlsruhe
[*breitner*|*herzog*|*muelle*|*joerg*]@ira.uka.de

Juli 1996

Vorwort

Sich verstärkender Wettbewerbsdruck und rasch veränderliche Märkte erfordern schnell verfügbare, umfassende und hochwertige Informationen für die Entscheidungsunterstützung. Ein wesentlicher Baustein eines Unternehmens–Informationssystems ist das Data Warehouse als eine integrierte Sicht auf die geschäftsrelevanten Unternehmensdaten. Durch Data Mining– und OLAP–Techniken wird dieses Informationssystem zu einer Entscheidungsunterstützungsumgebung ausgebaut.

Dieser Bericht umfaßt die Ausarbeitungen des im Sommersemester 1996 an der Fakultät für Informatik durchgeführten Seminars zu den wesentlichen, mit dem Thema Data Warehousing verbundenen Konzepten und Techniken.

In der Einleitung führt Zhenbo Wang das Thema Data Warehousing ein. Es werden die grundlegende Architektur von einem Data Warehouse und die zugehörigen Komponenten vorgestellt und diskutiert.

Jochen Elischberger stellt die neuen Herausforderungen vor, die sich für die Datenmodellierung aus den Anforderungen an ein Data Warehouse ergeben. Es wird einer neues Konzept zur Modellierung — das Stern–Schema — vorgestellt, welches eine Unterscheidung der Tabellen in Fakten– und Dimensionstabellen vornimmt.

Ein schwieriges Problem beim Aufbau einer Data Warehouse–Lösung ist die konzeptionelle Integration von Daten aus den verschiedenen Datenquellen. Ein Konzept zur Integration heterogener Datenquellen — die Mediator–Architektur — wird im Beitrag von Ansgar Zwick behandelt. Das dazugehörige Datenmodell und eine Anfragesprache werden vorgestellt. Einen alternativen Ansatz, die automatische Daten- und Schemaanalyse beschreibt Hans-Jörg Fischer. Insbesondere wird ein Datenmodell vorgestellt, welches in der Lage ist, komplexe semantische Eigenschaften abzubilden. Weiterhin wird auf die Probleme der Datenanalyse im Kontext relationaler Datenbanken eingegangen.

Für den Betrieb eines Data Warehouse ist die Aktualität der darin enthaltenen Daten entscheidend. Inkonsistente Zustände zwischen den Datenquellen und dem Data Warehouse sollen möglichst vermieden und der Aufwand zur Aktualisierung möglichst klein gehalten werden. Diesem Problem widmet sich der Beitrag von Gregor Bublitz, der untersucht, inwieweit bekannte Konzepte zur Sichtenmaterialisierung auf die Problematik anwendbar sind.

Effizienzaspekte beim Zugriff auf die Daten im Data Warehouse werden von Holger Schätzle behandelt. Dabei spielt die Technik der Materialisierung von Sichten und der darauf basierenden Indexstrukturen eine wesentliche Rolle. Falls sich während des Betriebes eines Data Warehouse dessen Datenstruktur ändert, müssen die Daten aufwendig

angepaßt werden. Lösungen für dieses Problem der Evolution von materialisierten Sichten stellt Jürgen Enge in seinem Beitrag vor.

Die analytische Verarbeitung (OLAP) von Daten aus dem Data Warehouse stellt neue Anforderungen an die Speicherung der Daten und die Zugriffsmöglichkeiten. Rainer Ruggaber betrachtet das OLAP–Konzept und stellt Operationen vor, um auf dem mehrdimensionalen Datenwürfel als zentraler Datenstruktur im OLAP operieren zu können. Wie diese Operationen durch spezielle Optimierungstechniken effizient unterstützt werden können, zeigt Stefan Weitland in seinem Beitrag. Mittels einer Gitternetzstruktur wird die Voraussetzung für die mehrfache Nutzung von Anfrageergebnissen geschaffen.

Neben der eher nutzergetriebenen analytischen Verarbeitung von Daten zielen Techniken des Data–Mining auf die automatische, systemgetriebene Erkennung von relevanten Informationen aus großen Datenbeständen. Eine spezielle Technik des Data–Mining, die Generierung von Assoziationsregeln, betrachtet Matthias Barth.

Abschließend werden von Reinhard Sablowski anhand eines Fragenkatalogs aus der Literatur bekannte Data–Warehouse–Projekte vorgestellt und analysiert sowie Erfolgsfaktoren bei der Einführung einer Data–Warehouse–Lösung herausgearbeitet.

Uwe Herzog
Christoph Breitner
Jutta Mülle
Jörg Schlösser

Inhaltsverzeichnis

Kapitel 1

Data Warehousing: Eine Einführung

zhenbo wang

Kurzfassung *Das Konzept „Data-Warehouse" bietet eine umfassende Architektur zur Informationsgewinnung und -bereitstellung für die Entscheidungsunterstützung (*decision support*) im Unternehmen. In diesem Kapitel werden die grundlegende Architektur von einem Data-Warehouse und die zugehörigen Komponenten vorgestellt und diskutiert.*

1.1 Vorwort

Ein Data Warehouse ist ein Lager, das integrierte Informationen für direkte Anfragen und Analysen zur Verfügung stellt. Das Thema über Data Warehousing umfasst Architekturen, Algorithmen, und Tools. Es werden ausgewählte Daten von verschiedenen Datenbasen oder anderen Informationsquellen zu einem Zentrallager zusammengebracht. Das Zentrallager ist das sogenannte Data Warehouse.

Die Kernidee des Data Warehousing Verfahrens ist, daß die wichtigen Informationen vor den Anfragen aus den Infomationsquellen geholt, gefiltert, und integriert werden. In den letzten Jahren wurde Data Warehousing immer populärer im Datenbankbereich. Im Folgendem wird das Konzept von Data Warehousing besprochen, es wird die allgemeine Architektur skizziert, und ein paar technische Fragen diskutiert, die zu diesem Forschungsthema gehören.

1.2 Einführung

Wie soll ein integrierter Zugriff (*integrated access*) auf mehrere, verteilte, heterogene Datenbasen und andere Informationsquellen realisiert werden? Das ist heute ein heißes Thema in der Datenbasisforschung und Datenbasisindustrie geworden. In der Forschungsgemeinschaft basieren die meisten Verfahren für das Problem des Integrierens von Daten auf folgendem zweistufigen Prozeß:

1. Wird eine Anfrage empfangen, wird eine passende Menge von Informationsquellen zu Antwort dieser Frage bestimmt, dann werden Subfragen und Befehle daraus erzeugt und zu bestimmten Quellen weitergeschickt.

2. Ergebnisse werden von der Informationsquellen geholt, diese werden dann umgewandelt, gefiltert, zusammengeschmolzen, und schließlich werden die Ergebnisse an den User oder die Anwendung zurückgegeben.

Man bezeichnet diesen Prozeß als ein lazy oder on-demand Ansatz der Datenintegration, weil die Informationen dann von den Quellen geholt werden, wenn eine Frage gestellt wird. Diesen Vorgang kann auch als eine *Vermittlung* bezeichnet werden. Die Module, die Ergebnisse zerlegen und kombinieren, werden als *Vermittler* (Mediatoren) bezeichnet [15].

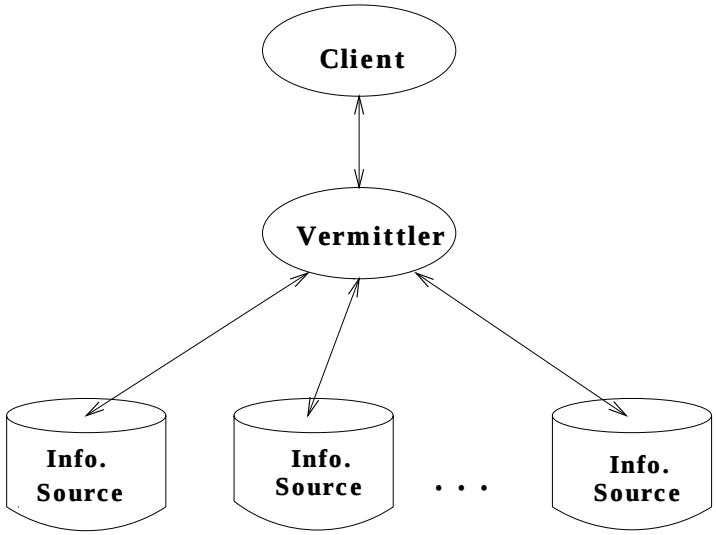


Abbildung 1.1: Basisarchitektur des *Vermittlungssystems*

Beim lazy Ansatz sind nicht nur die Zerlegung der Anfrage und Weitersendung der Subanfragen komplizierte Operationen, es ist auch zeitaufwendig, wenn sehr viele Quellen benötigt oder (und) die Quellen weit entfernt liegen. Eine Alternative zu lazy Ansatz ist *eager* oder *in-advance* Ansatz. In Eager-Ansatz ist der zweistufe Prozeß entsprenchend folgender:

1. Die Informationen in jeder Quelle, die uns interessieren, werden während des Laufzeit geholt, umgewandelt, gefiltert, und mit anderen wichtigen Informationen, die von anderen Quellen stammen, zusammengeschmolzen. Die Informationen werden schließlich in dem Zentrallager (Data Warehouse) gespeichert.

2. Wenn eine Anfrage gestellt wird, wird sie direkt bei dem zentralen Lager beantwortet und braucht nicht mehr auf die originalen Quellen zugreifen.

Dieses Verfahren wird im allgemeinen als Data Warehousing genannt, da das zentrale Lager wie eine Warehouse ist, das interessante Daten speichert.

Ein *lazy Ansatz* zur Integration der Daten ist geeignet für Fragen, die aktuellste Informationen verlangen, für Kunden mit nicht vorhersagbaren Bedürfnissen, und für Fragen, die mit sehr großen Daten und einer ganzen Menge von InformationsQuellen zu tun haben. Leider scheint es so, daß das lazy Ansatz ineffizient, langsam in der Bearbeitung der Anfrage, besonderes wenn ähnliche Anfragen mehrmals gestellt werden oder die Informationssouces langsam arbeiten, teuer und manchmal sogar unerreichbar ist. Es ist auch ineffizient, wenn die wesentliche Arbeit bei der Umwandlung, Filterung und Verschmelzung liegt. In solchen Fällen ist der Lazy Ansatz nicht geeignet.

Bei der Daten Warehouse kann Kunde dagegen direkt auf die integrierten Daten für die Nutzung der Anfrage und Analyse zugreifen. Deswegen ist das Warehousing-Verfahren geeignet für:

1. Der Kunde fordert die spezielle, vorhersagbare Teile von benötigten Informationen.
2. Der Kunde fordert eine schnelle Anfragenbearbeitung(die Daten sind günstig lokal im Warehouse gespeichert), er verlangt aber nicht unbedingt die aktuellste Information.
3. Der Kunde will Kosten sparen. Statt mehrfach auf verschiedene Quellen zugreifen, können die Anfragen schließlich mit einem einzigen Warehouse durchgeführt werden.
4. Der Kunde möchte einen eigene Kopie von der Information, damit er sie ändern, kommentieren und zusammenfassen kann, oder der Kunde möchte die Information einfach speichern, weil sie vielleicht nach einem Zeitraum bei der ursprünglichen Quelle gelöscht werden.

Ein möglicher Nachteil des Warehousing Ansatz ist, daß die Daten physikalisch von den Quellen kopiert werden, dies kostet extra Platz für die Speicherung. Aber in den letzten Jahren wurde die Hardware-Kosten für den Speicher stark reduziert, und außerdem werden die Daten zuerst gefiltert und zusammengefaßt bevor sie in dem Warehouse landen, so glaube ich, ist dies kein ernsthaftetes Problem . Ein wichtigeres Problem ist das Konsistenzproblem des Warehouse. Ein anderer Nachteil ist, daß der "Warehouse-Verwalter" vorher festlegen muß, von welchen Quellen die Daten geholt werden sollen, und welche Daten kopiert und integriert werden sollen.

Das lazy und warehousing Verfahren sind beide realisierbare Lösungen für das Problem Daten zu Integrieren und jede ist geeignet für bestimmte Szenarien.

1.3 Aktuelle Entwicklung und Trends

Die Databankforschung hat sich ursprünglich auf Lazy-Verfahren konzentriert. Seit den letzten Jahren hat die Databasis-Industrie ihre große Interessen schon auf die Data Warehouse gezeigt. Viele führende Softwarehersteller haben sich mit „Data Warehousing–Tools“ beschäftigt, mehrere kleine Firmen haben sich auschließlich auf die Data

Warehousing-Produkte konzentriert. Trotz schneller Fortschritte in kommerziellen Data Warehousing-Tools und Produkten sind die meisten benutzbaren Systeme relativ unflexibel und uneffizient. Ich glaube, daß eine wirklich allgemein effiziente, flexible und realisierbare Data Warehousing Architektur eine Menge von technische Verbesserungen erfordert.

Die Hauptfunktion von Data Warehousing in dem industriellen Abschnitt dient dazu, daß ein Unternehmen seine gesamten Informationen auf einem einzigen Platz sammeln kann zum Zweck einer tiefergehenden Analyse.

Es gibt die Meinung, daß das Data Warehousing Verfahren eigentlich eine Ergängzung für den Lazy Ansatz sein sollte, aber nicht ein Ersatz. Eine anderes vielversprechendes aber noch nicht ausreichend erforschtes Verfahren ist das sogenannte gemischte Verfahren (hybrid approach).Seine Basisarchitektur wird in der Abbildung 1.2 gezeichnet: Das

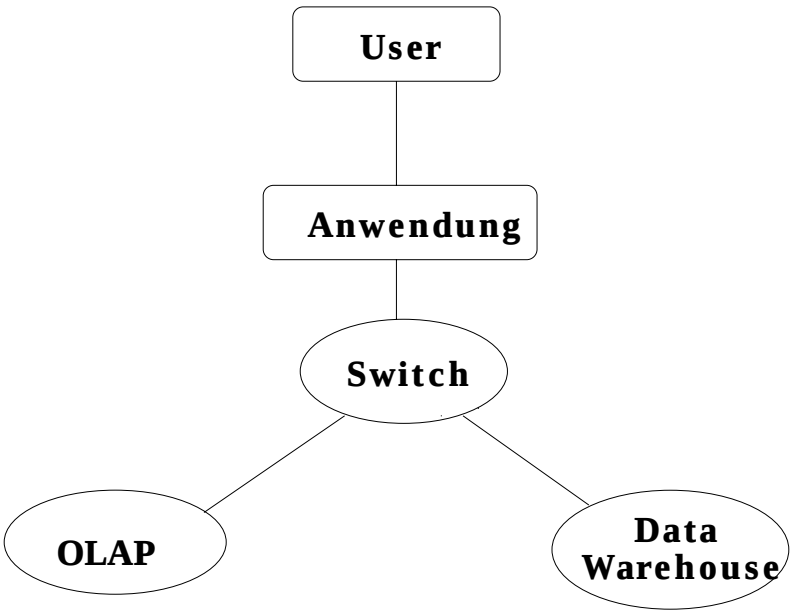


Abbildung 1.2: Basisarchitektur des *gemischten Models*

ist eine Kombination von Lazy Ansatz und Warehousing-Verfahren. Bei diesem Verfahren wird ein Teil der Information in der Zentrale gespeichert, und ein anderer Teil wird während der Anfrageababeitung von den Quellen geholt. Wir konzentrieren uns hier nur auf Data-Warehousing-System.

1.4 Data Warehousing: Systemarchitektur

Abbildung 1.3 zeichnet die Basisarchitektur eines Data Warehousing-Systems. Unten im Diagramm sind die Informationsquellen. Obwohl ihre Gestaltungen alle wie herkömmliche Databasis-Systeme erscheinen, enthalten in vielen Fällen die Informationsquellen nicht normale Datenbasen (z.B relationale Tabellen), sondern flache Daten, Bilder, HTML und SGML Dokumente, Altsystem (legacy Systems) usw. Die Komponenten Wrapper/Monitor setzen sich mit den Informationsquellen in Verbindung.

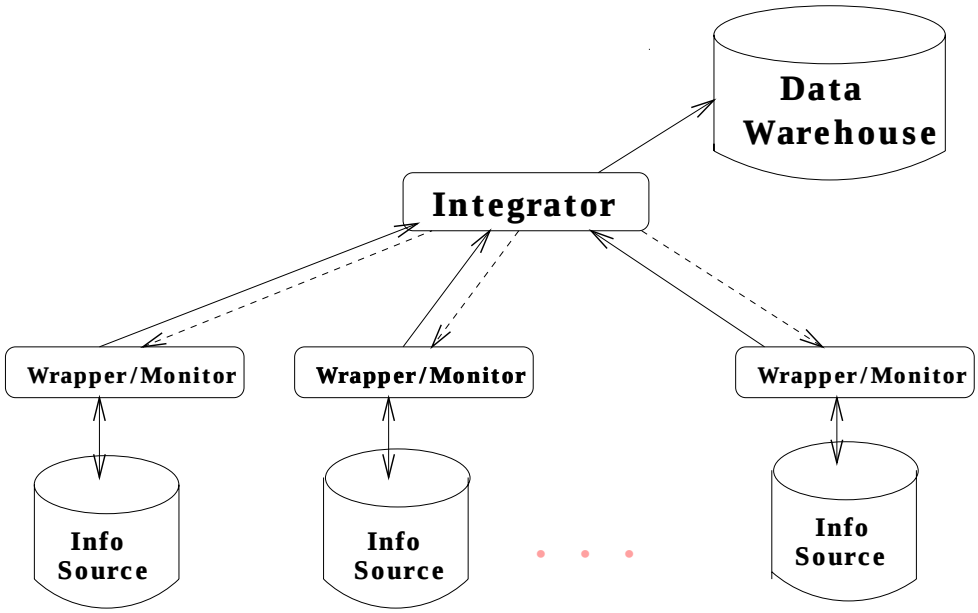


Abbildung 1.3: Basisarchitektur des Data Warehousing Systems

Die Komponente Wrapper (Verpacker) ist zuständig für die Umwandlung des sich in den Quellen befindenden Formats zu dem Format, das im Data Warehouse benutzt wird. Die Komponente Monitor ist zuständig für die automatische Überwachung der für Data Warehouse interessanten Änderungen in den Quellen und benachrichtigt dann den Integrator.

Wenn eine neue Informationsquelle fest an das Warehousing-System hinzugefügt wird, oder eine wichtige Information in der Quelle sich geändert hat, werden die neuen Informationen oder solche Änderungen weiter an das Data-Warehouse fortgeschrieben. Der Integrator ist verantwortlich für die Anbringung der Information in das Data Warehouse. Natürlich kann es sein, dass die Information zuerst gefiltert, zusammengefasst, oder mit aus anderen Quellen stammenden Informationen verschmelzt werden muß. Um eine neue Information in das Data-Warehouse zu bringen, ist es vielleicht noch notwendig, daß der Integrator noch weitere Informationen von gleichen oder anderen Quellen benötigt. Das Verhalten wird in Abbildung 1.3 durch nach unten zeigende Pfeile verdeutlicht.

Obwohl im Abbildung 1.3 nur ein zentralisiertes Data-Warehouse gezeichnet wird, kann natürlich das Warehouse als ein verteiltes Datenbasis-System implementiert werden. In der Praxis ist es wegen Parallelverarbeitung sogar eine Notwendigkeit, um eine angemessene Leistung zu erlangen.

Die Architektur und die Basisfunktionalität werden meist viel allgemeiner beschreiben als die, die in bisherigen kommerziellen Data-Warehouse realisiert sind. Die sich zur Zeit im Einsatz befindenden Data-Warehousing-Systeme setzen voraus, daß sowohl Quellen als auch Warehouse das gleiche Datenmodell (normalerwese relational) benutzen. Also das Fortpflanzen der Daten von Quellen zu den Data-Warehouse ist dort eine reine schubweise Verarbeitung (batch process), und die Frage des Integrators von Informationsquellen stellt sich dort nicht.

1.5 Forschungsprobleme

Basierend auf der im Abschnitt 1.4 beschriebenen allgemeinen Architektur für Data-Warehousing, werden hier eine Reihe von Forsungsproblemen skizziert, die mit dem Warehousing-Verfahren in Zusammenhang stehen.

1.5.1 Wrapper/Monitore

Die im Abbildung 1.3 gezeichnete Wrapper/Monitor-Komponenten haben zwei wesentliche Aufgaben:

1. Umwandlung:

   Durch diese Funktionalität macht es die unterliegende Informationsquellen so, als ob deren Model gleich das in Data Warehousing System angewandte Model wäre. Zum Beispiel, wenn die Informationsquelle aus eine Menge von flachen Daten (flat Files) besteht, aber die warehouse Model relational ist, dann muß der Wrapper/Monitor eine Schnittstelle unterstützen. Durch diese Schnittstelle können wir die Präsentation der Daten von der Informationsquellen als relational betrachten. Das Umwandelungsproblem wird bei meisten Verfahren zu Informations-Integrierung sowohl lazy und eager Ansatz behandelt. Das ist nicht spezifisch für data warehousing. Die Komponente Wrapper, die die Daten von Information Quellen zu einem gemeinsamen Modell umwandelt, kann auch Umwandler genannt werden [12].

2. Überwachen von Änderungen:

   Überwachen die Änderungen der Daten, die für die Data Warehouse von Bedeutung sind, und schreiben diese Änderungen in den Integrator fort. Diese Funktionalität verlässt sich auf die Umwandlung, weil, genau wie die Daten selbst, bei der Änderung der Daten auch deren Format und Model passend umgewandelt werden muß.

Ein anderes Verfahren ignoriert die Änderungsüberwachung und pflanzt einfach alle Daten von Informationsquellen in das Data Warehouse periodisch fort. Der Integrator kann die Daten mit bereits in Warehouse existierenden Daten kombinieren und überschreibt die Daten in dem Warehouse. Ignorierung der Änderungsüberwachung ist vielleicht akzeptierbar in manchen Szenarios, z.B. wenn das Warehouse nicht unbedingt neueste Informationen verlangt, und das Warehouse gelegentlich Off-Line bleibt. Trotzdem, wenn ein Warehouse aktuell, effizient, und ununterbrochen zugriffsbereit sein soll, glaube ich, daß die Überwachung und das Fortpflanzen der Änderung eine bessere Lösung ist.

Die Betrachtung der in der Änderungsüberwachung entstehenden Probleme, wird hier entlang der wichtigsten Typen von Informationsquellen geliedert.

- Kooperative Quelle:

  Die Quelle besitzt einen Triggers oder andere aktive Datenbasisfunktionalitäten [14]. Dadurch wird eine Nachricht bei einer wichtigen Änderung automatisch weitergesendet.

- Quelle mit Protokoll:

  Die Quelle enthält ein Protokoll. Informationen über Änderungen werden dort geschrieben. Also kann die Komponente Monitor einfach das Protokoll abfragen und sich informieren.

- Abfragbare Quelle:

  Die Quelle erlaubt, daß Wrapper/Monitor die Information bei der Quelle nachfragt. Durch periodische Anfragen können die wichtigen Änderungen entdeckt werden.

- Snapshot-Quelle :

  Die Quellen unterstüzt nicht Trigger, Protokoll oder Abfrage, stattdessen unterstützt es nur einen Snapshot. Die Änderung kann dann entdeckt werden, wenn man den aktuellen Snapshot mit dem vorherigen Snapshot vergleicht.

Die Funktionalität jeder dieser Typen zieht interessante Forschungsprobleme in der Änderungsüberwachung nach sich. Zum Beispiel in der kooperativen Quellen, obwohl der Trigger und andere aktive Datenbasen tief erforscht sind, bevor man solche Funktionalitäten in der Warehouse Kontext einsetzt, muß man immer darauf achten, ob diese geeignet für den Umwandler(Wrapper) sind. Ähnlich gilt für Quellen mit Protokollen und abfragbaren Quellen, zusätzlich muß hier noch die Ausführungsfrequenz überlegt werden: wenn die Frequenz zu hoch ist, nimmt die Leistung ab, ist die Frequenz zu niedrig, erhält das Warehouse die nötige Information zu spät. Bei der Snapshot-Quelle bleibt noch ein anderes Problem: wie findet man eine effiziente und realisierbare Methode, um zwei riesige Datenbasen zu vergleichen? Die Snapshot-Methode scheint unelegant zu sein, aber sie wird häufig in der Praxis benutzt [9]. Ein relevantes Problem hierbei ist eine geeignete Repräsentation für die Änderung der Daten zu unterstützen, besonders wenn die Quelle kein relationales Modell benutzt.

Letztlich möchten wir noch erwähnen, daß verschiedene Informationsquellen verschiedene Wrapper/Monitore benötigen. Es ist klar, weil die Funktionalität der Wrapper/Monitore von dem Typ der Quelle abhängig ist. Offensichtlich ist es unerwünscht, wenn in der Praxis für jede Informationsquelle, die bei einem Warehouse-System teilnimmt, ein eigener Wrapper/Monitor implementiert wird, besonders wenn immer mehr interessante Quellen auftauchen. Deshalb wäre eine wichtige Forschungstätigkeit die Schaffung einer Technik und Tools, die automatisch oder semi-automatisch die Implementierung der Wrapper/Monitore unterstützt.

1.5.2 Integrator

Angenommen das Warehouse ist mit seiner ersten Menge von Daten, die von der Information Quellen geholt wurden, gefüllt. Die Tätigkeit des Integrators ist dann, die von Wrapper/Monitor ausgeschickten Änderungsmeldungen zu empfangen und diese Änderungen in das Data-Warehouse zu schreiben. Siehe Abbildung 1.3 .

Wenn wir von einer ausreichend abstrakten Ebene aus das Warehouse betrachten, können die Daten in dem Warehouse als eine materialisierte Sicht oder eine Menge von Sichten

betrachtet werden, und die Basis-daten in der Information Quellen plazieren. Betrachtet man das Problem in dieser Art, ist die Tätigkeit des Integrators im Wesentlichen die Ausführung der materialisierte Sicht Maintenance. In der Tat gibt es eine enge Verbindung zwischen der Sicht-Maintenance und dem Data-Warehouse. Trotzdem können aus verschiedenen Gründen viele herkömmliche Sicht-Maintenance-Techniken nicht angewendet werden. Wegen all dieser Gründe ergibt sich ein Forschungsproblem in Verbindung mit Data-Warehousing:

- In den meisten Data-Warehousing-Szenarios, werden die Sichten viel komplizierter im dem Data-Warehouse gespeichert als in herkömmlichen Sichten, zu mindest gibt es solche Tendenz. Sogar wenn das Warehouse und die Information Quellen beide das relationale Model benutzen, werden zum Beispiel die Sichten, die in dem Warehouse gespeichert sind, wahrscheinlich nicht einfach durch eine Standard-Sicht-Definitionssprache (z.B. SQL) definiert. Das Problem liegt darin, daß das Data Warehouse einen bedeutsamen Anteil von historischen Informationen enthält, und die Information Quellen beinhältet keine solche Informationenen.

- Ein Data-Warehouse tendiert zur Speicherung von Aggregaten und zusammengefaßten Daten. Obwohl in manchen Fällen Aggregat-Daten auch durch eine herkömmliche Sicht-Definitions-Sprache beschreibbar sind. Aber die Schnelligkeit der zuständigen Operators in dieser Sprache ist begrenzt, wir bräuchten eigentlich eine viel schnellere Sicht-Definitions-Sprache. Weiterhin scheint zur Zeit eine effiziente Sicht-Maintenance bei Aggregaten und zusammmengefaßter Information ein offenes Problem geworden zu sein.

- Die Informationsquellen ändern die Basisdaten unabhängig von dem Warehouse, wo die Sichten gespeichert sind, und die basisdaten stammen vielleicht von dem Altssystem (Legacy system) und die können nicht oder wollen nicht an der Sicht Maintenance teilnehmen. Also kooperieren die Informationsquellen nicht zu 100 Prozent mit dem Warehouse. Die meisten materialisierten Sicht-Maintenancestechniken verlassen sich in der Wirklichkeit darauf, daß die Basis Daten Änderung ausschließlich die Sicht-Maintenance verbindet, und die Sicht Modifikation ereignet sich in derselben Transaktion wie die Änderung. Bei der Warehousing-Umgebung ist der allgemeine Fall so:

  -Maintenance des Sichten (der Integrator) des Systems ist nur locker mit der Basis-Daten (die Informationsquellen) verbunden.

  -Die unterliegende Informationsquellen nehmen an der Sicht-Maintenance nicht teil aber sie benachrichtigen über ihre Veränderung.

  Bei diesem Szenario, wird sicher etwas Ungewöhnliches auftauchen wenn wir versuchen die Sichten in Verbindung mit den Basisdaten kosistent zu halten. Und dazu wird der Algorithmus komplizierter als herkömmliche Sicht-Maintenance-Algorithmen.

- Im Data-Warehouse brauchen die Sichten vielleicht nicht jedesmal angepaßt werden, wenn eine Änderung oder eine Menge von Änderungen auf den Basisdaten vorliegt. Nur dann wenn ausreichende Änderungen vorliegen, wird das Warehouse

aktiv. Daher werden für die die Sicht-Maintenance andere Algorithmen als die in der herkömmlichen Sicht-Maintenance benutzten Algorithmen verlangt.

- In der Data warehousing Umgebung ist es vielleicht notwendig, daß die Basis-Daten bevor sie im Warehouse gespeichert werden, zuerst vorverarbeitet (data scubbing) werden. Die Verarbeitung besteht aus Aggregieren, Zusammenfassen und dem Erstellen einer Probe der Daten, um den Umfang des Warehouse zu reduzieren. Bei fehlerhaften Daten soll der Integrator entweder Korrigieren oder Ignorieren, einen Default-Wert einfügen, Duplikate und inkonsistente Daten eliminieren.

Abschließend gilt, daß obwohl die Integratoren vollkommen auf Datenmodellen, die im Data-Warehousing-System eingesetzt werden, basieren können, werden denoch bei jedem Data-Warehouse immer verschiedene Integratoren erforderlich . Der Grund ist, daß verschiedene Mengen von Sichten über verschiedene Basisdaten dort gespeichert werden. Genau wie beim Wrapper/Monitor ist es unerwünscht, dass jeder Integrator ganz von vorne aufgebaut werden muß. Es ist viel besser, eine Technik und Tools zu schaffen, die Integratoren automatisch oder semi-automatisch zu produzieren. Dieses allgemeine Verfahren wird in der herkoemmlichen Sicht-Maintenance benutzt, aber es gibt noch eine Menge von interessanten Problemen zu lösen, um diese Verfahren dem Data-Warehousing anzupassen, diese werden im nächsten Abschnitt diskutiert.

1.6 Data Warehouse: Spezifikation

In dem vorigen Abschnitt wurde die Beziehung zwischen Maintenance der Data Warehouse und Maintenance der materialisierten Sichten skizziert. Es wurde gezeigt, daß es nützlich ist, wenn die Funktionalität für das Spezifizieren eines Integrators auf einem hohen Level erfolgt, anstatt jedesmal für jeden Integrator von grund auf Vorne zu codieren. Deshalb wird in einer idealen Architektur der Inhalt des Data-Warehouse als eine Menge von Sichten-Definitionen spezifiziert. So können die Änderungsaufgaben des Warehouse bei dem Integrator und die Änderungsüberwachungsaufgaben im Wrapper/Monitor automatisch ausgeführt werden.

Für die herkömmliche Sicht-Maintenance, ein Algorithmus wird entwickelt, der die aktive Datenbasis Regeln für die Pflege der SQL–definierten Sichten automatisch erzeugt [2]. Jede Regel wird aktiviert, wenn eine vielleicht auf die Sicht einflußhabende Änderungsmeldung vorkommt. Dann ändert die Regel die Sichten nach einem bestimmten Maß. Ein ähnliches Verfahren kann im Data-Warehouse benutzt werden, wenn dort ein regelgesteuerter Integrator eingesetzt wird. Jede Integratorregel wird durch eine vom Wapper/Monitor geschickte Änderungsmeldung hinaktiviert. Ähnlich wie Sicht-Maintenance-Regeln, müssen auch Integrator-Regeln das Warehouse ändern, um die Änderung der Basisdaten wiederzuspiegeln. Bei dem Warehousing-Szenario, müssen die Regeln jedoch noch kompliziertere Funktionen leisten, z.B. zusätzliche Daten von der Quelle holen Fragen, und eine Vorverarbeitung („scrubbing") der Daten. Trotz der zusätzlichen Komplexitäten bei der Warehousing-Umgebung, soll es immer möglich sein, automatisch oder semi-automatisch die geeigneten Regeln aus der Warehouse-Spezifikation zu produzieren.

Die Forschungsherausforderung in der Schaffung der idealen Architektur ist es, eine Warehouse-spezifische Sprache, Funktionalität der Regeln, Wrapper/Monitor-Schnittstelle und einen geeigneten Algorithmus, der die automatische Erzeugung des Integrators und den relevanten Änderungsüberwachung-Mechanismus unterstützt, zu realisieren. Bei dem WHIPS Data Warehousing Projekt in der Standford Universität wird nach solchen Zielen gestrebt [7].

1.7 Optimierungen

In diesem Abschnitt umreißen wir drei Optimierungen, die die Leistung der im Abschnitt 1.4 beschriebenen Architektur verbessern können:

- Filtern unwichtiger Änderungen in den Quellen.
- Speichern zusätzlicher Daten in dem Warehouse für „Selbst-Maintenance".
- Effizientes Management mehrerer materialisierter Sichten.

1.7.1 Filtern von Änderungen

Bisher wurde gesagt, daß alle Änderungen der Daten in der Quelle, die vielleicht wichtig für das Warehouse sind, von Wrapper/Monitor zu dem Integrator weitergeschickt werden. In einem relationalen Fall, z.B ein Tupel einer Relation R wird geändert, dann müssen alle Sichten, bei der die Relation R teilgenommen hat, entsprechend angepaßt werden. In der Tat bleiben viele Sichten unverändert nach der Duchführung der Änderungsmeldung der Relation R. Die Aufgabe der Filtern ist zu bestimmen, welche Sichten unverändert bleiben nach Ausführung der Änderungsmeldung und dann die Änderungsmeldung für solche Sichten zu filtern. Es ist sicher viel besser, wenn so viel wie möglich Änderungen gefiltert werden, bevor sie zum Integrator geschickt werden.

1.7.2 Aktualisierung von Sichten

Der Integrator empfängt eine Änderungsmeldung Um die Änderung in das Warehouse zu integrieren, braucht der Integrator vielleicht noch zusätzliche Informationen von derselben Quelle oder anderen Quellen. Ein einfaches Beispiel: Wenn das Warehouse zwei Relationen R und S verbindet und auf der Relation R wird ein Tupel eingefügt, dann muß das eingefügte Tupel mit den Inhalten der Relation S verbunden werden. Die entstehende Anfrage zur der Quelle verursacht es vielleicht unvermeidbar, daß die Ausführungszeit verlängert wird.

Wenn bei der Sicht-Maintenance, die zusätzlichen Fragen über Basis-Daten nicht einmal für die Maintenance eines bestimmten Sichten nötig sind, dann besitzt dieses Sicht Selbst-Maintenance-Fähigkeit. Die meisten Sichten besitzen diese Eigenschaften nicht völlig. Die Selbst-Maintenance-Fähigkeit ist durch die Speicherung der zusätzlichen Daten realisiert.

Zum Beispiel in einem extremen Fall werden alle relevanten Daten von den Quellen im Warehouse gespeichert, und die Sichten können völlig wiedergeherstellt werden, wenn es nötig ist. Es scheint, es ist ein offenes Forschungsproblem geworden, wie man die minimale zusätzliche Menge von Informationen bestimmt, die für ein bestimmte Sicht zur Selbst-Maintenance ausreicht. Was uns also interessiert ist, wie man die Kosten der Speicherung der zusätzlichen Daten in dem Warehouse gegen die Kosten für die zusätzliche Anfrage an die Quellen abwägt.

1.7.3 Optimierung mehrerer Sichten

Ein Data-Warehouse enthält vielleicht mehrere Sichten, zum Beispiel, zum Unterstützen verschiedener Arten von Analysen. Wenn solche Sichten Beziehungen zueinander haben, z.B wenn sie auf überlappenden Basisdaten definiert sind, ist es viel effizienter, wenn man nicht alle Sichten materialisiert, sondern nur bestimmte gemeinsame „Subviews", oder ein Teil der Basisdaten, von dem die Sichten des Warehouse hergeleitet werden können. Wenn es so funktioniert, kann dieses Verfahren die Kosten der Speicherung und die Anstrengungen zur Integration der Änderungen der Basisdaten im Warehouse reduzieren. Wegen diesen Sparmaßnahmen muß man vielleicht gegen die verlangsamte Anfrage-Bearbeitung kämpfen, wenn zu viele Sichten nicht voll materialisiert sind.

1.8 Anmerkungen

Wir stellen hier noch ein paare wichtige Streitfragen vor, die von der Data-Warehousing Umgebung stammen:

- Warehouse Management:
  Wir haben uns bis jetzt nur primär auf die Probleme, die sich mit „steady state" (stabil bleibende) Data-Warehousing-Systemen beziehen, konzentriert. Aber die Themen Warehouse-Design, Warehouse-Laden, und Metadaten-Management sind noch wichtiger. In der Praxis, gewinnen der Zeit solche Themen die meiste Aufmerksamkeit der Data-Warehousing Industrie.

- Quellen und Warehouse-Entwicklung:
  Eine Warehousing Architektur muß geschickt auf die Änderung der Informationsquelle reagieren: Das Schema muß geändert werden, wenn eine neue Informationsquelle aufgenommen oder eine alte Informationsquelle entfernt wird. Alle solche Änderungen sollen so wenig wie möglich andere Komponenten des Warehousing Systems stören.

- Duplikate und inkonsistente Informationen:
  Bei mehreren heterogenen Quellen wird man bestimmten Problemen begegnen: gleiche Kopie aus mehreren Quellen (repräsentiert in gleicher oder verschiedener Weise) oder die Informationen sind nicht aufeinander konsistent. In früheren Abschnitten haben wir die Vorverabeitung („scrubbing") der Daten von der einzelnen Quelle beschrieben. Zusätzlich ist es für den Integrator erwünscht, auch die Daten, die aus

mehren Quellen stammen, zu verarbeiten, um Duplikate und inkonsistente Daten so weit wie möglich zu eliminieren.

- Verfallende Information:
  Ein besonderes Merkmal des Data-Warehouse ist es, daß sie sogar manche Informationen aus historischen Gründen noch bewahren, die sich in ihrer ursprünglichen Quelle nicht mehr befinden. Trotzdem es ist auch unerwünscht, daß das Warehouse alle Information für immer beibehält. Hier werden Techniken gebraucht, um die aktuelle Forderung in einer Warehousing-Umgebung festzustellen, und um die unbrauchbaren veralteten Daten aus dem Warehouse wegzuräumen.

1.9 Zusammenfassung

Bei dem Problem, wie man mehrere, distributive, heterogene Informationsquellen integriert, ist Daten-Warehousing eine realisierbare und in manchen Fällen sogar bessere Alternative als traditionelle Lösungen. Traditionelle Verfahren fragen die Quellen an, holen Informationen von den Quellen und verschmelzen diese Informationen nachdem die Fragen gestellt sind. Bei dem Daten-Warehousing Verfahren dagegen werden Informationen ständig durch Anfragen geholt, bearbeitet, und verschmolzen, dadurch sind die Informationen bei direkten Anfragen und Analysen im Warehouse verfügbar.

Obwohl das Konzept für das Daten-Warehousing sehr bekannte in der Datenbasis Industrie ist, glauben wir, daß die oben vorgestellten wichtigen Probleme zuerst gelöst werden müssen, um in der Zukunft das flexible, effiziente, und mächtige Data-Warehousing-System realisieren zu können.

Kapitel 2

Datenmodellierung für Data Warehousing

Jochen Elischberger

Kurzfassung *Viele Unternehmen haben mit Hilfe von Transaktionsverarbeitungs-Systemen eine Unmenge an Daten über ihre Geschäftsabläufe angesammelt. Heute besteht die Herausforderung, aus diesen Daten mittels Entscheidungsunterstützungs-Systemen und Data-Warehousing wertvolle Informationen zu gewinnen. Da Transaktionsverarbeitungs-Systeme aber für Einfüge- und Änderungsoperationen auf atomaren Daten konzipiert wurden, wird ein neues Schema zur Modellierung der Daten benötigt, das eine effiziente Analyse zuläßt, das* ***Stern-Schema****. Im Stern-Schema werden die Daten entsprechend den intuitiven Vorstellungen der Benutzer angeordnet, was eine leichte und schnelle Analyse ermöglicht. Dazu werden die Daten in zwei Gruppen – Fakten und Dimensionen – eingeteilt, die jeweils eigene Tabellen erhalten. Während die Faktentabellen diejenigen Daten enthalten, die analysiert werden sollen, stellen die Dimensionstabellen Beschreibungsklassen für die Fakten dar. Sie enthalten Daten, die den Inhalt der Faktentabellen näher erläutern oder zusammenfassen. Da die Tabellen vollständig denormalisiert sind, wirkt das Schema elegant und übersichtlich. Eine Vielzahl von Variationen beim Entwurf erlaubt eine gute Anpassung an die Anforderungen der Benutzer. Durch die Möglichkeit, in die Dimensionstabellen Attribute für unterschiedliche Detailstufen einzufügen und Tabellen mit vorberechneten Aggregationen zu erstellen, können auch komplexe Abfragen, die eine große Menge an Daten betreffen, schnell beantwortet werden. Für den Benutzer lassen sich Abfragen leicht aufstellen, da es ausreicht, für die Attribute der Dimensionstabellen die gesuchten Werte vorzugeben.*

2.1 Anforderungen an Systeme zur Entscheidungsunterstützung

In den letzten Jahren haben immer mehr Unternehmen Data-Warehouses implementiert, um die von ihnen gespeicherten Daten für Analysten besser zugänglich zu machen und eine effektive Entscheidungsunterstützung zu ermöglichen. Allerdings sind die bisher eingesetzten Systeme und Methoden der Transaktionsverarbeitung nur eingeschränkt für

Data-Warehousing geeignet. Eine Gegenüberstellung der verschiedenen Systeme zeigt die unterschiedlichen Zielsetzungen.

2.1.1 Transaktionsverarbeitungssysteme

Über Jahre hinweg haben Unternehmen mächtige Transaktionsverarbeitungs-Systeme (Online Transaction Processing - OLTP) mit interaktivem Datenzugriff aufgebaut, um Geschäftsaktivitäten wie zum Beispiel Bestellungen oder Verkäufe aufzuzeichnen oder zu automatisieren. Typische Merkmale solcher Systeme sind:

- Hohe Transaktionsrate: Maßeinheit für die Leistung eines solchen Systems ist die sogenannte tps-Rate (transactions per second). Eine Verarbeitungsleistung von mehreren 10.000 tps ist hier keine Seltenheit.
- Einfache und in Transaktionen vorformulierte SQL-Abfragen. Damit können Antwortzeiten vorausberechnet und garantiert werden.
- Detaillierte Daten: Die Daten der Geschäftsvorgänge werden auf atomarer Ebene gesammelt und einzeln gespeichert. Es findet keinerlei Aggregation statt.
- Ständige Veränderungen: Aufgrund der feingranularen Speicherung finden laufend Einfüge-, Änderungs- oder Löschoperationen auf der Datenbasis statt.
- Keine Redundanz: Jedes Datum wird nur einmal abgespeichert, um Änderungsoperationen zu beschleunigen und zu vereinfachen und die Datenintegrität sicherzustellen.

Aufgrund der beiden letztgenannten Punkte sind diese Systeme in der Regel darauf ausgerichtet, viele Einfüge-, Änderungs- oder Löschoperationen vorzunehmen. Dies wird durch relationale Datenbanken erreicht, die weitestgehend normalisiert sind, das heißt meistens in dritter Normalform vorliegen. Dies führt jedoch zu einer starken Zersplitterung der Datenbank in eine Vielzahl von Tabellen und zwischen ihnen bestehenden Verknüpfungen.

2.1.2 Entscheidungsunterstützungssysteme

Entscheidungsunterstützungs-Systeme (Decision Support Systems - DSS) sollen Analysten die Möglichkeit geben, schnell und einfach Informationen (im Gegensatz zu Daten) aus der Datenbank zu erhalten. Um allgemeine Fragen beantworten zu können, müssen oft Daten eines längeren Zeitraums (z.B. Woche, Monat) zusammengefaßt werden.
Typische Merkmale dieser Systeme sind:

- Relativ niedrige Transaktionsrate: Da die Anfragen nur von den Analysten manuell gestellt werden und interaktiv vorgenommen werden, reduziert sich die Anzahl der Transaktionen deutlich.
- Umfangreiche, komplexe SQL-Abfragen: Die SQL-Abfragen finden interaktiv statt, so daß eine Vorformulierung nicht möglich ist.

- Verständlichkeit: Der Benutzer muß die Datenstrukturen leicht verstehen können. Dies wird oft durch Denormalisierung und Aggregation erreicht.
- Seltenere Veränderungen: Die Veränderungen, die an der Datenbank vorgenommen werden, finden kontrolliert und meistens in regelmäßigen Abständen (täglich, wöchentlich, ...) statt.
- Redundanz: Die Denormalisierung führt oft zu erheblicher Redundanz.

2.1.3 Fazit

Der Aufbau der Transaktionsverarbeitungs-Systeme mit der Vielzahl an Tabellen und Verknüpfungen ist für den Benutzer schwer zu verstehen. Dadurch wird eine effektive Analyse der Daten verhindert. Diese Systeme erfüllen somit nicht die Anforderungen, die Entscheidungsunterstützungs-Systeme und Data-Warehousing an Datenbanksysteme stellen. Somit stellt sich die Frage nach einem neuen Datenbankschema, das einerseits leicht verständlich ist und andererseits eine hohe Leistungsfähigkeit besitzt, um damit Entscheidungsunterstützungs-Systeme zu implementieren. Ein Schema, das im Bereich des Data-Warehousing immer mehr an Bedeutung gewinnt, ist das Stern-Schema.

2.2 Stern-Schema

Der Name Stern-Schema kommt daher, daß die graphische Darstellung des Schemas in einem Diagramm die Form eines Sterns besitzt. Das Stern-Schema ist einfach aufgebaut und besteht aus relativ wenigen Tabellen und fest vorgegebenen Verknüpfungspfaden. Es entwickelt sich immer mehr zum Standard für Data-Warehouse-Datenbanken, da es eine ganze Reihe von Vorteilen bietet:

- Eine entsprechend dem Stern-Schema entworfene Datenbank bietet schnelle Antwortzeiten für Abfragen.
- Der Entwurf kann leicht verändert oder erweitert werden, um ihn der Entwicklung des Data-Warehouse anzupassen.
- Der Aufbau entspricht den intuitiven Vorstellungen der Benutzer, so daß diese leicht damit umgehen können.
- Das Stern-Schema erleichtert das Verständnis und die Arbeit mit der Datenbank sowohl für die Benutzer als auch für die Entwickler.

2.2.1 Aufbau

Das Stern-Schema besteht aus zwei Arten von Tabellen, den **Faktentabellen** und den **Dimensionstabellen**. Ein Stern-Schema besitzt mindestens eine Faktentabelle und mindestens eine Dimensionstabelle. Die Faktentabelle enthält diejenigen Daten eines Unternehmens, die analysiert werden sollen. Diese Daten entstehen meist infolge der Geschäftstätigkeit beziehungsweise sind aus diesen Daten abgeleitet. Beispiele sind die Verkaufszahlen

eines Produktes in einer Region oder der Umsatz eines Kunden mit seiner Kreditkarte für eine bestimmte Ware. Diese Daten beinhalten meistens Zahlenwerte, welche als Grundlage für eine Reihe von Operationen zur Verdichtung der Daten dienen, zumeist Summenbildung, Zählung oder Durchschnittbildung. Bei der graphischen Darstellung steht eine Faktentabelle im Mittelpunkt des Sterns. Eine Faktentabelle besteht oft aus vielen Spalten und mehreren Millionen Zeilen, weswegen sie auch *Haupttabelle* (major table) genannt wird.

Die Dimensionstabellen, die auch als *Nebentabellen* (minor tables) bezeichnet werden, sind kleiner und enthalten die Beschreibungen der Fakten, das heißt Daten, die den Inhalt der Faktentabellen näher erläutern oder zusammenfassen. Dimensionstabellen sind somit „Beschreibungsklassen“ für die Fakten. Bei der Repräsentation eines Stern-Schemas in Form eines Diagramms umgeben die Dimensionstabellen die zugehörige Faktentabelle und bilden die Spitzen des Sterns.

Für jede Tabelle ist gemäß den Forderungen des relationalen Modells ein Primärschlüssel definiert, der aus einer oder mehreren Spalten besteht mit denen sich jede Zeile eindeutig bezeichnen läßt. Eine Faktentabelle besitzt in der Regel für jede zugehörige Dimensionstabelle eine Identifikationsspalte, um eine eindeutige Zuordnung zu ermöglichen. Solch eine Spalte, deren Werte zugleich Primärschlüssel der zugehörigen Dimensionstabelle sind, heißt *Fremdschlüssel*.

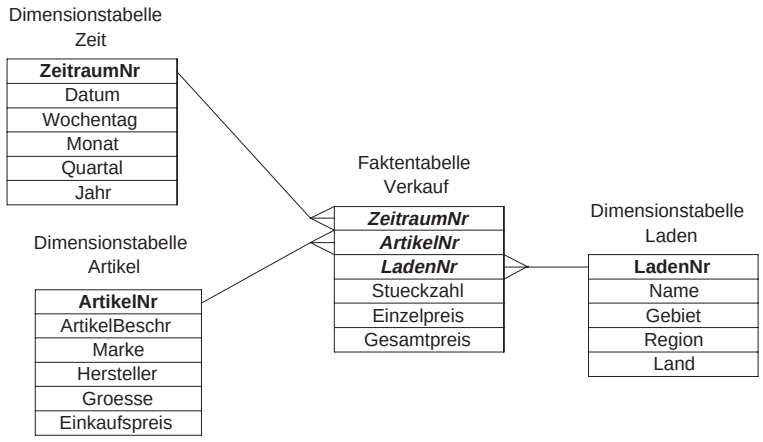


Abbildung 2.1: Verkaufsdatenbank als Beispiel für ein Stern-Schema

Beispiel: In Abbildung 2.1 ist eine Verkaufsdatenbank als Beispiel für ein Stern-Schema dargestellt. Der Primärschlüssel der Faktentabelle Verkauf wird durch die drei Spalten ZeitraumNr, ArtikelNr und LadenNr gebildet. Diese drei Spalten stellen Fremdschlüsselspalten dar, da sie in den entsprechenden Dimensionstabellen als Primärschlüssel dienen. Über diese Fremdschlüssel werden Faktentabelle und Dimensionstabellen in eine n:1-Verknüpfung gesetzt, das heißt, daß Werte, die in der Faktentabelle mehrfach auftreten können, in der zugehörigen Dimensionstabelle genau einmal vorkommen. So besitzt zum Beispiel jedes Produkt, das in der Dimensionstabelle Artikel steht, genau eine ArtikelNr. Diese ArtikelNr kann in der Verkaufstabelle mehrfach auftreten, nämlich bei Verkäufen des gleichen Produktes in verschiedenen Läden und zu unterschiedlichen Zeiten.

Tabellen-Attribute

Die Spalten der Dimensionstabellen, die keine Schlüssel sind, werden als **Dimensions-Attribute** bezeichnet. Diese ermöglichen es, den Inhalt der einzelnen Dimensionstabellen auf einer höheren Ebene zusammenzufassen oder die einzelnen Dimensionen genauer zu beschreiben. Da es oft sehr viele Aspekte gibt, unter denen die Dimensionen betrachtet werden, sind Dimensionstabellen mit mehr als hundert Dimensions-Attributen keine Seltenheit. Bei obigem Beispiel ist Region in der Dimensionstabelle Laden ein Attribut, das jedem Laden die Region zuordnet, in der er liegt. Bei Abfragen können dann ganz bestimmte Regionen ausgewählt werden.

Denormalisierung

Eine vollständig normalisierte Datenbank, wie sie für die online Transaktionsverarbeitung verwendet wird, besteht aus dutzenden oder hunderten von Tabellen und macht es somit nahezu unmöglich, die enthaltenen Daten zu analysieren. Ein denormalisiertes Schema ist dagegen einfach und elegant. Es entspricht den natürlichen Vorstellungen der Benutzer und erleichtert somit das Verständnis. Die Anzahl der Verknüpfungen, die bei einer durchschnittlichen Abfrage verarbeitet werden, ist deutlich reduziert was zu einer Verbesserung der Leistungsfähigkeit der Datenbank führt. Da eine Aktualisierung der Daten nur in regelmäßigen, größeren Abständen vorgenommen wird, sind die negativen Auswirkungen der Denormalisierung begrenzt.

Die Dimensionstabellen des Stern-Schemas sind daher in der Regel komplett denormalisiert. Beispielsweise ist die Tabelle Laden nicht in dritter Normalform, weil zwischen den Nichtschlüsselattributen Gebiet und Region sowie Region und Land eine funktionale Abhängigkeit besteht. Die denormalisierten Dimensionstabellen können aus der vollständig normalisierten Struktur der Transaktionsverarbeitungs-Systeme hervorgehen. Eine Variante des Stern-Schemas, bei der die Dimensionstabellen nicht denormalisiert sind, ist das Schneeflocken-Schema, das im folgenden Abschnitt genauer betrachtet wird.

2.2.2 Verschiedene Arten des Stern-Schemas

Einfaches Stern-Schema

Beim einfachen Stern-Schema ist der Primärschlüssel der Faktentabelle aus einem oder mehreren Fremdschlüsseln zusammengesetzt und zwar ausschließlich aus Fremdschlüsseln. Das Beispiel aus Abbildung 2.1 ist ein einfaches Stern-Schema, da die drei Fremdschlüssel ZeitraumNr, ArtikelNr und LadenNr den Primärschlüssel der Faktentabelle bilden.

Viel-Stern-Schema

Es gibt Fälle, in denen die Verkettung der Fremdschlüssel nicht ausreicht, um jede Zeile der Faktentabelle eindeutig zu kennzeichnen. Diese Anwendungen verlangen ein Viel-Stern-Schema. Ein Viel-Stern-Schema ist daher im Gegensatz zum einfachen Stern-Schema da-

durch charakterisiert, daß es über einen Primärschlüssel verfügt, der nicht nur aus Fremdschlüsseln besteht. Oftmals ist kein einziger Fremdschlüssel Teil des Primärschlüssels. Die Fremdschlüssel dienen dann ausschließlich der Referenzierung von Dimensionstabellen.

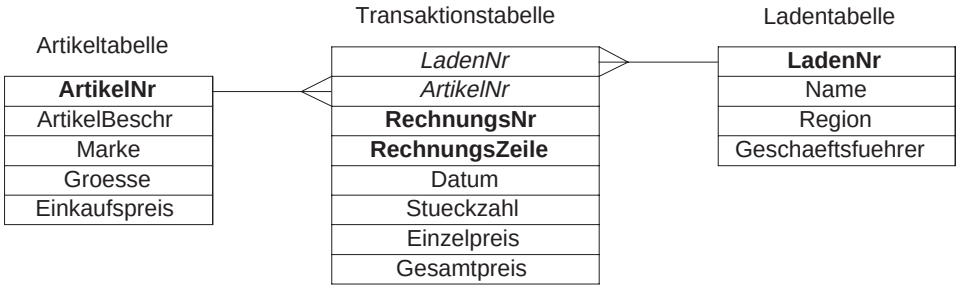


Abbildung 2.2: Beispiel für ein Viel-Stern-Schema

Beispiel: Abbildung 2.2 zeigt eine Verkaufsdatenbank als Beispiel für ein Viel-Stern-Schema. Es mußte ein Viel-Stern-Schema verwendet werden, da die Fremdschlüssel LadenNr und ArtikelNr nicht ausreichen, eine Zeile der Transaktionstabelle eindeutig zu identifizieren. Der Primärschlüssel für die Faktentabelle besteht aus den zwei Spalten RechnungsNr und RechnungsZeile. Diese Schlüssel ermöglichen eine eindeutige Kennzeichnung jeder Zeile. Die Fremdschlüssel der Faktentabelle sind LadenNr und ArtikelNr, die die Artikel- und Laden-Dimensionstabelle referenzieren.

Es ist zu beachten, daß beim Viel-Stern-Schema der durch die Verkettung der Fremdschlüssel der Faktentabelle gegebene Wert in mehreren Zeilen auftauchen kann und somit die eindeutige Kennzeichnung einer Zeile nicht mehr zuläßt. So kann bei obigem Beispiel ein Laden den gleichen Artikel mehrmals täglich verkaufen. Jede Zeile wird stattdessen eindeutig durch die Primärschlüssel RechnungsNr und RechnungsZeile gekennzeichnet.

Mehrfache Faktentabellen

Ein Stern-Schema kann auch mehrere Faktentabellen enthalten. Dies ist beispielsweise dann der Fall, wenn die Fakten in den einzelnen Tabellen nicht direkt miteinander in Beziehung stehen, zum Beispiel wenn sie aus unterschiedlichen zeitlichen Erhebungen stammen. In Abbildung 2.3 sind beispielsweise zwei getrennte Faktentabellen für die Verkäufe der Jahre 1995 und 1996 realisiert.

Ein weiterer wichtiger Grund, mehrere Faktentabellen anstelle einer einzigen zu halten, ist der Wunsch, dadurch die Leistung verbessern zu können. So ist es oft sinnvoll, mehrere Faktentabellen zu haben, in denen die einzelnen Daten unterschiedlich stark zusammengefaßt sind, wie zum Beispiel getrennte Faktentabellen für die täglichen, wöchentlichen und monatlichen Verkäufe, wie dies in Abbildung 2.4 zu sehen ist. Somit kann jede Abfrage in einer Tabelle mit angemessener Genauigkeit und daraus resultierend optimaler Geschwindigkeit stattfinden.

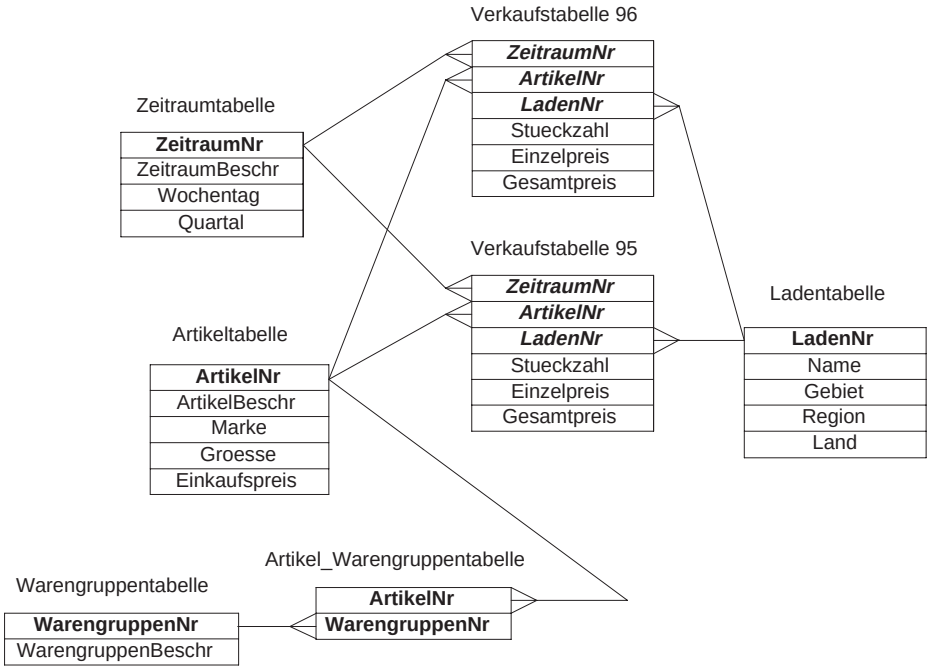


Abbildung 2.3: Stern-Schema mit mehreren Faktentabellen und Assoziativtabelle

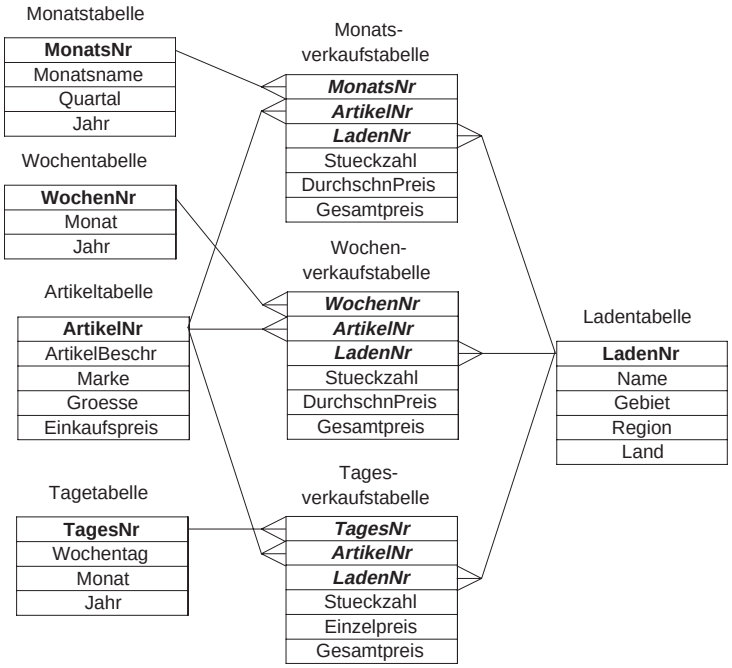


Abbildung 2.4: Getrennte Faktentabellen für tägliche, wöchentliche und monatliche Verkäufe

Bestehen zwischen zwei Dimensionstabellen m:n-Beziehungen, so schlagen sich diese meistens ebenfalls in Form von zusätzlichen Faktentabellen nieder. Diese Tabellen erhalten den besonderen Namen **Assoziativtabelle**. Im Beispiel der Verkaufsdatenbank gehört jedes Produkt zu einer oder mehreren Warengruppen. Außerdem enthält jede Warengruppe viele Produkte. Diese m:n-Verknüpfung kann durch eine Faktentabelle dargestellt werden, die die möglichen Kombinationen von Produkten und Warengruppen festlegt. Dies ist ebenfalls in Abbildung 2.3 zu sehen.

Schneeflocken-Schema

Durch die Weiterentwicklung der Datenzugriffswerkzeuge (data access tools), die zwischen den Benutzer und die eigentliche Datenhaltung treten und damit die tatsächliche Realisierung verdecken können, kann der physikalische Aufbau variiert und dadurch mehr Flexibilität erreicht werden. Eine Variation des Stern-Schemas ist es, alle Dimensionsinformationen in dritter Normalform zu speichern, während die Faktentabelle(n) unverändert bleiben. Diese Art des Stern-Schemas wird **Schneeflocken-Schema** genannt. Wenn die Benutzer allerdings direkt mit der physikalischen Tabellenstruktur arbeiten, sollte darauf geachtet werden, die Anzahl der Tabellen möglichst gering zu halten, um Verwirrung zu verhindern.

2.3 Abfragen im Stern-Schema

Für effektive und sinnvolle Abfragen muß dem Benutzer der Inhalt der Datenbank genau bekannt sein. Eine einfache Möglichkeit, die zulässigen Werte für bestimmte Attribute herauszufinden, sind Abfragen in der zugehörigen Dimensionstabelle. Durch die Abfrage

```
SELECT Name FROM Laden;
```

wird beispielsweise eine Liste aller Läden erstellt, die in der Ladentabelle aus Abbildung 2.1 enthalten sind. Durch solche und ähnliche Abfragen erhält der Benutzer einen guten Überblick über die vorhandenen Dimensionen. Diese Kenntnis der Werte ermöglicht es ihm dann, konkrete Abfragen an die Faktentabellen zur Analyse der Daten zu stellen. Auch bei komplexen Abfragen reicht es oft aus, die Werte für die betroffenen Dimensions-Attribute festzulegen beziehungsweise den Wertebereich entsprechend einzuengen. Die Suche in den Dimensionstabellen ist effizienter, da diese viel kleiner als die Faktentabellen sind und der Aufwand somit deutlich reduziert ist. Eine Abfrage für das Stern-Schema in Abbildung 2.1 könnte beispielsweise folgendermaßen lauten:

```
SELECT  ArtikelBeschr, Stueckzahl
FROM    Zeit z, Verkauf v, Laden l, Artikel a
WHERE   Quartal = 1
AND     Jahr = 1996
AND     Name = Karlsruhe1
AND     Einkaufspreis > 1000
AND     v.ArtikelNr = a.ArtikelNr
AND     v.ZeitraumNr = z.ZeitraumNr
AND     v.LadenNr = l.LadenNr;
```

Dadurch erhält man die Beschreibung aller Artikel, deren Einkaufspreis höher als 1000 DM war und die im ersten Quartal 1996 im genannten Laden verkauft wurden, sowie die jeweils verkaufte Stückzahl.

2.3.1 Aggregation

Unter **Aggregation** versteht man das Ansammeln und Zusammenfassen von Daten, die bestimmte Attribute erfüllen. Dafür ist es vorteilhaft, wenn die Daten der Faktentabelle Zahlenwerte sind, da diese mittels Addition oder Durchschnittbildung leicht zusammengefaßt werden können. Allerdings muß beachtet werden, daß Fakten nicht bezüglich aller Dimensionen gleichermaßen aggregiert werden können. Während Mitarbeiterzahlen von einzelnen Läden beispielsweise problemlos addiert werden können, ist es sinnlos, die Mitarbeiterzahlen verschiedener Zeitpunkte zusammenzuzählen. Eine sinnvolle SQL-Abfrage lautet beispielsweise

```
SELECT    ArtikelBeschr, sum(Stueckzahl), avg(Einzelpreis)
FROM      Zeit z, Verkauf v, Laden l, Artikel a
WHERE     Quartal = 1
AND       Jahr = 1996
AND       ArtikelBeschr = 'Waschmaschine'
AND       Region = 'Hessen'
AND       v.ArtikelNr = a.ArtikelNr
AND       v.ZeitraumNr = z.ZeitraumNr
AND       v.LadenNr = l.LadenNr;
GROUP BY  ArtikelBeschr;
```

Man erhält die Anzahl aller im ersten Quartal 1996 in Hessen verkauften Waschmaschinen sowie den durchschnittlichen Preis.

Beim Entwurf einer Datenbank müssen Entscheidungen darüber getroffen werden, ob Aggregationen vorausberechnet werden und diese vorberechneten Daten im Data-Warehouse abgespeichert, das heißt materialisiert werden. Die wichtigsten Gründe für das Materialisieren von Aggregationen sind die Verbesserung der Performance bei Abfragen und die Verminderung der notwendigen Prozessortakte.

Es ist allerdings sinnlos, eine bestimmte Materialisierung anzulegen, wenn das Erstellen längere Zeit (mehrere Stunden) in Anspruch nimmt und das Ergebnis unter Umständen

nur einmal innerhalb eines ganzen Jahres abgefragt wird. Wenn andererseits eine Aggregation von vielen Benutzern pro Tag verwendet wird, bringt es viele Vorteile, diese vorzuberechnen und abzuspeichern. Es ist aber nicht notwendig alle möglichen Attributkombinationen aggregiert zu speichern. Nicht nur die Häufigkeit, mit der die Benutzer eine Aggregation benutzen, sondern auch die mögliche Abnahme der Zeilen ist ein wichtiges Kriterium bei der Entscheidung, welche Aggregationen vorgenommen werden sollen.

Wenn eine Faktentabelle zum Beispiel 10.000.000 Zeilen enthält und durch eine Aggregation eine Tabelle mit 9.000.000 Zeile entsteht, so ist diese Aggregation weit weniger sinnvoll als eine andere, bei der 1.000.000 Zeilen übrigbleiben. Das Zusammenfassen von Daten auf ausgewählten Stufen wird als *dünne* (sparse) Aggregation bezeichnet. Es ist wichtig, geeignete Stufen zu wählen, um neben der Performance auch den Bedarf an Plattenspeicherplatz zu optimieren.

2.3.2 Attribut-Hierarchien

In vielen Fällen ist es notwendig, die Daten auf unterschiedlichen Detailstufen zu betrachten, um eine optimale Analyse zu erreichen. So kann es vorkommen, daß man die Daten zuerst auf einer sehr stark aggregierten Ebene betrachtet und anschließend einzelne Aspekte genau analysiert und dazu eine niederere, detailliertere Stufe wählt. Dieser Vorgang wird Abrollen (drill down) genannt. Das entgegengesetzte Vorgehen, nämlich zuerst detailliertere Daten zu betrachten und diese dann immer mehr zusammenzufassen, wird Aufrollen (roll up) genannt.

So will man beispielsweise zunächst die Verkäufe eines bestimmten Produktes im ganzen Land betrachten und später zur genaueren Untersuchung, die verkauften Stückzahlen des Produktes in den einzelnen Regionen vergleichen. Falls dabei auffällt, daß in bestimmten Regionen die Verkaufszahlen überdurchschnittlich hoch sind, ist es naheliegend, in diesen Regionen jeden Laden einzeln zu betrachten, um die Ursache dafür aufzudecken.

Beim Entwurf des Stern-Schemas muß von Anfang an darauf geachtet werden, daß die Dimensionstabellen die entsprechenden Attribute zum Aufbau von Hierarchien enthalten. In einer Tabelle werden oft auch mehrere Hierarchien aufgestellt. So ist bei der Unterteilung in Gebiete und Regionen oft eine Unterscheidung zwischen geographischen und organisatorischen Grenzen notwendig, da diese oft nicht übereinstimmen.

Im Beispiel aus Abbildung 2.1 sind in allen drei Dimensionstabellen Attribut-Hierarchien definiert. Für die Dimension Laden ist ein Aufrollen von Laden über Gebiet und Region zu Land möglich. In der Artikeltabelle können einzelne Artikel, die derselben Marke angehören, zusammengefaßt werden. Mehrere Marken können dann einem Hersteller zugeordnet werden. In der Dimensionstabelle für die Zeit stellt Jahr die oberste Verdichtungsstufe dar. Ein Jahr setzt sich aus vier Quartalen zusammen, von denen jedes drei Monate umfaßt. Die detailliertesten Informationen enthält schließlich das Attribut Datum.

Das Attribut Wochentag dagegen ist nicht Teil der Hierarchie sondern dient lediglich der genaueren Beschreibung eines Datums. Entsprechendes gilt für die Attribute Groesse und Einkaufspreis in der Artikeltabelle. Diese Spalten, die auch *erweiterte* Attribute genannt werden, haben keinen Einfluß auf die Körnigkeit der Daten.

2.3.3 Materialisierung von Aggregationen

Entscheidet man sich, die häufig abgefragten Aggregationen zu materialisieren, so existieren verschiedene Möglichkeiten, diese in das Stern-Schema einzufügen. Eine Variante ist es, für die unterschiedlichen Aggregationsstufen getrennte Faktentabellen aufzustellen. Dies führt dazu, daß für die einzelnen Ebenen der Attribut-Hierarchie auch getrennte Dimensionstabellen erstellt werden, die dann jeweils mit der zugehörigen Faktentabelle verknüpft sind. Abbildung 2.4 zeigt dies am Beispiel der Verkaufsdatenbank, bei der Aggregationen für die Verkäufe einer Woche und eines Monats vorgenommen wurden.

Eine andere Möglichkeit besteht darin, aggregierte und nicht aggregierte Daten in einer gemeinsamen Faktentabelle zu speichern. Um die Daten zu unterscheiden und einen gezielten Zugriff auf die einzelnen Hierarchie-Ebenen zu ermöglichen, erhalten die Dimensionstabellen ein zusätzliches Attribut Stufe, das den Namen der jeweiligen Hierarchie-Ebene enthält. Bei aggregierten Daten sind die Dimensions-Attribute, die sich auf detailliertere Daten beziehen, undefiniert (NULL). Abbildung 2.5 zeigt Beispiele für verschiedene Tu-

LadenNr	Name	Gebiet	Region	Land	Stufe
245	Karlsruhe1	Nordbaden	BaWue	Deutschland	Laden
168	Karlsruhe2	Nordbaden	BaWue	Deutschland	Laden
257	Stuttgart1	Nordwuerttemberg	BaWue	Deutschland	Laden
363	NULL	Nordwuerttemberg	BaWue	Deutschland	Gebiet
284	NULL	Suedbaden	BaWue	Deutschland	Gebiet
984	NULL	NULL	Hessen	Deutschland	Region
349	NULL	NULL	NULL	Frankreich	Land

Abbildung 2.5: Dimensionstabelle Laden mit Attribut Stufe

pel der Dimensionstabelle Laden, die unterschiedlichen Aggregationsstufen angehören. Bei Abfragen muß darauf geachtet werden, daß die gesuchten Attribute auf der betreffenden Stufe auch definiert sind.

2.4 Laufzeit

Ein großes Problem des Stern-Schema-Konzeptes ist die Leistungsfähigkeit bei der Beantwortung von Abfragen. Da viele Datenbasisverwaltungssysteme bisher für Transaktionsverarbeitungs-Systeme eingesetzt wurden, treten beim Übergang zu Entscheidungsunterstützungs-Systemen einige Probleme auf.

2.4.1 Problematik bei paarweisen Verbindungen

Die bisher benutzten Systeme können meist nur zwei Tabellen auf einmal verbinden. Da die meisten Abfragen an Entscheidungsunterstützungs-Systeme aber mehrere Tabellen betreffen, müssen diese Abfragen jeweils in eine Reihe von paarweisen Verbindungen unterteilt werden. Die Operation wird somit sehr teuer, da viele Zwischenergebnisse erzeugt werden, die meist auch sehr groß sind.

Eine zusätzliche Schwierigkeit ist die Wahl der Reihenfolge, in der die paarweisen Verbindungen vorgenommen werden. Da bei unterschiedlichen Reihenfolgen die Größe der Zwischenergebnisse sehr stark variieren kann, wirkt sich die Reihenfolge entscheidend auf die Kosten aus. Es ist allerdings fast unmöglich, eine optimale Lösung zu finden, da es für die paarweise Verbindung von n Tabellen $n!$ verschiedene Reihenfolgen gibt. Die Anzahl der möglichen Kombinationen wächst somit exponentiell mit der Zahl der zu verbindenden Tabellen.

Um dieses Problem zu umgehen, ziehen die meisten Datenbasisverwaltungssysteme zur Verbindung nur solche Tabellen in Betracht, die direkt miteinander verknüpft sind. Für ein Stern-Schema ist diese Vorgehensweise aber ungeeignet, da fast ausschließlich Faktentabellen mit Dimensionstabellen verknüpft sind. Somit würde die erste paarweise Verknüpfung höchstwahrscheinlich eine Faktentabelle betreffen. Da die Faktentabellen aber die mit Abstand größten Tabellen sind, führt dieses Vorgehen zu einem sehr großen Zwischenergebnis, welches wiederum die Laufzeit für die Ausführung der nachfolgenden Verbindungen verlängert.

2.4.2 Kartesisches Produkt

Das Verbinden von benachbarten Tabellen im Stern-Schema sollte aus den oben genannten Gründen möglichst vermieden werden. Es stellt sich somit die Frage nach einer effizienteren Vorgehensweise. Eine Verbesserung kann erreicht werden, wenn für die paarweise Verbindung Tabellen ausgewählt werden, die nicht benachbart sind. Diese Operation ist die Bildung des Kartesischen Produktes zweier Relationen, die jedes Tupel aus der einen mit jedem Tupel aus der zweiten Relation zusammensetzt. Diese sehr teure Operation wird von Datenbasisverwaltungssystemen normalerweise vermieden kann aber bei einem Stern-Schema Vorteile bringen. Da die Dimensionstabellen viel kleiner sind als die zugehörige Faktentabelle, ist es besser, zuerst die Kartesischen Produkte zwischen den Dimensionstabellen zu bilden und die Verbindung mit der großen Faktentabelle am Schluß vorzunehmen.

Beispiel: Angenommen im Stern-Schema aus Abbildung 2.1 besitzt die Tabelle Zeit 300 Zeilen, die Tabelle Artikel 500 Zeilen, die Tabelle Laden 200 Zeilen und die Faktentabelle eine Million Zeilen. Eine Abfrage wählt beispielsweise aus Zeit 30, aus Artikel 50, aus Laden 20 und aus der Faktentabelle schließlich 1.000 Zeilen als Endergebnis aus. Wird diese Abfrage mit der Methode des paarweisen Verbindens bearbeitet, so würden bei der Verbindung von Artikel mit Verkäufen vielleicht 100.000 Zeilen entstehen, bei der Verbindung dieser 100.000 Zeilen mit der Zeittabelle vielleicht 10.000 und bei der letzten Verbindung mit der Ladentabelle schließlich 1.000 Zeilen. Insgesamt würden also 111.000 Zeilen erzeugt.

Bei der Bildung des Kartesischen Produktes von 50 Artikeln, 20 Läden und 30 Zeiten entstehen 30.000 Zeilen. Wenn diese dann mit der Verkaufstabelle verbunden werden, erhält man die resultierenden 1.000 Zeilen. Somit werden also 31.000 Zeilen erzeugt, was einer deutlichen Leistungsverbesserung gegenüber der ersten Methode entspricht.

2.4.3 Indizes und Aggregation

Da aber die Bildung des Kartesischen Produktes bei großen Dimensionstabellen keine akzeptable Lösung darstellt, ist es sinnvoll noch weitere Techniken zur Beschleunigung der Abfragebearbeitung einzusetzen. Eine Möglichkeit, die Leistungsfähigkeit zu verbessern, ist das Verwenden von Indizes. Um die Laufzeit bei Verbindungen von Dimensionstabellen mit der Faktentabelle zu verringern, können die Fremdschlüssel der Faktentabelle und die zugehörigen Primärschlüssel der Dimensionstabellen indiziert werden. Somit erhält man bei Selektionen in den Dimensionstabellen Indexwerte, die es erlauben, die zugehörigen Zeilen in der Faktentabelle sehr schnell zu identifizieren.

Da sich viele Abfragen auf eine hohe Aggregationsstufe beziehen, ist es oft von Vorteil, Aggregationen in getrennten Faktentabellen zu materialisieren. Die Abfragen, die aggregierte Daten betreffen, können dann mittels den entsprechenden Tabellen, die deutlich kleiner als die Faktentabellen auf detaillierteren Stufen sind, beantwortet werden. Die Anzahl der zu verarbeitenden Zeilen wird dadurch deutlich reduziert und es können bedeutende Laufzeitverbesserungen erreicht werden.

2.5 Vorgehen zum Aufstellen eines Stern-Schemas

Da viele Unternehmen jahrelang Transaktionsverarbeitungs-Systeme verwendeten, stellt sich heute oft die Aufgabe, die darin enthaltenen Daten für Analysten und Entscheidungsträger in Form eines Stern-Schemas für Data-Warehousing zugänglich zu machen. Der Übergang muß sehr sorgfältig geplant werden und sollte in mehreren Schritten erfolgen. Eine mögliche Vorgehensweise soll hier sowohl allgemein als auch an einem Beispiel vorgestellt werden (vergleiche [19]).

2.5.1 Analyse

Die erste Aufgabe besteht darin, die zentralen und relevanten Faktoren der Geschäftstätigkeit, die später analysiert werden sollen, zu finden und anzuordnen. Dazu kann eine Liste der Begriffe erstellt werden, die für die Situation des Unternehmens und für die Analyse von Bedeutung sind. Dabei sollten Namen verwendet werden, mit denen man durch tagtägliche Arbeit vertraut ist, damit sie später bei der Formulierung der Abfragen leichter wiedererkannt werden können. Oft können diese Begriffe in Geschäftsberichten oder -tabellen gefunden werden. Es ist sinnvoll, immer eine kurze Beschreibung des gefundenen Begriffs mit anzugeben.

Als Beispiel wird die Datenbank einer Hotelkette betrachtet. Eine Liste der wichtigen Begriffe könnte folgendermaßen aussehen:

Belegungen:	Momentan belegte Zimmer
Kunden:	Kunden der Hotelkette
Reservationen:	Vorliegende Zimmerreservationen
Hotels:	Alle Hotels, die zur Kette gehören
Zimmer:	Verschiedene Arten von Zimmern

Als nächstes muß die Rolle, die diese Begriffe im Geschäftsablauf inne haben, bestimmt werden. Dazu sollte eine Aufteilung in zwei Kategorien vorgenommen werden: Zum einen die zentralen Geschäftsaktivität(en) und zum anderen die einzelnen Faktoren, die diese beinflussen. Diese Faktoren sind meistens Personen, Orte oder Sachen, die direkt an der Aktivität beteiligt sind.

Im obigen Beispiel stellen die augenblicklich belegten Zimmer sowie die Zimmerreservationen die zentralen Aktivitäten dar. Die Faktoren, die diese Aktivitäten genauer charakterisieren, sind die Kunden, die ein Zimmer belegen beziehungsweise reservieren, die Hotels, in denen die jeweiligen Zimmer liegen, sowie der Typ der Zimmer.

2.5.2 Implementierung

Um die gefundenen Begriffe im Entscheidungsunterstützungs-System zu repräsentieren, muß nun jedem Begriff eine Tabelle oder zumindest eine Spalte einer Tabelle zugeordnet werden. Ein zentraler Geschäftsvorgang wird in einer Faktentabelle abgespeichert. Diese Faktentabelle muß Spalten für alle Begriffe, die direkt mit der Aktivität verbunden sind, sowie Identifikationsnummern für alle Faktoren, die die Aktivität beinflussen, enthalten.

Für diese einzelnen Faktoren werden jeweils Dimensionstabellen erstellt. Dabei sollte jede Person, jeder Ort und jede Sache, die an der Aktivität der Faktentabelle beteiligt ist, eine eigene Dimensionstabelle erhalten. Die Spalten der Dimensionstabellen enthalten alle Begriffe und Informationen, die den Inhalt genauer beschreiben, sowie Identifikationsnummern, um die Beziehung zur Faktentabelle herzustellen. Dazu ist es notwendig, noch einmal eine genaue Analyse der Geschäftstätigkeit und der sie beeinflußenden Faktoren vorzunehmen, um zusätzliche Attribute zur genaueren Beschreibung der Dimensionen und zum Aufstellen von Hierarchien zu erhalten.

Im Beispiel der Hoteldatenbank werden zwei Faktentabellen für die momentan belegten Räume und für die Reservationen angelegt. Es entsteht somit ein Stern-Schema mit mehreren Faktentabellen. Dimensionstabellen werden aufgestellt für die Hotels, die Kunden und die Zimmer. Für die Hotels werden als Attribute die Adresse (Strasse, PLZ, Stadt), die Anzahl an Zimmern (#Zimmer) sowie der Name des Hotels (Hotelname) festgelegt. Attribute für die Kundentabelle sind ebenfalls die Adresse (Strasse, PLZ, Stadt) und der Name des Kunden (Kundenname) und außerdem die Nummer der Kreditkarte (KreditkartenNr). Die Zimmer werden genauer beschrieben durch die Anzahl an Betten (#Betten), eine Angabe über den Komfort (ZimmerBeschr) und ein Attribut das angibt, ob in dem Zimmer Rauchverbot besteht (Raucherstatus). Die Faktentabellen erhalten noch Spalten für die Anzahl der Nächte die in einem Hotel verbracht werden (#Naechte) und den dafür anfallenden Preis (Preis).

Eine weitere Dimensionstabelle, die in der Regel jedes Stern-Schema besitzt, ist eine Tabelle für die Zeit beziehungsweise für bestimmte Zeiträume. Dies beruht auf der Tatsache, daß sehr oft verschiedene Zeitabschnitte miteinander verglichen werden, um Geschäftsentwicklungen aufzuzeigen. Desweiteren ist es fast immer notwendig, die Daten eines längeren Zeitraumes (zum Beispiel eines Quartals oder eines Geschäftsjahres) zusammenzufassen, um Geschäftsberichte oder dergleichen zu verfassen.

Auch das Stern-Schema der Hoteldatenbank erhält eine Dimensionstabelle für die Zeit. Als Attribute dienen das genaue Datum sowie der Wochentag und das Quartal. Das aus dieser Vorgehensweise resultierende Stern-Schema ist in Abbildung 2.6 zu sehen. Es handelt sich um ein Viel-Stern-Schema, bei dem die Primär- und die Fremdschlüssel der Faktentabellen nicht übereinstimmen.

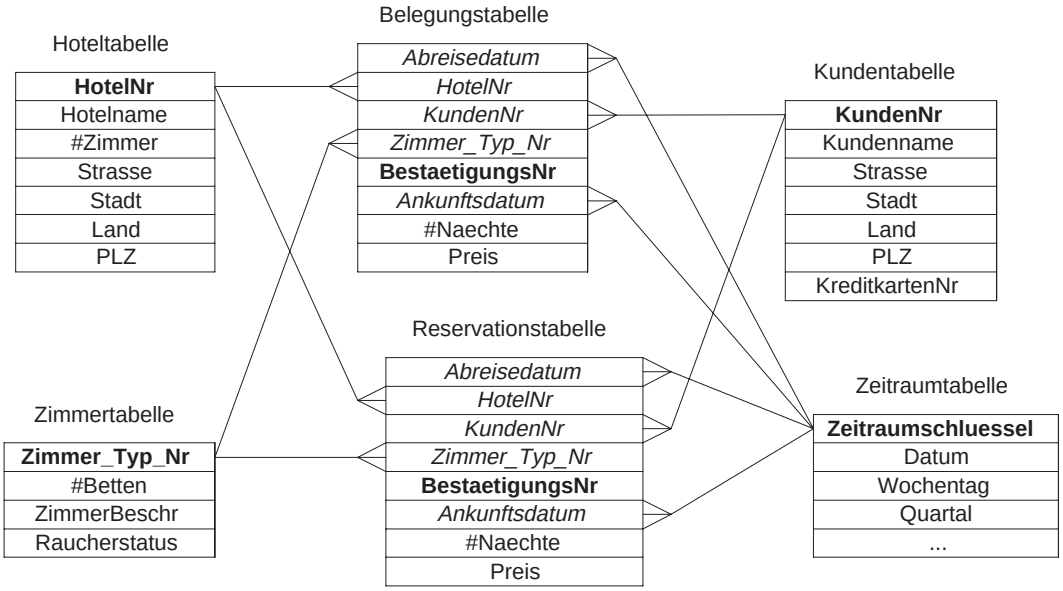


Abbildung 2.6: Reservationssystem einer Hotelkette als Beispiel

2.6 Zusammenfassung

Aufgrund der vielen Vorteile des Stern-Schemas hinsichtlich der Leistungsfähigkeit bei der Beantwortung von Abfragen, der Flexibilität bezüglich Erweiterungen oder Veränderungen und des leicht verständlichen Aufbaus, ist es hervorragend geeignet, um damit Entscheidungsunterstützungs-Systeme zu implementieren. Um die Möglichkeiten des Stern-Schemas allerdings voll nutzen zu können, muß der Entwurf sehr sorgfältig vorgenommen werden. Es ist wichtig, daß das Schema voll an die zu analysierenden Geschäftsabläufe angepaßt ist und alle wichtigen Fakten und Dimensionen beziehungsweise Attribute enthält.

Als Literatur zur Erstellung dieser Ausarbeitung dienten [18], [19], [20] und [21].

Kapitel 3

Mediatoren

Ansgar Zwick

Kurzfassung *Ein Instrument zur Integration von heterogenen Informationsquellen ist die Mediator-Architektur. In dieser Ausarbeitung soll das Konzept von Mediatoren vorgestellt werden. Mediatoren liefern Daten zu einer bestimmten Anfrage. Den Daten muß dabei ein bestimmtes Modell und den Anfragen eine Anfragesprache zugrunde liegen. Dazu werden im folgenden ein Datenmodell (OEM) und zwei verschiedene Anfragesprachen (MSL und LOREL) vorgestellt. MSL und LOREL sind die Anfragesprachen der Mediatorsysteme TSIMMIS und MedMaker. MSL ist aber nicht nur Anfragesprache, sondern damit werden Mediatoren auch spezifiziert. Welche Probleme dabei auftauchen und wie diese durch MSL unter Verwendung des Datenmodells OEM gelöst werden, wird dabei ebenso zur Sprache kommen, wie eine kurze Darstellung der Funktionsweise eines Mediators, der mit MSL spezifiziert wurde. Im letzten Teil findet sich eine Aufwandsabschätzung für die Lösung des Problems, ob eine Anfrage beantwortbar ist. Es geht dabei nicht darum, ob zu einer bestimmten Anfrage Daten lieferbar sind, sondern ob zu einer Anfage, die keiner Sicht entspricht, eine Kombination von mehreren Sichten angegeben werden kann, die zu der ursprünglichen Anfrage expandiert werden können, und somit die Anfrage realisieren. Es soll dabei vor allem die Vermittlung der Problemthematik im Vordergrund stehen, bevor eine kurze Aufwandsanalyse dazu erfolgt. Wir werden dabei feststellen, daß der Aufwand für die Lösung des Problems polynomial für den Fall ist, daß keine arithmetischen Vergleiche stattfinden, und exponentiell für den Fall mit arithmetischen Vergleichen.*

3.1 Mediator-Architektur

Es existieren mehrere Ansätze zur Integration von heterogenen Informationen. Einer davon ist die Mediator-Architektur, die wir am Beispiel einer Universität näher erläutern wollen, wo Informationen auf viele verschiedene Arten gespeichert sein können:

- Manche der Datenbanken sind relational, andere nicht. Manche Informationen wiederum stecken vielleicht nicht einmal in Datenbanken, sondern in einer Tabellenkalkulation oder in einer Text-Datei.

- Für die gleiche Information können verschiedene Typen verwendet worden sein. Z.B. kann eine Matrikelnummer als Integer oder als String gespeichert sein.
- Die verwendeten Einheiten können sich unterscheiden: Ein Lohn kann z.B. auf Stunden- oder auf Monatsbasis, d.h. als Stunden- oder als Monatslohn vorliegen.
- Die Sichtweisen können differieren: Z.B. taucht ein Hiwi in einer Lohnliste als Lohnempfänger auf, während er im Studentensekretariat als Student geführt wird.

Ein Weg nun, um solch unterschiedliche Informationsquellen zu integrieren, führt über Mediatoren, mit denen diverse Anfragen z.B. über Studenten an einer Uni bewältigt werden können. Ein Mediator kann dabei seine Informationen aus vorliegenden Quellen oder von anderen Mediatoren beziehen. Ein sehr einfaches Netzwerk von Mediatoren sehen wir in Abb. 3.1. (Anmerkung: Werden die Informationen aus den Quellen geholt, dann wird ein sog. Translator (oder auch *Wrapper*) dazwischengeschaltet. (Nicht abgebildet))

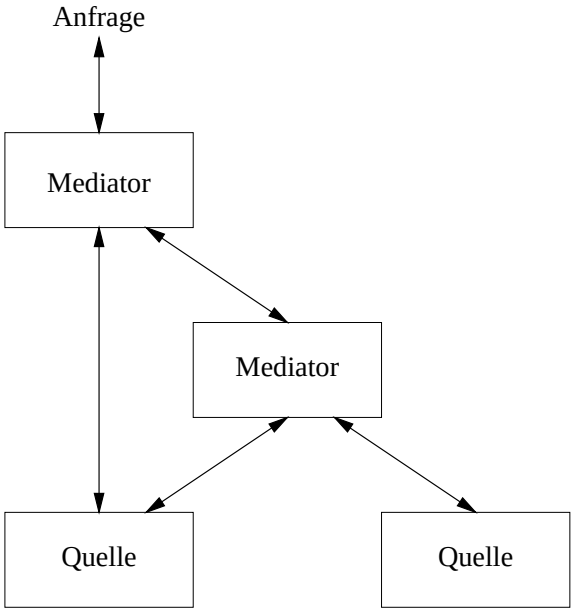


Abbildung 3.1: Ein Netzwerk von Mediatoren und Informationsquellen

3.1.1 Anforderungen an ein Mediator-System

Aus Abb. 3.1 lassen sich mehrere Anforderungen ableiten, um ein System von Mediatoren einfach anlegen und handhaben zu können:

1. Es muß ein gemeinsames Datenmodell geben, das flexibler ist als diejenigen, die in herkömmlichen Datenbanken eingesetzt werden. Dieses Datenmodell sollte folgende Punkte unterstützen:
   (a) Eine große Vielfalt an Strukturen, inklusiv verzweigter Strukturen, wie sie in einem Typ-System üblicher moderner Programmiersprachen eingesetzt werden. (z.B. record, array, ...)

(b) Eine elegante Behandlung fehlender oder verwandter Informationen.

(c) Meta-Informationen, d.h. Informationen über die Strukturen selber und die Bedeutung der gespeicherten Daten.

2. Es muß eine gemeinsame Anfrage-Sprache existieren, die es erlaubt, daß

   (a) neue *Mediatoren* hinzugefügt werden können, die die vorhandenen um zusätzliche Funktionalität ergänzen.

   (b) neue *Quellen* hinzugefügt werden können, die den existierenden Mediatoren als zusätzliche Informationsquellen dienen können.

3. Es müssen Werkzeuge existieren, um das Erstellen von Mediatoren bzw. Mediator-Systemen zu vereinfachen.

Im folgenden werden wir zwei Systeme vorstellen, die diesen Anforderungen genügen. Beide verwenden dabei als Datenmodell das OEM, das in Kap. 3.2 erläutert wird.

3.2 Object-Exchange Model (OEM)

Wie oben geschildert, ist ein flexibles Datenmodell eine der Voraussetzungen für eine Mediator-Architektur. Das im folgenden vorgestellte OEM (Object-Exchange Model) erfüllt alle in Kap. 3.1 gestellten Anforderungen: Zum ersten sind die Daten in OEM selbstbeschreibend. D.h., daß die Bedeutung der Daten den Daten selbst zugefügt werden. Zum zweiten werden durch die Flexibilität von OEM eine Vielzahl von Strukturen unterstützt. OEM kann als objekt-orientiert angesehen werden, in dem Sinne, daß das fundamentale Konzept von OEM auf Objekten aufbaut. Das Typ-System von OEM ist sehr elementar. Es sind nur atomare Typen wie `string, integer, ...` oder der Typ `set` erlaubt. Mit diesem System können aber alle herkömmlichen Strukturen wie `array, record, ...` simuliert werden. Drittens folgt OEM keinem starren Schema, so daß fehlende oder verwandte Informationen keine Schwierigkeiten darstellen.

3.2.1 OEM Objekte

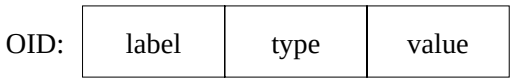


Abbildung 3.2: Ein OEM Objekt

Wie aus Abb. 3.2 ersichtlich, besteht ein OEM Objekt aus vier Komponenten:

1. *Object-ID.* Eine OID kann ein Ausdruck sein, der die Herkunft eines Objekts angibt. Oder es kann eine Referenz auf ein Objekt im Arbeitsspeicher sein, die benutzt wird, um eine Anfrage zu beantworten.

2. *Label.* Es beschreibt die Bedeutung eines Objekts. Man könnte es als Klasse ansehen, obwohl es zu einem bestimmten Label keine Aussage über Unterobjekte gibt. Es ist insbesondere nicht festgelegt, wieviele und was für Unterobjekte ein Objekt mit einem bestimmten Label haben muß. Ein Label sollte einen für einen Menschen verständlichen Ausdruck enthalten, der es einem Benutzer erleichtert, bestimmte Objekte zu finden. Insofern enthält ein Label die gesamte Meta-Information über ein Objekt, was beim OEM selbstbeschreibend genannt wird.

3. *Type.* Der Typ von *value* ist entweder atomar (`string, integer, ...`) oder `set`.

4. *Value* ist entweder ein atomarer Wert oder eine Menge von Referenzen auf Objekte.

Wie Objekte zu einer Struktur zusammengefügt werden können, kann man in Abb. 3.3 erkennen. In den folgenden Abschnitten werden wir zwei Mediator-Systeme vorstellen, die beide OEM als Datenmodell benützen.

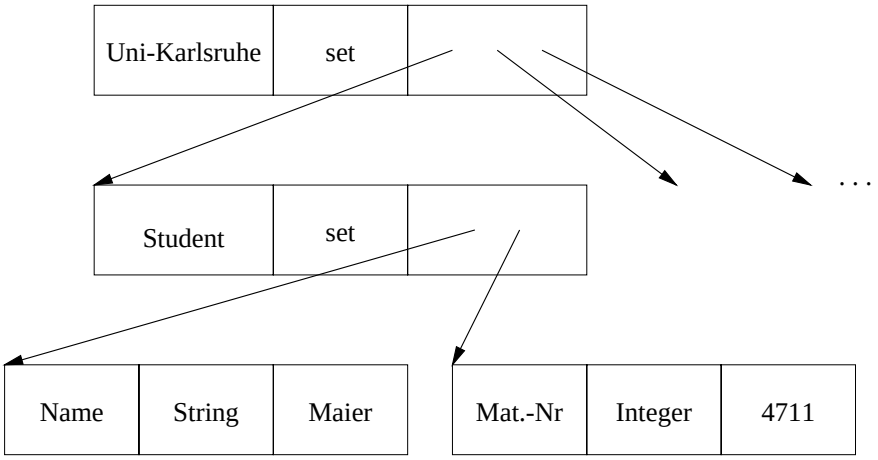


Abbildung 3.3: Eine Struktur von OEM Objekten

3.3 TSIMMIS

Der 'Stanford-IBM Manager of Multiple Information Sources' kurz TSIMMIS ist eines der Mediator-Systeme, die zur Zeit entwickelt werden [22]. TSIMMIS benützt als Datenmodell OEM und als Anfragesprache die Sprache LOREL. Für die Erzeugung von Mediatoren und Translatoren sollen Werkzeuge zur Verfügung gestellt werden, die aber momentan noch in Entwicklung sind.

3.3.1 LOREL

Die Anfragesprache LOREL könnte salopp als 'SQL mit Punkten' beschrieben werden. Sie ist vor allem entwickelt worden, um ähnliche Anfragen wie in SQL zu erlauben. Es soll hier keine genaue Beschreibung dieser Sprache erfolgen, sondern mit einem Beispiel wollen wir versuchen, ein grobes Bild dieser Sprache zu vermitteln. So könnte eine Anfrage

zu einer Matrikelnummer von Student Maier, die an ein System wie in Abb. 3.3 gestellt wird, folgendermaßen lauten:

```
select Uni-Karlsruhe.Student.Mat-Nr
where Uni-Karlsruhe.Student.Name = 'Maier'
```

Ein Ausdruck mit Punkten (Hier z.B. `Uni-Karlsruhe.Student.Mat-Nr`) stellt dabei einen Pfad von *Labels* in der Struktur der OEM-Objekte dar. Man kann sich die Arbeitsweise dabei folgendermaßen vorstellen: Mit obiger Anfrage würde nach einem[1] Objekt mit dem Label `Student` gesucht, das folgende Bedingungen erfüllt: Es muß ein Unterobjekt eines Objekts mit dem Label `Uni-Karlsruhe` sein. Und es muß ein Unterobjekt mit dem Label `Name` und dem Wert 'Maier' besitzen. Wird ein solches Objekt gefunden, dann wird überprüft, ob es ein weiteres Unterobjekt diesmal mit dem Label `Mat-Nr` besitzt, und wenn ja, wird dessen Wert als Antwort auf die Anfrage zurückgeliefert. Obige Anfrage bezieht sich auf das Wurzelobjekt mit dem Label `Uni-Karlsruhe`. Falls mehrere Wurzelobjekte verfügbar wären, dann könnte man die gewünschte(n) Wurzel mit einer from-Klausel spezifizieren. Diese Klausel würde wie in SQL zwischen der select- und der where-Klausel stehen:

```
select Uni-Karlsruhe.Student.Mat-Nr
from Uni-Karlsruhe
where Uni-Karlsruhe.Student.Name = 'Maier'
```

LOREL wurde vor allem entwickelt, um SQL-Benutzern einfache Anfragen an eine Mediatorarchitektur zu ermöglichen. Eine andere flexiblere Anfragesprache, die auf Prädikatenlogik basiert und eher im Hinblick auf die Spezifikation von Mediatoren entwickelt wurde, werden wir in Kap. 3.4.1 kennenlernen.

3.4 MedMaker

MedMaker ist wie auch TSIMMIS ein System zur Integration von heterogenen Informationen, das auf Mediatoren aufbaut [23]. Ebenso wie bei TSIMMIS wird auch hier als Datenmodell OEM eingesetzt. Für Anfragen allerdings, wie auch zur Beschreibung von Mediatoren, wird die deklarative Sprache MSL eingesetzt, die ein hohes Abstraktionsniveau zur Verfügung stellt und die im folgenden Abschnitt näher erläutert werden soll.

3.4.1 MSL

Das Ziel ist es, zu einer gegebenen Anzahl von Translatoren (auch *wrappers*), die OEM Objekte exportieren, Mediatoren bauen zu können, um verschiedene Informationen zu integrieren und zu ordnen. Speziell wird im folgenden von Mediatoren die Rede sein, die integrierte OEM-*Sichten* der zugrundeliegenden Informationen zur Verfügung stellen. Es

[1] Der Einfachheit halber wollen wir hier annehmen, daß maximal ein passendes Objekt existiert.

soll dabei der Aufwand einer harten Codierung von Mediatoren überflüssig gemacht werden, indem eine deklarative Hochsprache zur Verfügung gestellt wird, mit der Mediatoren beschrieben werden können. Wenn der Mediator zur Laufzeit dann eine Anfrage erhält, wird die erforderliche Information von einem 'Mediator Specification Interpreter' (MSI) entsprechend der Spezifikation gesammelt und integriert. Dies läuft dann ähnlich ab, wie das Beantworten einer Anfrage an eine relationale Datenbank. Jedoch führten die speziellen Anforderungen zu einer Anzahl von Konzepten, die in bisherigen Sprachen zur Beschreibung von Sichten nicht vorhanden sind, wie z.B.:

- MSL Mediator Spezifikationen sind ohne Änderung einsatzfähig, selbst wenn die zugrunde liegenden Informationsquellen (innerhalb eines bestimmten Rahmens) weiterentwickelt werden.
- MSL kann mit Unregelmäßigkeiten der Quellen umgehen, ohne zu fehlerhaften oder unerwarteten Resultaten zu führen.
- MSL kann Informationsquellen integrieren, deren vollständige Strukturen nicht bekannt sind.
- MSL kann sowohl Werte (*values*), als auch die Semantik wiedergebende Labels eines Objekts verändern und dadurch semantische Diskrepanz verhindern.

Im nächsten Abschnitt soll anhand eines Beispiels die Funktionsweise von MSL verdeutlicht werden.

3.4.2 Beispiel für eine Mediator–Beschreibung mit MSL

In einem etwas größeren Beispiel sollen die Funktionalität und insbesondere die Vorteile von MSL veranschaulicht werden. Wir gehen wieder von einem System aus, das Informationen über die Personen einer Universität liefern soll. Es stehen dafür zwei Quellen zur Verfügung. Die erste besteht aus einer relationalen Datenbank der Fakultät Informatik mit zwei Tabellen folgenden Aufbaus:

```
angestellter(vorname, zuname, titel, institut)
student(vorname, zuname, mat-nr)
```

Ein Translator `INFO` exportiere daraus Informationen als OEM Objekte. Eine Struktur von OEM Objekten, die `INFO` liefert, könnte folgendermaßen aussehen (jede Zeile repräsentiere dabei ein OEM Objekt, wobei eine zu einem Objekt gehörende Objekt-ID durch ein & eingeleitet werde):

```
<&a1, angestellter, set, {&v1, &z1, &t1, &i1}>
   <&v1, vorname, string, 'Max'>
   <&z1, zuname, string, 'Maier'>
   <&t1, titel, string, 'Professor'>
   <&i1, institut, string, 'Datenbanken'>
```

```
<&a2, angestellter, set, {&v2, &z2, &t2}>
   <&v2, vorname, string, 'Harry'>
   <&z2, zuname, string, 'Hirsch'>
   <&t2, titel, string, 'Doktorand'>
:
:
<&s3, student, set, {&v3, &z3, &m3}>
   <&v3, vorname, string, 'Otto'>
   <&z3, zuname, string, 'Schmidt'>
   <&m3, mat-nr, integer, 1234>
:
:
```

Weiterhin gebe es eine allgemeine Informationsquelle der Universität mit dem Translator `whois`, der OEM-Objekte mit folgender Struktur zur Verfügung stelle (nähere Angaben über die Informationsquelle seien hier nicht vorhanden):

```
<&p1, person, set, {&nm1, &f1, &s1, elm1}>
   <&nm1, name, string, 'Max Maier'>
   <&f1, fakultaet, string, 'Info'>
   <&s1, status, string, 'angestellter'>
   <&elm1, e-mail, string, 'maier@info'>
<&p2, person, set, {&nm2, &f2, &s2, &m2}>
   <&nm2, name, string, 'Uli Ungut'>
   <&f2, fakultaet, string, 'Info'>
   <&s2, status, string, 'Student'>
   <&m2, mat-nr, integer, '4711'>
:
:
```

Man beachte, daß bei Objekten aus dieser Quelle Unregelmäßigkeiten auftreten können. Z.B. enthält das Objekt `&p1` ein Unterobjekt mit einer e-mail-Adresse, während dies bei dem Objekt `&p2` nicht der Fall ist.

Zur Integration der Informationen aus beiden Quellen existiere ein Mediator `med`, der auf die beiden Translatoren `INFO` und `whois` zugreife und eine Menge von Objekten mit dem Label 'info-person' zurückliefere. Dabei soll jedes 'info-person'-Objekt eine Person repräsentieren, die in beiden Quellen vorkommt, und es soll die zugehörigen Informationen aus beiden Quellen vereinigen. Da zum Beispiel die Person 'Max Maier' in beiden Quellen existiert, würde der Mediator `med` die zugehörigen Informationen kombinieren und folgendes Objekt liefern:

```
<&ip, info-Person, {&ipn,&ips,&ipt,&ipi,&ipe}>
   <&ipn, name, string, 'Max Maier'>
   <&ips, status, string, 'Angestellter'>
   <&ipt, titel, string, 'Professor'>
   <&ipi, institut, string, 'Datenbanken'>
   <&ipe, e-mail, string, 'Maier@info'>
```

Mögliche Probleme

Anhand des vorigen Beispiels werden eine Anzahl von Problemen deutlich:

- *Verschiedene Domänen der Schemata.* In `whois` besteht der Name aus einem langen String, der sowohl den Vor- als auch den Nachnamen enthält, während `INFO` für einen Namen zwei Objekte, nämlich eins für den Vornamen und eins für den Nachnamen, liefert.

- *Schematische Diskrepanz.* Die Daten der einen Datenquelle sind die Metadaten der anderen Quelle: Im Beispiel taucht der Status einer Person (`angestellter` oder `student`) bei `whois` als *Wert* auf, weil er ein Teil der Daten war, während er bei `INFO` als *Label* (d.h. in den Metadaten) auftaucht, weil er dort Teil des relationalen Schemas der Datenbank war.

- *Schema Evolution.* Das Format und der Inhalt von Informationsquellen können sich nach einer bestimmten Zeit ändern bzw. weiterentwickelt werden; dies geschieht dabei oft, ohne daß der Entwickler eines Mediators davon erfährt. Z.B. könnte bei einer der beiden obigen Quellen oder bei beiden das Geburtsdatum einer Person hinzugefügt werden oder die e-mail-Adresse könnte weggelassen werden. Es ist dann wünschenswert, daß die Spezifikation eines Mediators auf möglichst viele solcher Änderungen unempfindlich reagiert. Wenn z.B. das Geburtsdatum hinzugefügt oder weggelassen wird, dann sollte es auch automatisch in der Sicht, die der Mediator `med` liefert, hinzugefügt oder weggelassen werden, ohne daß man die Spezifikation des Mediators ändern muß.

- *Unregelmäßigkeiten der Struktur.* Die Quelle `whois` liefert kein regelmäßiges Schema, was man daran sieht, daß bei einer Person als Unterobjekt eine e-mail-Adresse vorkommt und bei einer anderen nicht. Dies sollte jedoch zu keinem unkontrollierten Verhalten des Mediators führen.

Die Spezifikation des Mediators `med`.

MSL kann alle angegebenen Probleme lösen. Bevor wir aber direkt in die genaue Beschreibung von MSL einsteigen, wollen wir einfach einmal die Spezifikation des Mediators `med` in MSL ansehen. Dabei sollte es in etwa deutlich werden, wie MSL durch das hohe Abstraktionsniveau und die Flexibilität von OEM die obigen Probleme löst. Die folgende MSL Spezifikation MS1 definiert den oben beschriebenen Mediator `med`. Die Spezifikation wird dabei in den danach folgenden Absätzen genauer erklärt.

MS1:

```
Rules:
  <info-person {<name N> <status S> Rest1 Rest2}>
   :- <person {<name N> <fakultaet 'INFO'> <status S> | Rest1} > @whois
      AND  decomp(N, VN, ZN)
      AND  <S {<vorname VN> <zuname ZN> | Rest2} > @INFO
```

```
External:
  decomp(string, string, string) (bound, free, free)
      impl by name_to_vnzn
  decomp(string, string, string) (free, bound, bound)
      impl by vnzn_to_name
```

Wie wir sehen besteht eine Spezifikation eines Mediators aus Prolog-ähnlichen Regeln, welche die Sicht definieren, die ein Mediator zurückgeben soll. Jede Regel (im obigen Beispiel gibt es nur eine Regel) besteht aus einem Kopf und einem Rest, die durch das Symbol :- getrennt werden. Der Kopf beschreibt dabei die Objekte der Sicht, die ein Mediator liefert. Im Rest der Regel findet man die Beschreibung der Objekte, die aus den Quellen extrahiert werden sollen.

In den Informationsquellen (in unserem Beispiel `INFO` und `whois`) wird nach Objekten gesucht, die zum Rest der Regel passen, und bei passenden Objekten werden die entsprechenden Variablen gebunden (Variablen beginnen immer mit einem Großbuchstaben (z.B. N, S, VN)). Mit diesen Bindungen werden dann die Objekte, die zurückgeliefert werden sollen, entsprechend dem Kopf der Regel gebildet.

Um die virtuellen Ergebnisobjekte der Sicht eines Mediators zu erzeugen, kann man sich Mustervergleiche vorstellen. Die Spezifikation beruht auf Mustern der Form `<object-id, label, type, value>`, was dem Aufbau von OEM-Objekten entspricht. Dies ist naheliegend, da ja OEM als Datenmodell eingesetzt wird. An jede Position kann entweder eine Konstante oder eine Variable gesetzt werden. Unwichtige Positionen können auch weggelassen werden. Wenn eine Position fehlt, wird dabei angenommen daß es *type* ist, wenn es zwei sind, müssen es *type* und *object-id* sein. Dann sind nur noch *Label* und *value* ausschlaggebend.

Wird im Rest der Regel keine object-id angegeben, dann bedeutet das, daß sie für die Suche nach Objekten unwichtig ist. Wird dagegen im Kopf der Regel keine angegeben, dann heißt dies, daß es uninteressant ist, welche object-ids der Mediator für die generierten Objekte benützt.

Im folgenden soll anhand von obigem Beispiel dargestellt werden, wie ein Mustervergleich eines Mediators arbeitet. Wenn z.B. ein *label* (oder *value*)-Feld im Rest einer Regel eine Konstante enthält, dann werden aus den Quellen nur Objekte herangezogen, die an dieser Position mit der Konstante übereinstimmen. In unserem Beispiel werden durch den Ausdruck `<fakultaet 'INFO'>` nur Objekte akzeptiert, die das *Label* `fakultaet` und den Wert (*value*) 'INFO' besitzen. Wenn ein Feld andererseits eine Variable enthält, werden alle OEM Objekte herangezogen, aber gleichzeitig wird die Variable jeweils an den Wert des speziellen Objekts gebunden. Z.B. würde das Muster `<name N>` folgende Objekte akzeptieren: `<&1, name, string, 'Tom'>` oder `<&2, name, string, 'Max'>`, wobei die Variable N jeweils an 'Tom' oder 'Max' gebunden würde.

Schauen wir uns einmal das Muster

```
<person {<name N> <fakultaet 'INFO'> <status S> | Rest1} > @whois}
```

genauer an. Hiermit wird nach Objekten gesucht, die das label 'person' besitzen. Ferner müssen diese Objekte folgende Unterobjekte haben: Ein Unterobjekt mit dem label 'name', an dessen Wert die Variable N gebunden wird, ein Unterobjekt mit dem label 'fakultaet' und dem Wert 'INFO' und ein Unterobjekt mit dem label 'status', an dessen Wert die Variable S gebunden wird. Die Variable Rest1 wird dabei an die übrigen Unterobjekte gebunden, falls vorhanden. In unserem Beispiel würde damit u.a. das Objekt `&p1` akzeptiert werden. Die Variable N würde dann an 'Max Maier' und die Variable S an 'Angestellter' gebunden werden, und Rest1 würde die restlichen Unterobjekte übernehmen, also an {`<&elm1, e-mail, string, 'Maier@info'>`} gebunden werden. Wir werden diese Bindungen im folgenden $b_{w,1}$ nennen. Es können auch andere Objekte akzeptiert werden, wie z.B. in unserem Fall das Objekt `&a2`. Selbstverständlich sind dann die Variablen an entsprechend andere Werte gebunden.

Auf dieselbe Weise wird nun die Quelle `INFO` aufgrund des Musters

```
<S {<vorname VN> <zuname ZN> | Rest2} > @INFO
```

nach passenden Objekten durchsucht. Dabei werden bei Erfolg die Variablen S, VN, ZN, Rest2 an entsprechende Werte gebunden. Es würde z.B. das Objekt `&a1` akzeptiert mit den Bindungen von S an 'Angestellter', VN an 'Max', ZN an 'Maier', und Rest2 an {`<&i1, institut, string, 'Datenbanken'>`} . Diese Bindungen wollen wir $b_{c,1}$ nennen.

Im nächsten Schritt müssen nun die verschiedenen Bindungen im Rest der Regel des Mediators miteinander verglichen werden. Wir nennen eine Bindung $b_{w,i}$ aus `whois` passend zu einer Bindung $b_{c,i}$ aus `INFO`, wenn beide Bindungen für gemeinsame Variablen (in diesem Falle R) übereinstimmen. Die Bindungen $b_{w,1}$ und $b_{c,i}$ würden also 'zueinander passen', da bei beiden die Variable S an 'Angestellter' gebunden ist. (Für ein gültiges Ergebnisobjekt müssen allerdings natürlich noch die Bindungen von `decomp` berücksichtigt werden.)

Externe Prädikate

Der Vergleich zwischen einem Namen und einem Vor- und einem Zunamen wird in dem Prädikat `decomp(N, VN, ZN)` vorgenommen. Im Prinzip können wir uns `decomp` als ein Prädikat vorstellen, das WAHR zurückgibt, wenn N einer Zusammensetzung aus einem Vornamen VN und einem Zunamen ZN entspricht. In der Praxis ist `decomp` jedoch durch zwei Funktionen `name-to-lnfn` und `lnfn-to-name` realisiert. Diese Funktionen können in einer beliebigen Programmiersprache implementiert sein. Der Sinn von zwei Funktionen liegt in unserem Fall in einer optimierten Ausführungszeit. Wird nämlich `decomp` mit einem Gesamtnamen aufgerufen, dann gibt die Funktion `name-to-lnfn` einen Vor- und einen Zunamen zurück. Und umgekehrt kann `lnfn-to-name` einen Gesamtnamen berechnen. Würde `decomp` als Prädikat mit drei gebundenen Variablen realisiert werden, dann müßten um z.B. einen Gesamtnamen zu einem Vor- und Zunamen zu finden, alle Objekte aus `INFO` nach einem passenden Gesamtnamen durchsucht werden. Durch die Funktion `lnfn-to-name` wird jedoch zu einem Vor - und Zunamen sofort ein Gesamtname konstruiert, ohne daß man suchen muß.

3.4.3 Anfragen in MSL

Wir haben nun gesehen, wie mit MSL ein Mediator spezifiziert werden kann. Doch MSL ist zugleich auch eine einfache und mächtige Anfragesprache. Eine Anfrage in MSL könnte z.B. lauten:

```
MM :- MM:<info-person {<name 'Max Maier'>} > @med
```

Wir erkennen, daß der Ausdruck `info-person {<name 'Max Maier'>}` dieselbe Syntax und Semantik verwendet, die auch für die Definition von Sichten in den Regeln eingesetzt wird. Neu ist allerdings die *Objekt-Variable* MM im Rest der Regel. Diese Variable wird an ein ganzes Objekt gebunden. Der Operator : schränkt aber dabei die Objekte auf 'info-person'-Objekte ein, die ein Unterobjekt 'name' mit dem Wert `Max Maier` besitzen. Der Kopf bestimmt, daß jedes Objekt, das MM bindet, dem Ergebnis hinzugefügt wird. Im Gegensatz zur Mediator-Spezifikation, wo MSL auch eingesetzt wird, werden hier aber keine virtuellen sondern 'reale' Objekte erzeugt und dem Benutzer zur Verfügung gestellt.

3.5 Beantwortbarkeit von Anfragen

In vorigen Abschnitt haben wir an einem Beispiel gesehen, wie in MSL Mediatoren spezifiziert werden. Nun wollen wir wissen, wie hoch grundsätzlich der Aufwand in logischen Sprachen ist (wozu auch MSL gehört) um herauszufinden, ob eine Anfrage, die mit Mustervergleichen gelöst wird, beantwortbar ist. Beantwortbar heißt hier nicht, daß entsprechende Daten in den Quellen vorhanden sind und gefunden werden, sondern daß die Anfrage expandierbar ist, bzw. eine gültige Lösung für die Anfrage existiert. Entspricht die Anfrage einer Sicht, dann ist sie trivialerweise beantwortbar. Entspricht die Anfrage keiner Sicht, dann ist es in logik-basierten Sprachen unter bestimmten Umständen trotzdem möglich, die Anfrage zu beantworten. Ob dies möglich ist, kann allerdings nicht sofort angegeben werden. Doch dazu mehr in den nächsten Abschnitten. Wir werden dabei im folgenden annehmen, daß Prädikate über die zu integrierenden Informationen definierbar sind. (Anmerkung: Sowohl MSL als auch LOREL können zum großen Teil als Prädikatenlogik interpretiert werden.)

3.5.1 Beispiel

Eine Informationsquelle sei in irgendeiner Form als Genealogie gegeben (d.h. es gibt 'Eltern' und 'Kinder', und wenn ein Element a ein Kind eines Elements b ist, so gilt, daß b sich unter den Eltern von a befinden und umgekehrt), und es gebe nur zwei Anfragen, die an die Informationsquelle gestellt werden können: (Es soll uns hier nicht weiter interessieren, warum dies so ist, aber es kann dafür durchaus Gründe geben. s. [24])

1. Irgendein Element a sei gegeben, finde die Eltern von a.
2. Finde alle Elemente, die Eltern (in der Struktur der Informationsquelle) haben.

Und obwohl nur diese zwei Anfragemuster existieren, ist es doch möglich, die Anfrage A 'Finde die Großeltern eines Elements a' zu beantworten, nämlich durch:

i. Finde die Menge P aller Eltern von a mittels Anfrage (1).
ii. Finde für jedes Element p in der Menge P die Eltern mittels Anfrage (1).
iii. Vereinige die Mengen aus (ii) und gib die Vereinigung als Antwort auf die Anfrage A zurück.

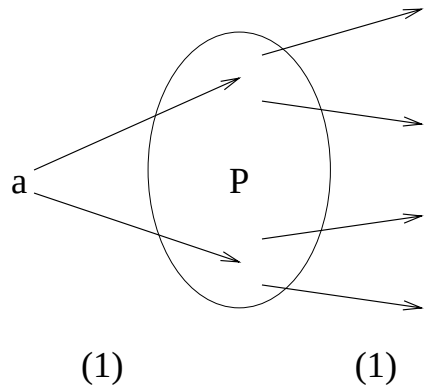


Abbildung 3.4: Die Suche nach den Großeltern

Nun wollen wir die Anfrage B 'Finde die Enkelkinder eines Elements g' beantworten. Wir benötigen dazu sowohl Anfrage (1) als auch (2):

i. Finde alle Elemente mittels (2).
ii. Finde die Eltern aller Elemente aus (i) mittels (1).
iii. Finde die Eltern aller Elemente aus (ii) mittels (1).
iv. Ermittle alle Elemente, von denen in (iii) herausgefunden wurde, daß sich das Element g unter den Großeltern befindet.

Dies ist eine sehr teure Strategie, aber die beste unter den gegebenen Umständen, d.h. den gegebenen Anfragen (1) und (2).

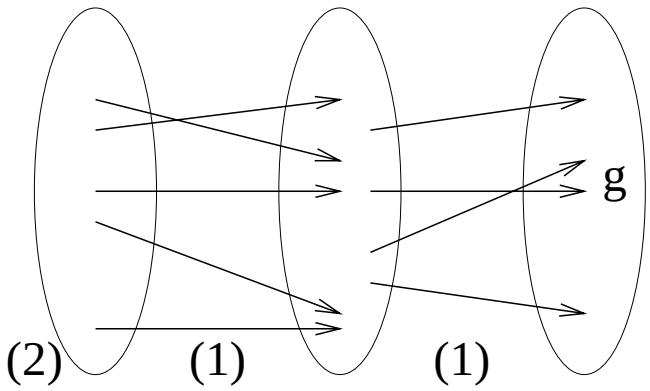


Abbildung 3.5: Die Suche nach Enkeln

3.5.2 Formales Modell

Wir wollen nun ein allgemeines formales Modell aufstellen, um Anfragen der Art wie im vorigen Beispiel handhaben zu können, und stellen an dieses Modell folgende Forderungen:

1. Es gebe bestimmte Prädikate, die den Anfragen zur Verfügung stehen. Diese Prädikate sollen aus dem Modell stammen, das den zu integrierenden Informationen zugrunde liegt. In unserem Beispiel wäre *parent(C,P)* ein solches Prädikat.

2. Wir nehmen an, daß es gewisse Anfragemuster (*query templates*) gibt, auch Sichten (*views*) genannt. Eine Sicht (genauer die Regel, die die Sicht realisiert) besteht aus einem Kopf und einem Rest. (Man vergleiche dies mit der Sprache MSL in Kap. 3.4.1, wo Sichten auf dieselbe Weise definiert werden.) Ein Kopf hat folgende Komponenten:

   (a) Ein Prädikat, das die Sicht beschreibt,

   (b) Argumente des Prädikats und

   (c) ein Bindungsmuster (*binding pattern* oder *adornment*), das angibt, welche Argumente des Prädikats gebunden sein sollen (und damit als Parameter für die Anfrage dienen) und welche Argumente frei sind (und damit die Antwort der Anfrage enthalten).

Im allgemeinen Fall kann der Rest einer Sicht irgendein Programm sein, das eine Lösung der Anfrage mittels den zu Verfügung stehenden Prädikaten ermittelt.

Aber zurück zu unserem Beispiel. Zu den zwei Anfragen, die unser genealogisches System zur Verfügung stellt, formulieren wir zwei Sichten (wie erwähnt werden die Parameter einer Anfrage zu gebunden Variablen einer Sicht, und die freien Variablen sind Platzhalter für die Antwort der Anfrage):

- Zur Anfrage (1) die Sicht: v_1^{bf}(C,P) :- `parent(C,P)`
- und zu (2) die Sicht: v_2^{f}(C) :- `parent(C,P)`

Wobei ein hochgestelltes bf bedeutet, daß die erste Variable gebunden (*bound*) und die zweite frei (*free*) ist. Beispielsweise würde ffb für 'frei,frei,gebunden' stehen. Die zwei obigen Sichten könnten z.B. durch einen Translator erbracht werden.

Anfragen

Eine Anfrage wird genau wie eine Sicht formuliert. D.h. mit einem Kopf, einem Bindungsmuster für den Kopf und mit einem Rest über die Prädikate, die die Informationsquelle beschreiben. Unser Beispiel zur Ermittlung der Großeltern eines Elements würde man folgendermaßen notieren:

$grandparents^{bf}$(C,G) :- `parent(C,P) & parent(P,G)`

Das Argument C ist also gebunden und enthält das Element, dessen Großeltern wir suchen, während G die zu findenden Großeltern repräsentiert, die nach dem Programm im Rest zu ermitteln sind. Falls wir das Bindungsmuster umdrehen, wird die Variable G gebunden und die Variable C frei, was die Anfrage nach den Enkeln eines Elements ausdrückt:

$grandchildren^{fb}$(C,G) :- `parent(C,P) & parent(P,G)`

Gültige Lösungen

Eine Anfrage wird beantwortet durch ein Programm, das Sichten benützt um Informationen aus der externen Welt zu erhalten, wobei vorübergehend interne Daten produziert werden können, arithmetische Vergleiche eingesetzt und Prädikate verwendet werden können, die unabhängig von den Informationsquellen sind. Da also die Anfragen aus verknüpften *Prädikaten* bestehen, jedoch durch verknüpfte *Sichten* (die durch Prädikate definiert sind) beantwortet werden müssen, ist eine Anfrage nur beantwortbar, wenn es eine Kombination von Sichten gibt, deren Expansion äquivalent zu der ursprünglichen Anfrage ist. Diese Kombination von Sichten nennen wir 'Lösung'. Wir werden dabei im folgenden annehmen, daß die Lösungen der Anfragen *konjunktive* Ausdrücke sind in der Form von Chandra und Merlin [1977] (s. [25]), oder Programme, die *konjunktiv* verknüpfte Sichten benützen (man erinnere sich, daß jede Sicht in unserem Sinne ein Prädikat mit einem Bindungsmuster ist.) Für eine *gültige Lösung* in unserem Sinne werden folgende Forderungen erhoben:

1. Wenn in einem Unterausdruck U_i eine Variable in einem Prädikat laut Bindungsmuster *gebunden* sein muß, dann muß diese Variable entweder schon im Kopf der Regel als gebundene Variable vorkommen, oder sie muß in einem Prädikat in einem der ersten i-1 Unterausdrücke (gebunden oder frei) auftauchen.

2. Die Expansion der Lösung (in der jeder Unterausdruck durch den Rest einer Sicht ersetzt wird) muß zur gegebenen Anfrage äquivalent sein.

Wenn wir unser Beispiel betrachten mit der Sicht v_1^{bf} und der Anfrage $grandparents^{bf}$, dann ist eine gültige Lösung:

$grandpa\bar{r}ents^{bf}$(C,G) :- v_1^{bf}(C,P) & v_1^{bf}(P,G)

Diese Lösung erfüllt unsere Forderung (1) bezüglich der Bindungsmuster, da die im ersten Unterausdruck gebundene Variable C im Kopf gebunden wird, und da die im zweiten Unterausdruck gebundene Variable P im ersten Unterausdruck auftaucht (und damit gebunden wird).

Um Forderung (2) zu überprüfen müssen wir die Lösung expandieren, d.h. die Sichten durch ihre Definitionen (genauer den Rest der Definitionen) ersetzen.

Wenn wir also v_1^{bf} entsprechend durch den Rest der zugehörigen Definition ersetzen (wobei selbstverständlich die Variablen angepaßt werden müssen) gelangen wir zu folgender Expansion:

$grandparents^{bf}$(C,G) :- `parent(C,P) & parent(P,G)`

Wie wir sehen ist dieser Ausdruck identisch zu der entsprechenden ursprünglichen Anfrage und darum äquivalent zu jener Anfrage. Wir stellen fest, daß beide Forderungen erfüllt sind.

Nun wollen wir eine Lösung zu der Anfrage nach den Enkeln finden. Folgende Lösung ist *nicht* gültig:

$grandchildren^{fb}$(C,G) :- v_1^{bf}(C,P) & v_1^{bf}(P,G)

Sie ist ungültig, weil die im ersten Unterausdruck gebundene Variable C im Kopf nicht gebunden wird. Es hilft auch nichts, die Unterausdrücke v_1^{bf}(C,P) und v_1^{bf}(P,G) zu vertauschen, da dann die Variable P nicht im Kopf gebunden wird.

Wir können jedoch unter Verwendung der Sicht v_2^f eine gültige Lösung konstruieren:

$grandchildren^{fb}$(C,G) :- v_2^f(C) & v_1^{bf}(C,P) & v_1^{bf}(P,G)

In diesem Fall ist es nicht nötig, daß die im ersten Unterausdruck vorkommende Variable C im Kopf gebunden wird, da sie in v_2^f frei vorkommt. Andererseits bindet der erste Unterausdruck die Variable C, was für den zweiten Unterausdruck wichtig ist. Auch der dritte Unterausdruck stellt sich unserer Forderung (1) bezüglich der Bindungen nicht in den Weg. Es ist auch nicht relevant, daß die dort freie Variable G im Kopf gebunden wird. Wenn wir nun noch expandieren kommen wir zu folgendem Ergebnis:

$grandchildren^{fb}$(C,G) :- `parent(C,X) & parent(C,P) & parent(P,G)`

Man beachte, daß lokale Variablendefinitionen bei der Expansion bei Bedarf durch neue lokale Variablen ersetzt werden müssen, um Kollisionen zu vermeiden (z.B. ist die Variable X eine neue lokale Variable). Diese Expansion ist nicht identisch mit der ursprünglichen Anfrage, aber man erkennt, daß sie äquivalent sind. Der erste Unterausdruck unserer Lösung war nötig, um die Forderung (1) bezüglich der Bindungen zu erfüllen, aber wenn man die expandierte Lösung mit der ursprünglichen Anfrage vergleicht, sieht man, daß nach der Expansion der Ausdruck `parent(C,X)` redundant ist.

Arithmetische Vergleiche

Man kann sowohl Sichtdefinitionen als auch Lösungen um konjunktive Anfragen erweitern, die in Unterausdrücken den arithmetischen Wert zweier Variablen vergleichen. Die Forderungen zur Gültigkeit einer Lösung werden dabei nicht verändert. Allerdings kommt eine Forderung hinzu: Wenn in einem Unterausdruck ein Vergleich der Form `X < Y` vorkommt, dann muß gewährleistet sein, daß sowohl X als auch Y entweder im Kopf oder in einem der vorangegangenen Unterausdrücke gebunden werden.

3.5.3 Aufwandsabschätzung

Die fundamentale Frage lautet: Wenn es eine Menge von Sichten und eine Anfrage gibt, gibt es dann eine Lösung für die Anfrage unter Verwendung der Sichten ? Wenn die Sichten, die Anfrage und die Lösung konjunktive Ausdrücke sind ohne arithmetische Vergleiche, dann kann ein Entscheidungsalgorithmus angegeben werden und gezeigt werden, daß dieses Problem NP-vollständig ist.

Dieses Problem ist nicht trivial. Es kann einfach nachgewiesen werden, ob die Forderung (1) bezüglich der Bindungen erfüllt ist und ob zwei konjunktive Anfragen äquivalent sind (beide Probleme liegen in NP), aber man kann nicht direkt eine Obergrenze zum Finden einer Lösung zu einer bestimmten Anfrage angeben.

Lemma: Q sei eine konjunktive Anfrage (mit einem Bindungsmuster) mit n Unterausdrücken und m verschiedenen Variablen. Weiterhin gebe es eine Menge von Sichten, die aus konjunktiven Anfragen mit einem Bindungsmuster für den Kopf bestehen. Wenn es dann eine Lösung in unserem Sinne (d.h. aus konjunktiven Anfragen) gibt, dann gibt es eine Lösung mit höchstens m+n Unterausdrücken unter Verwendung von höchstens m verschiedenen Variablen. (Der ausführliche Beweis dazu ist in [24] zu finden.)

Mit diesem Lemma läßt sich nun folgendes Theorem aufstellen:

Theorem: Es seien Sichten gegeben, die durch konjunktive Anfragen und einem Bindungsmuster definiert sind, und es gebe eine Anfrage desselben Typs. Dann existiert ein nicht-deterministischer Algorithmus mit polynomialem Aufwand, der entscheidet, ob zu der gegebenen Anfrage eine Lösung aus den gegebenen konjunktiven Sichten existiert, und wenn ja, diese Lösung angibt.

Verwendet man in der Anfragesprache zusätzlich arithmetische Vergleiche, so läßt sich allerdings nur noch ein Algorithmus mit exponentiellem Aufwand angeben. (s. [24])

Es bleibt anzumerken, daß ein polynomialer Aufwand für Anfragen ohne arithmetische Vergleiche hoch erscheinen mag. Hierzu muß jedoch gesagt werden, daß dies trotzdem als positives Ergebnis gewertet wird, da Anfragen meistens ziemlich kurz sind, der Aufwand aber auf alle Fälle nie höher als polynomial ist.

3.6 Zusammenfassung

In dieser Ausarbeitung haben wir das Konzept einer Mediatorarchitektur kennengelernt. Hierbei ist es durch vielfältige Kombinationen von Mediatoren möglich, heterogene Informationen zusammenzuführen und auszuwerten. Als Schnittstelle zu den eigentlichen Informationsquellen werden dazu sog. Translatoren (auch wrappers) eingesetzt. Zur Formulierung von Anfragen an einen Mediator (entweder durch einen Benutzer oder einen anderen Mediator) ist eine Anfragesprache nötig. Hierzu wurde die SQL-nahe Sprache LOREL und die auf Prädikatenlogik basierende Sprache MSL vorgestellt, wobei letztere auch zur Spezifikation von Mediatoren eingesetzt werden kann. Genauer gesagt kann mit MSL eine sog. Sicht definiert werden, die ein Mediator liefern soll. Es wurde gezeigt, daß unter bestimmten Umständen aber auch Anfragen beantwortet werden können, die keiner vorhandenen einzelnen Sicht entsprechen. Und es wurde eine Aufwandsabschätzung gegeben für das Feststellen, ob eine solche Anfrage durch das konjunktive Verknüpfen mehrerer vorhandener Sichten beantwortbar ist. Ohne arithmetische Ausdrücke war dieser Aufwand polynomial und mit arithmetischen Ausdrücken exponential. Letztendlich ist damit auch eine Grundlage einer Aufwandsabschätzung gegeben für das mögliche automatische Erstellen von Translatoren.

Kapitel 4

Datenintegration durch automatische Daten-/Schemaanalyse

Hans-Jörg Fischer

Kurzfassung *An dieser Stelle wird auf das Problem der Daten- und Schemaanalyse eingegangen und für beide Teilprobleme Algorithmen vorgestellt. Bei der Schemaanalyse wird hierbei auf das CANDIDE-Datenmodell zurückgegriffen, das die Möglichkeit bietet, auch komplexere semantische Eigenschaften abzubilden. Das Problem der Datenanalyse wird im Kontext mit relationalen Datenbanken betrachtet, wobei die* Sorted Neighborhood *Methode vorgestellt wird.*

4.1 Einleitung

Die Notwendigkeit der Datenintegration kann durch eine Vielzahl von Anwendungen, insbesondere auch beim Data-Warehousing entstehen: es ergibt sich die Notwendigkeit, mehrere völlig unabhängig voneinander modellierte Datenbanken zu einer zusammenzufassen. Meist wurde beim Entwurf dieser Quelldatenbanken nicht nur ganz unterschiedliche Schwerpunkte bezüglich der Datenrepräsentation gesetzt, sondern auch auf völlig verschiedene Datenmodelle zurückgegriffen.

Ideal wäre eine selbsttätig ablaufende Integration der verschiedenen Datenbanken. Nach obiger Problemstellung ist aber einleuchtend, daß man dieses Ziel sicher nicht erreichen wird; dafür ist die Semantik gegebener Objekte beispielsweise in einer relationalen Datenbank nur unzureichend dargestellt. Es ist im Integrationsprozeß nicht möglich, aus den Daten und den Schemainformationen genug Schlüsse zu ziehen, um die Datenschemata der Quelldatenbanken vollständig in Relation zueinander zu stellen und eine Datenintegration durchzuführen.

Daher wird man vielmehr versuchen, den Benutzer sinnvoll einzubinden: beispielweise kann das System anhand der gegebenen Daten Vorschläge zur Schemaintegration ausarbeiten, bei denen der Benutzer eine Wertung vornimmt. Auf jeden Fall wird ein manuelles Eingreifen notwendig sein.

An dieser Stelle wird die in [27] vorgestellte Methode der Schemaintegration eingegangen und bezüglich der Datenintegration auf die in [26] beschriebenen Vorgehensweisen zurückgegriffen. Hierbei wird unter anderem auf das CANDIDE Datenmodell bei der Schemaintegration und auf die *Sorted Neighborhood*-Methode zur Durchführung der Datenintegration eingegangen.

4.2 Integrationsprozeß

Der hier vorgestellte Integrationsprozeß gliedert sich in drei wesentliche Teile, wie sie in Abb. 4.1 veranschaulicht werden.

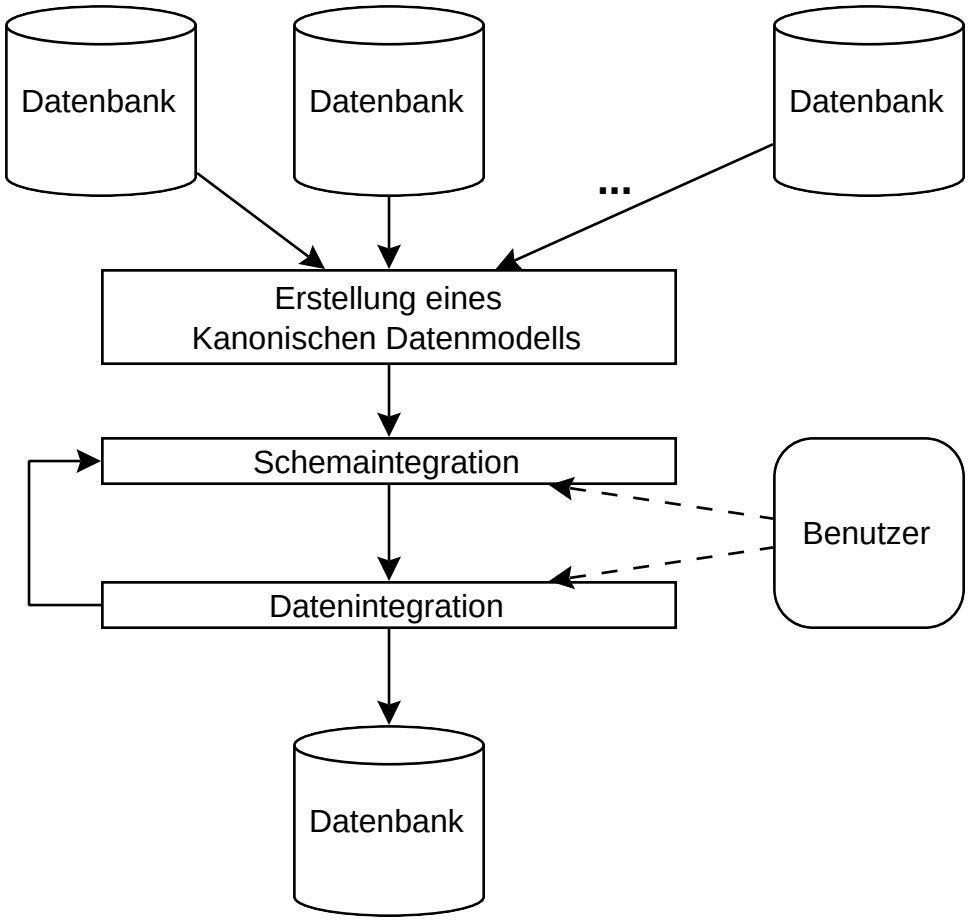


Abbildung 4.1: Ablauf der Datenbankintegration

Zuerst werden die Schemata der Quelldatenbanken in das *Kanonische Datenmodell* überführt, um einheitlich im weiteren Verlauf des Prozesses die Beziehungen zwischen den Datenbanken herausarbeiten zu können.

Daran schließt sich die Schemaintegration an, die die Schemata zusammenführt und dabei auf eine gegebene Menge von Axiomen zurückgreift, die an dieser Stelle nicht weiter vorgestellt werden soll. Bei der Datenintegration wird versucht, mehrfach gegebene Datensätze

als solche zu erkennen und diese dann zusammenzuführen. Diese beiden Teilprozesse haben unmittelbaren Einfluß aufeinander, da beispielsweise durch das Zusammenführen von Datensätzen durchaus neue Informationen über die gegebenen Schemata gewonnen werden können und damit das Ergebnis der Schemaintegration nochmals überarbeitet werden muß.

4.3 Kanonisches Datenmodell

Wie bereits angesprochen ist es notwendig, alle Datenbanken einheitlich mit einem Schema darzustellen. Dabei greift man auf das *Kanonische Datenmodell* zurück. Es dient dann im weiteren als Grundlage für die Schemaintegration.

Aus diesem Vorgehen lassen sich unmittelbar mehrere Forderungen an das *Kanonische Datenmodell* ableiten:

- Es muß in dem Sinn vollständig sein, daß alle in den Quelldatenbanken modellierten Eigenschaften erhalten bleiben. Ansonsten ist davon auszugehen, daß die Quelldaten in der neuen einheitlichen Datenbank nicht richtig bzw. vollständig dargestellt sind.
- Es sollte minimal sein, um die weitere Analyse zu vereinfachen, d. h. jedes Attribut mit allen seinen Eigenschaften läßt sich auf genau eine Art darstellen. Anders ausgedrückt ist das Datenmodell nicht redundant.
- Es muß Klassen- und Attributhierarchien darstellen können, um die im weiteren dargestellte Vorgehensweise der automatischen Schemaanalyse zu unterstützen.

Exemplarisch soll hier eine Untermenge des CANDIDE Modells vorgestellt werden.

4.3.1 Einführung in das CANDIDE Modell

CANDIDE gehört zur Familie der KL-ONE Systeme und ist im wesentlichen eine Erweiterung des KANDOR Wissens-Repräsentations-Systems. Bereits aufgrund dieser Verwandtschaft kann man davon ausgehen, daß CANDIDE vielfältige Möglichkeiten beim semantischen Modellieren bietet, da Wissens-Repräsentations-Systeme diesbezüglich hohe Anforderungen stellen.

Wesentlich bei diesem Datenmodell sind *Klassen* und *Attribute*, die beide je in einer von einander unabhängigen Hierarchie verwaltet werden.

Klassen haben eine beliebige Anzahl von Basisklassen. Eine Klasse besitzt hierbei mindestens die Eigenschaften aller ihrer Basisklassen; insbesondere besitzt sie immer eine Menge von Attributen, die eine Obermenge der Vereinigung der Attributmengen aller ihrer Basisklassen ist. Am Kopf der Hierarchie steht hierbei `Thing` als allgemeinste Repräsentation ohne Attribute.

Ein Attribut hat höchstens ein Basisattribut. Jedes Attribut impliziert sein Basisattribut, d. h. daß ein gegebenes Attribut bezüglich seiner Eigenschaften wie Wertebereich usw. als *Spezialisierung* seines Basisattributes gesehen werden kann. Die Wurzel der Hierarchie ist hierbei `Top`. Desweiteren kann an ein Attribut weitere Bedingungen gestellt sein:

`max` n: Das Attribut kann höchstens n Werte aus einem Wertebereich annehmen.

`some` n: Es gibt mindestens n Werte, jeder davon gehört zu einem bestimmten Wertebereich.
Dieser Operator entspricht dem Existenzquantor.

`exactly` n: Es gibt genau n Werte eines Wertebereichs. `exactly` ist hierbei eine Kombination von `max` und `some`.

`all`: Alle Werte des Attributes müssen zu einem bestimmten Wertebereich gehören.
Somit entspricht dieser Operator dem Allquantor.

Wertebereiche können als Klassenname gegeben sein (`class` *`<Klassenname>`*, oder aber als `string`, `integer`, `real`, oder mittels Konstruktoren wie `range` (Angabe eines Intervalls über `integer` oder `real`), `set` A (Menge), `setdif` A B (Menge $A \backslash B$) und `composite` A B (Menge $A \cup B$).

Dieser sehr hohe Detaillierungsgrad, mit denen Datenbanken in CANDIDE modelliert werden können, sind eine wesentliche Voraussetzung für die automatische Schemaanalyse: die Menge der Schlüsse, die man bei diesem Prozeß ziehen kann, stehen unmittelbar mit dem semantischen Reichtum des Datenmodells in Zusammenhang.

Beispiel

Im folgenden ist beispielhaft das Überführen einer Tabelle einer relationalen Datenbank in ein entsprechendes CANDIDE Modell dargestellt.

Gegeben ist die Tabelle `Employee`:

Name	SocialSecurity	Departement	Salary
George Fitzgerald	9374848	Financial	695.40
William Riker	1701170	Ingeneering	913.40
John Miller	8733769	Financial	1112.60

Durch automatische Schemaüberführung würde sich bei einer möglichen Herangehensweise folgende CANDIDE Definition ergeben:

```
class Employee defined
  superclass Thing
  Name           : all string
  SocialSecurity : all integer range (1701170, 9374848)
  Departement    : all string
  Salary         : all real range (695.40, 1112.60)
```

Die Tabelle wird somit durch eine Klasse dargestellt. Desweiteren hat das System versucht, die Definitionsbereiche der Daten bei den `integer` Feldern festzulegen.

Bei den Feldern `Name` und `Departement` war das nicht möglich, da auch innerhalb der relationalen Datenbank keine weiteren Informationen zur Verfügung standen.

Die Felder `SocialSecurity` und `Salary` wurden einfach über ihren Wertebereich, der sich aus den gegebenen Daten der Quelldatenbank ergeben hat, definiert. An dieser Stelle nun ist ein Eingriff des Benutzers notwendig, da diese Bereiche so nicht stimmen. Statt dessen wird man dem System mitteilen, daß das Feld `SocialSecurity` genau sieben Stellen haben muß und `Salary` zwischen `(0, infty)` liegt.

4.4 Schemaintegration

Betrachten wir nun den zweiten Schritt des Prozesses, die Schemaintegration. Sie läuft wie in Abb. 4.2 dargestellt ab und gliedert sich somit in drei Teile.

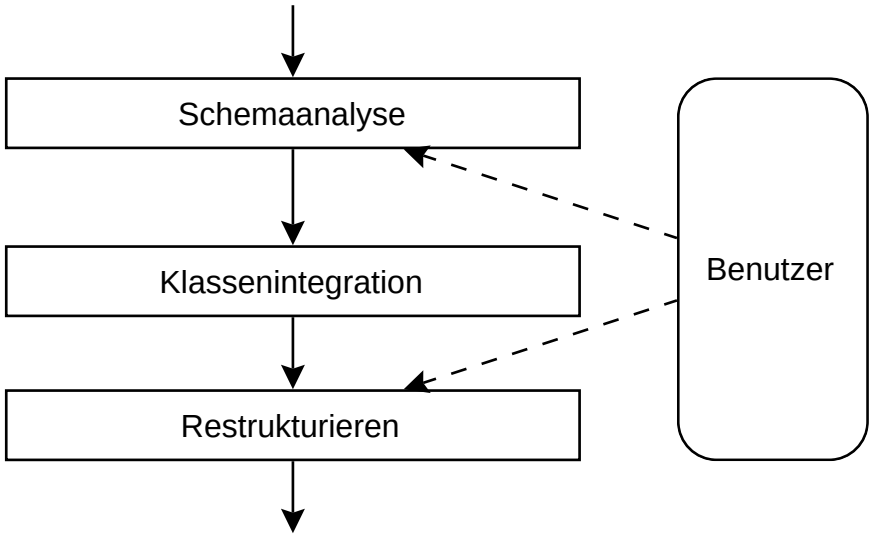


Abbildung 4.2: Ablauf der Schemaintegration

Als Ergebnis der Schemaanalyse erhält man ein globales Attributschema, in dem die Quellschemata aufgehen. Hierbei sind insbesondere die Relationen der einzelnen Attribute zueinander von Bedeutung.

Im zweiten Schritt dieses Teilprozesses werden die Klassen herausgearbeitet. In dem hier vorgestellten Ablauf ist dieser Prozeß vollständig automatisiert. Man kann jedoch auf das Ergebnis im folgenden dritten Schritt, beim Restrukturieren, einwirken und als Benutzer Änderungen einbringen.

Formal definieren wir $\mathrm{RWS}(U)$ als gewünschte Semantik unseres Universums $U = U(C, A)$. Weiter sei $\mathrm{RWAS}(a)$ die Semantik eines Attributes $a \in A$ und $\mathrm{RWCS}(c)$ diejenige einer Klasse $c \in C$.

Sei c_i mit $1 \leq i \leq n$ eine Aufzählung aller Klassen in C und entsprechend a_j mit $1 \leq j \leq m$ eine Aufzählung aller Attribute aus A. Dann gilt

$$\mathrm{RWS}(U) = \{\mathrm{RWCS}(c_i)\} \cup \{\mathrm{RWAS}(a_j)\}, \qquad 1 \leq i \leq n, 1 \leq j \leq m,$$

das heißt daß die Semantik des Universums als Vereinigung derer der Klassen und deren Attribute aufgefaßt werden kann. Beispielsweise kann man so das semantische Universum einer Studentendatenbank als Vereingung der Semantik der Klassen `Student` und die deren Attribute `stud_name`, `stud_id` betrachten.

4.4.1 Schemaanalyse

Die Schemaanalyse beginnt mit dem Ermitteln der Verhältnisse, die zwischen einzelnen Attributen bestehen. Auf dieser Basis werden dann entsprechende Hierarchien entworfen. Hierbei werden Korrespondenzen zwischen dem semantischen Raum RWAS(A), also dem Definitionsbereich, und den Attributen ermittelt, und damit dann die Attribute wie folgt in Relation gestellt:

- `contained-in/contains` *(beinhaltet/beinhalten)* Ein Attribut $a \in A$ impliziert das andere Attribut $b \in A$ in dem Sinne, daß a eine Spezialisierung von b ist. Somit ist b Basisattribut von a.
- `equivalent` *(entsprechend)* Die Attribute sind äquivalent. Dies ist genau dann der Fall, wenn beide Attribute sich gegenseitig implizieren. Diese beiden Attribute sind im weiteren zu einem einzigen zusammenzuführen.
- `disjointed` *(disjunkt)* Die Attribute sind disjunkt, d.h. unabhängig.

Nachdem alle equivalenten Attribute zusammengeführt wurden ist somit eine strenge partielle Ordnung $(A, \subset)$ über den Attributen A gegeben.

Wie man bereits im CANDIDE Beispiel gesehen hat ist es dem System im Allgemeinen nicht möglich, alle Attributrelationen korrekt festzustellen. Man ist letztlich auf den Benutzer angewiesen, um ein vollständiges und richtiges Schema zu erstellen. Dieser wird beispielsweise an dem vom System vorgeschlagenen Schema Änderungen anbringen, wobei darauf zu achten ist, daß die Attributhierarchie nicht zu flach ist. In diesem Fall könnte im folgenden keine sinnvolle Klassenhierarchie erstellt werden, da die Möglichkeit für das System, Schlüsse über Abhängigkeiten zu ziehen, zu beschränkt wären.

Beispiel

Ein Beispiel hierfür ist in Abb. 4.3 und Abb. 4.4 gegeben. Unter einem semantischen Block versteht man hierbei eine Teilmenge des semantischen Universums.

1. $SC_1 = \{Q_1, Q_2\}$
   Der semantische Block SC_1, im folgenden als `person_name` bezeichnet, beinhaltet `stud_name`, `grad_name` und `emp_name`, d. h. daß bei jedem Auftreten von `stud_name`, `grad_name` und `emp_name` nur Werte zu finden sind, die zur semantischen Gruppe `person_name` gehören. (das in diesem Zusammenhang auftretende Problem, zwei Werte als *äquivalent* einzustufen, wird bei der Datenintegration weiter behandelt). Außerdem ist die Menge aller Instanzen von `grad_name` eine Teilmenge derer von `stud_name`: RWAS(`grad_name`) $\subset$ RWAS(`stud_name`).

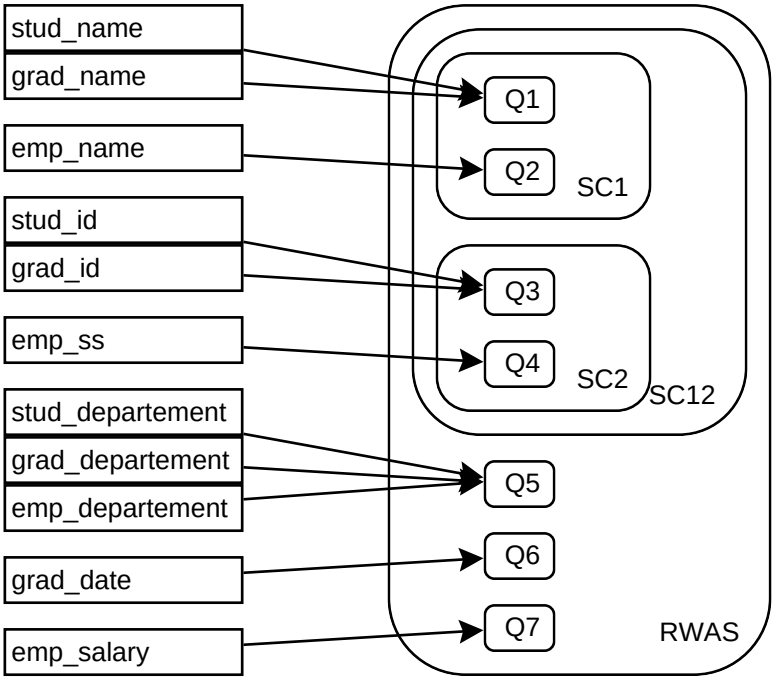


Abbildung 4.3: Abbilden der Attribute in den Semantischen Raum

2. $SC_2 = \{Q_3, Q_4\}$
   Der semantische Block SC_2 (`person_num`) beinhaltet `stud_id`, `grad_id` und `emp_ss`. Auch hier gilt RWAS(`grad_id`) $\subset$ RWAS(`stud_id`).

3. $SC_{12} = \{SC_1 \cup SC_2\}$
   Dieser semantische Block, desweiteren `person_identifier` genannt, beinhaltet die beiden Blöcke SC_1 und SC_2.

Aus diesen Abhängigkeiten im semantischen Raum, wie sie in Abb. 4.3 dargestellt sind, ergibt sich dann die Attributhierarchie aus Abb. 4.4: beispielsweise gilt

$$\text{RWAS}(\texttt{grad\_name}) \subset \text{RWAS}(\texttt{stud\_name}),$$

was unmittelbar dazu führt, daß `stud_name` als Basisattribut von `grad_name` in der Attributhierarchie auftaucht.

4.4.2 Klassenintegration

In dieser Stufe der Integration werden Klassen als Sinngruppen bzw. semantische Objekte erzeugt und die Beziehungen der Klassen untereinander ermittelt. Dies erfolgt auf Basis der Attributverhältnisse, die im vorherigen Schritt ermittelt wurden. Klassen können in vier verschiedenen Relationen stehen:

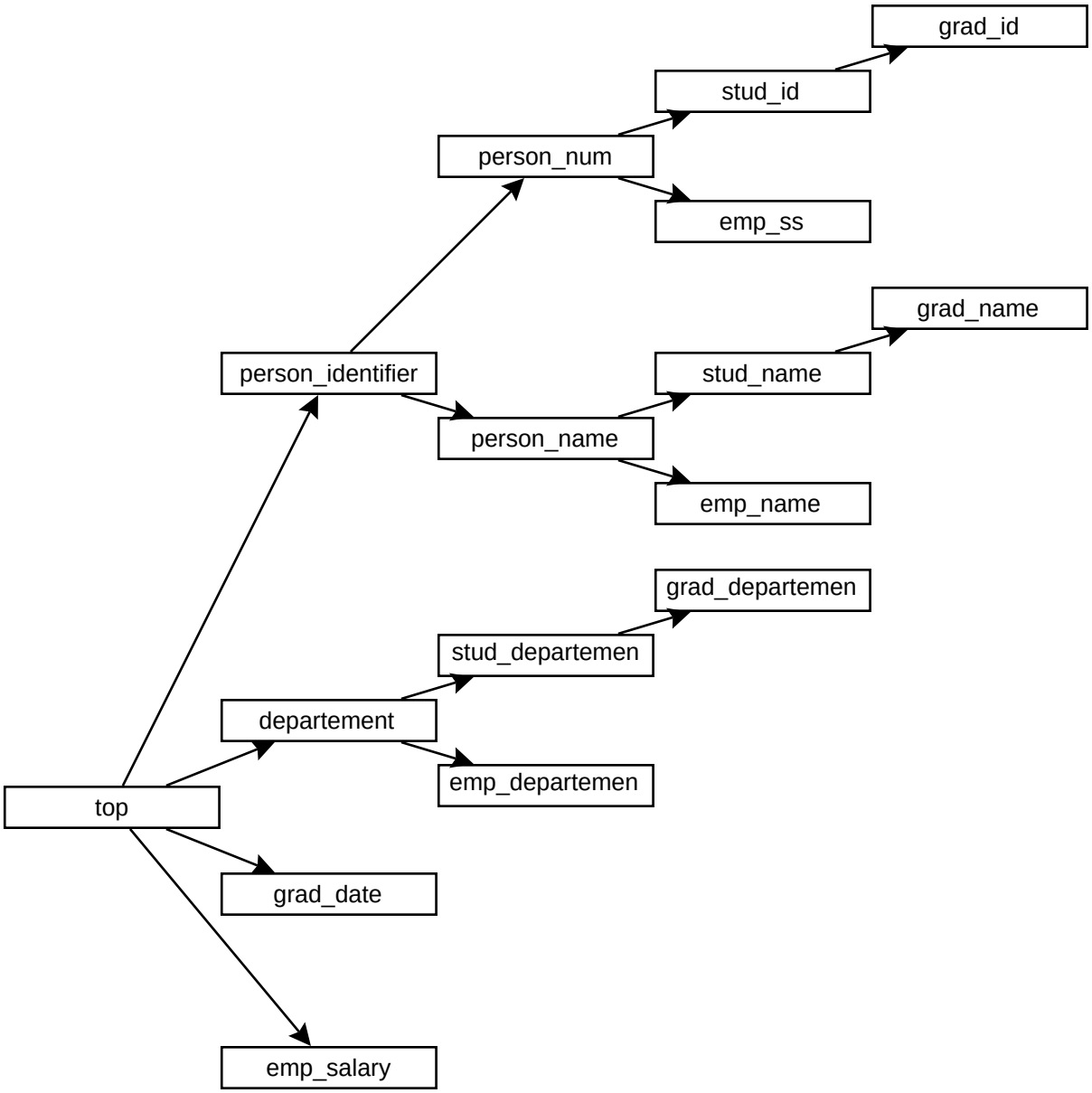


Abbildung 4.4: Attributhierarchie

- `equal` *(identisch)* Die beiden verschiedenen gegebenen Klassen entsprechen sich vollständig:

  $$\texttt{equivalent}(f, g) = \texttt{true} \Leftrightarrow E[f] \equiv E[g]$$

  Demnach sind sie im weiteren Prozeß zusamennzuführen, da sie das gleiche semantische Objekt modellieren.

- `contained-in/contains` *(beinhaltet/beinhalten)* Eine der beiden Klassen ist eine Spezialisierung der anderen:

  $$\texttt{subsume}(f, g) = \texttt{true} \Leftrightarrow E[f] \supseteq E[g]$$

  Somit ist die eine Klasse eine Basisklasse der anderen.

- `overlap` *(überschneiden)* Die Klassen modellieren verwandte Bereiche:

  $$\texttt{overlap}(f, g) = \texttt{true} \Leftrightarrow E[f] \cap E[g] \neq \emptyset$$

  Es ist eine neue Klasse in das Schema einzufügen, welche Basisklasse von beiden gegebenen Klassen ist und den überdeckten Bereich beider modelliert (beispielsweise zwei Klassen `car` und `truck` mit einer neuen gemeinsamen Basisklasse `vehicle`).

- `disjointed` *(disjunkt)* Die Klassen sind disjunkt:

  $$\texttt{disjoint}(f, g) = \texttt{true} \Leftrightarrow E[f] \cap E[g] = \emptyset$$

  Somit haben die beiden Klassen keinen direkten semantischen Zusammenhang.

Hierbei stehen f, g für Klassen aus C und $E[f]$, $E[g]$ für ihre entsprechenden Instanzen.

Die Möglichkeit der Automatisierung ergibt sich aus dem Zurückführen der vielfältigen Relationen auf Teilverhältnisse, die vom System vergleichsweise einfach erkannt werden können und auf deren Basis die neuen Klassen in das Schema eingefügt werden können. Hierbei ist die korrekte Stelle, an der die neue Klasse einzufügen ist, durch zwei Faktoren bestimmt. So muß die neue Klasse unterhalb derer, die sie zusammenfassen und dabei selbst die größte bisherige Spezialisierung haben, stehen. Andererseits muß sie oberhalb von Klassen stehen, deren Attribute die ihrigen implizieren.

Beispiel

Die Rolle, die bei diesem Prozeß der Attributhierarchie zukommt, kann man anhand des folgenden Beispiels entnehmen: gegeben seien die beiden Definitionen in CANDIDE:

```
class C1 defined
  superclass Thing
  attributes
    child : all class Man

class C2 defined
  superclass Thing
  attributes
    son : all class Person
```

Anhand der Definition läßt sich vermuten, daß `C2` in der Klassenhierarchie oberhalb von `C1` steht. Dies gilt genau dann, wenn

`child : all class Man` $\rightarrow$ `son : all class Person`.

Aus der Menge der gegebenen von Abhängigkeitsregeln folgt u. a.:

`if attribute` r `is above` s `then` r `: all` $f \rightarrow s$`: all` f.
`if subsume`(g, f) `= true then` s `: all` $f \rightarrow s$ `: all` g

Da `child` in der Attributhierarchie oberhalb von `son` steht, erhält man durch Anwendung der 1. Regel

`child : all class Man` $\rightarrow$ `son : all class Person`

Da `Person` in der Klassenordnung oberhalb von `Man` steht, erhalten wir durch die 2. Regel

`son : all class Man` $\rightarrow$ `son : all class Person`

Aus diesen beiden Folgerungen und der Transitivität der Implikation schließen wir

`child : all class Man` $\rightarrow$ `son : all class Person`

Wir haben somit auf Basis unserer Informationen über die Verhältnisse von `child/son` und `Person/Man` festgestellt, daß `C2 C1` zusammenfasst.

Man erkennt auch sofort, daß eine möglichst detaillierte Attributhierarchie für das gelingen der Klassenintegration von wesentlicher Bedeutung ist: eine zu flache Ordnung läßt nicht hinreichend viele Schlüsse zu, um eine exakte Klassenhierarchie zu erzeugen.

4.4.3 Restrukturierung

Nachdem die gegebenen Schemata im o. g. Prozeß zusammengeführt wurden kann der Benutzer auf einen Satz von Operatoren zurückgreifen, um die entstandene Klassenstruktur zu verändern:

- `delete`(f): Löschen der Klasse f aus dem Schema und Überprüfung auf Konsistenz. Falls das resultierende Schema inkonsistent ist, wird der Benutzer darüber informiert. Inkonsistenzen können beispielsweise dann auftreten, wenn eine Klasse `Student` ein Attribute der Art *zugehörend* zu einer Klasse `Departement` hat und versucht wurde, letzteres zu löschen.

- c = `generalize1`(f, g): Die beiden Klassen f, g werden herangezogen, um eine neue Klasse c als Verallgemeinerung von beiden zu erzeugen. Hierbei werden die gemeinsamen Attribute beider Klassen verwendet und die an sie gestellten Bedingungen mit *logisch oder* verknüpft. Somit gilt $E[f] \supseteq E[c] \wedge E[g] \supseteq E[c]$.

- $c =$ `generalize2`(f, g): Wie bei `generalize1`, jedoch werden die beiden Klassen f, g aus dem Schema entfernt.
- $c =$ `specialize1`(f, g): Die beiden Klassen f, g werden verwendet, um eine neue Klasse c als Spezialisierung beider zu erstellen. Hierbei erhält c die Vereinigung der Attribute beider Klassen, wobei die mitgeführten Bedingungen mit *logisch und* verknüpft werden: $E[f] \subseteq E[c] \wedge E[g] \subseteq E[c]$.
- $c =$ `specialize2`(f, g): Wie bei `specialize2`, jedoch werden die beiden Klassen f, g aus dem Schema entfernt.

Diese Operationen sind völlig ausreichend, um jede beliebige Veränderung an der Klassenhierarchie vorzunehmen und so die Unzulänglichkeiten der maschinell erstellten Klassenstruktur auszugleichen.

4.5 Datenintegration

Nachdem bisher die Problematik der Schemaintegration diskutiert wurde, wird nun das Problem, die eigentlichen Datensätze aus mehreren Quellen möglichst effizient und richtig zusammenzuführen, betrachtet.

Hierbei ist nicht nur das *Zusammenkopieren* der Datensätze von belang (*merge*), sondern auch das *Zusammenführen* derjenigen Datensätze, die durch fehlerhafte Daten mehrfach vorhanden sind (*purge*). In diesem Zusammenhang können zwei gegensätzliche Fehler auftreten:

- Zwei verschiedene Datensätze werden fälschlicherweise zusammengeführt, d. h. zwei semantisch unabhängige Klasseninstanzen der Quelldatenbanken werden als eine in der Zieldatenbank repräsentiert. Es tritt somit Datenverlust und/oder Datenverfälschung auf.
- Ein mehrfach gegebener Datensatz wird nicht als solcher erkannt und nicht zusammengeführt. Somit tritt ein Datensatz in zwei unterschiedlichen syntaktischen Formulierungen auf.

Man wird die Parameter der Datenintegration in Abhängigkeit davon wählen, welcher der beiden Fälle schwerwiegendere Folgen nach sich zieht.

Beispiel

Es seien zwei Adreßdatenbanken gegeben, die zusammengeführt werden sollen. Hierbei sind drei Datensätze gegeben:

Nr.	Name	Straße	Ort
1.	Michael Schmidt	Böhmstraße 6	99999 Beispielstadt
2.	Michaela Schmidt	Böhmstraße 6	99999 Beispielstadt
3.	Michael Schmitt	Boehmstraße 6	99999 Beispielstadt

Es ist zu vermuten, daß sich alle Datensätze auf den gleichen logischen Satz beziehen. Möglich wäre allerdings ebenfalls, daß in der Familie Schmidt sowohl ein Michael als auch eine Michaela vorhanden sind. Obwohl der erste und dritte Datensatz sich logisch näherstehen als der erste und zweite, unterscheiden sie sich in mehr Zeichen. Man sieht also, daß durch einen einfachen Vergleich von Zeichen keine sinnvolle Beziehung zu ermitteln ist. Möglich wäre in diesem Fall eine phonetische Analyse der Bezeichner, bei der dann `dt` und `tt` sowie `ö` und `oe` äquivalent sind.

Im Rahmen dieses Textes wird im weiteren nur das Problem der Datenintegration in relationalen Datenbanken mit der *Sorted Neighborhood* Methode behandelt; der vorgestellte Algorithmus läßt sich jedoch mit einigen Abwandlungen auch auf CANDIDE-basierte Datenbanken anwenden.

4.5.1 Sorted Neighborhood–Methode

Dieser Algorithmus läßt sich in drei Stufen gliedern:

1. *Erstellen von Schlüsseln.* Hierbei ist wesentlich, daß im Sinne der Zusammenführung ähnliche bzw. passende Datensätze nahezu gleiche Schlüssel erhalten, damit im folgenden Schritt der Sortierung verwandte (d. h. semantisch ähnliche) Datensätze nahe zusammen kommen. Die Wahl des Schlüssels ist wesentlich für das Ergebnisses des Algorithmus.

2. *Sortieren der Daten* auf Basis des erstellten Schlüssels; hierzu verwendet man die einschlägig bekannten Sortierverfahren.

3. *Zusammenführen ähnlicher Datensätze.* Hierbei bewegt man ein Fenster fester Größe w durch die geordnete Liste der Datensätze (vgl. Abb. 4.5). Hierbei wird jeder Datensatz mit den $w \Leftrightarrow 1$ vorgehenden Datensätzen verglichen. Man erkennt sofort, daß ähnliche Datensätze nur dann zusammengeführt werden können, wenn sie höchstens einen Abstand von $w \Leftrightarrow 1$ haben. Die Argumentation, das Datenfenster auf die volle Datenbankgröße zu erweitern und somit das Problem der Schlüsselwahl zu entschärfen, verbietet sich durch den hohen Rechenaufwand, der mit dieser Lösung verbunden wäre. Im allgemeinen wird man daher einen Kompromiß suchen.

Man braucht also einen Durchlauf durch die Datenbank zur Erstellung der Schlüssel, mindestens einen (i. a. mehr) zum Sortieren und mindestens einen weiteren zum Zusammenführen, wobei weiter absehbar ist, daß bei sehr großen Datenbanken Festplatten Ein-/Ausgaben einen wesentlichen Anteil am Zeitaufwand haben. Um das Resultat der Datenintegration zu verbessern, wird man i. A. den oben angegebenen Durchlauf mehrfach wiederholen. Hierbei kann man bei verschiedenen Durchläufen sowohl die Größe des Fensters w variieren, als auch auf verschiedene Schlüssel zurückgreifen, die die Datensätze auf unterschiedliche Art und Weisen beleuchten und entsprechend anders sortieren.

Da wir mit sehr großen Datenbanken arbeiten versuchen wir, die Datenbank in Blöcke (*clusters*) so aufzuteilen, daß potentiell passende, d. h. zusammenzuführende, Datensätze im gleichen Block zu finden sind. Wir haben dann insbesondere die Möglichkeit, die unterschiedlichen Blöcke auf verschiedenen Rechnern weiterzuverarbeiten und damit Parallelverarbeitung zu ermöglichen.

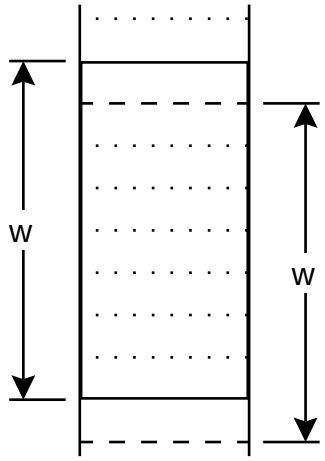


Abbildung 4.5: Fensterdurchlauf beim Zusammenführen

Aufteilung in Blöcke

Die einfachste Methode, Blöcke zu bilden, wäre zweifellos die Aufteilung der Liste in gleichgroße Teillisten, die dann entsprechend auf die Maschinen verteilt werden. Diese Vorgehen verbietet sich jedoch aus dem Grunde, daß semantisch gleiche Datensätze, die an Blockgrenzen liegen, dann nicht mehr zusammengeführt werden können, da sie auf verschiedenen Maschinen abgearbeitet werden.

Statt dessen wird an dieser Stelle die *Clustering* Methode vorgestellt. Hierzu bilden wir über jeden Datensatz einen n stelligen Schlüssel, der in einen n dimensionalen Würfelraum zeigt. Durch diese Zerlegung im Würfelraum erziehlt man eine wesentlich höhere Lokalität unter ähnlichen Datensätzen, als es bei einer linearen Betrachtungsweise möglich wäre. Diese einzelnen Würfel können nun auf den einzelnen Maschinen weiterverarbeitet werden, wobei auf die *Sorted Neighborhood* Methode zurückgegriffen werden kann und auf den ersten Schritt (*Erstellen von Schlüsseln*) des Algorithmusses verzichtet werden kann, da man auf die bereits erstellten Schlüssel zurückgreifen kann.

Auch bei diesem parallelisierten Verfahren empfiehlt es sich, den Vorgang mehrfach mit unterschiedlichen Schlüsseln zu wiederholen. Somit ist gewährleistet, daß man auch bei vergleichsweise ungünstigen Randbedingungen die meisten ähnlichen Datensätze zusammengeführt werden.

4.6 Schlußbemerkung

Beim Zusammenführen von Datenbanken ist sowohl die Schemaintegration, als auch die Datenintegration von wesentlicher Bedeutung. Hier wurden verschiedene Probleme, die damit verbunden sind, als auch exemplarisch eine Methode vorgestellt.

Bezüglich der Schemaintegration wurde eine Untermenge des CANDIDE Datenmodells eingeführt und auf dessen Basis eine Vorgehensweise zur Integration beschrieben.

Bei der Datenintegration wurde das Problem auf relationale Datenbanken beschränkt und die *Sorted Neighborhood* Methode und eine diesbezügliche Möglichkeit der Parallelisierung umrissen.

Es ist offensichtlich, daß ein vollständig selbsttätig ablaufender Integrationsprozeß nicht zu realisieren ist: es stehen hierfür nicht genug semantische Informationen in den Quelldatenbanken zur Verfügung. Die vorgestellten Algorithmen bieten statt dessen die Möglichkeit, das unumgängliche manuelle Eingreifen in den Integrationsprozeß sinnvoll durch Maschinenläufe zu unterstützen und hierbei auch auf die Möglichkeiten der Parallelverarbeitung zurückzugreifen.

Kapitel 5

Datenaktualisierung beim Data Warehousing

Gregor Bublitz

Kurzfassung *Die Aufgabe des Data–Warehousing ist es, Daten von mehreren verteilten Datenquellen zu integrieren bzw. dem Anwender eine globale Sicht auf die Gesamtheit der Daten zur Verfügung zu stellen. Um eine schnelle Anfrage zu gewährleisten, wird die Sicht materialisiert (d.h. physisch gespeichert) und muß demnach aktuell gehalten werden. Da das Sichtenmanagement (Warehouse) und die realen Daten aber räumlich getrennt sind, müssen bei der Datenaktualisierung zeitliche Verzögerungen berücksichtigt werden, so daß herkömmliche Sichterhaltungsalgorithmen nicht mehr verwendet werden können. Um inkonsistente Zustände zwischen Warehouse und Quellen zu vermeiden, wird im folgenden der ECA (Eager Compensating Algorithmus) und eine Variante davon vorgestellt. Ein anschließender Vergleich mit dem RV–Algorithmus (Recomputing the View), der die Sicht des Warehouses komplett neu berechnet, zeigt, daß der ECA–Algorithmus auch effizient ist. Ein weiterer Algorithmus der sich mit Sichtenerhaltung beim Warehouse beschäftigt, ist der Strobe–Algorithmus, der hier aber nur kurz vorgestellt wird.*

5.1 Überblick

5.1.1 Problemstellung

Grundsätzlich unterscheidet man beim Data–Warehousing zwei Ausprägungen: Einerseits das virtuelle Warehouse und andererseits das materialisierte Warehouse. Das virtuelle Warehouse hat keine eigenen Daten gespeichert und muß daher bei Anfragen, die vom Anwendern an das Warehouse gestellt werden, bis auf die Ursprungsquellen durchgreifen und sich die benötigten Information zusammensuchen. Das materialisierte Warehouse hingegen speichert sogenannte materialisierte Sichten (Materialized View, MV), d.h. speichert die Daten auch physisch, um dann Anfragen direkt auf dieser Sicht auswerten zu können. Diese Methode ermöglicht es, eine schnelle Anfragebearbeitung zu garantieren, wobei man allerdings den Aufwand für die Datenaktualisierung in Kauf nehmen muß.

Bei der Datenaktualisierung sind noch Spielräume vorhanden, wann und wie oft diese durchgeführt werden soll. Benötigt man nicht zu jedem Zeitpunkt unbedingt eine aktuelle Sicht, so kann die Datenaktualisierung in monatlichen oder täglichen Intervallen vollzogen werden. Falls aber eine stets aktuelle Sicht gefordert ist, muß jede Änderung sofort also in Echtzeit an die Sicht weitergegeben werden. Wir werden uns im folgenden auf den letztgenannten Fall konzentrieren.

Die Möglichkeit zur Materialisierung von Sichten (physische Speicherung) besteht auch in zentralen Datenbanksystemen und wird dort angewendet, wenn man bei einer Anfrage nicht zu lange auf die Auswertung der Sicht warten will. Um die Sicht aktuell zu halten, müssen Änderungen in den Relationen an die Sicht weitergegeben werden. Das geschieht im zentralen Fall als atomare Operation, d.h. Änderungen können sich nicht zeitlich überlappen.

Beim Data–Warehousing sind Warehouse und Quellen nicht mehr zentral vorhanden, sondern auf verschiedenen Rechnern verteilt (siehe Abbildung 5.1).

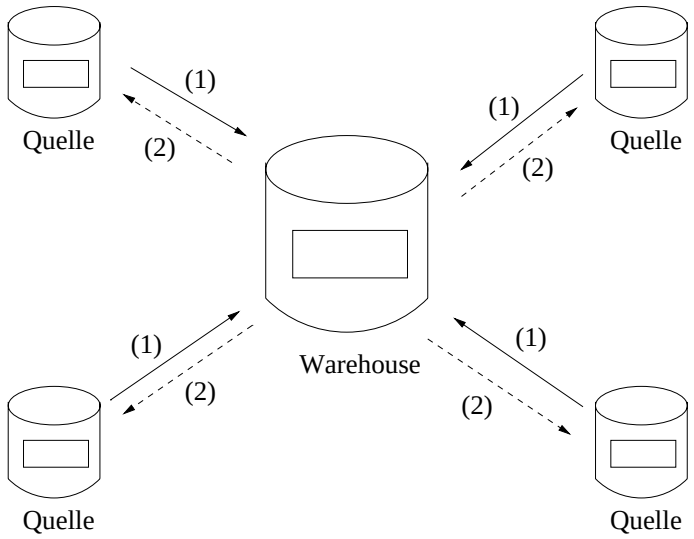


Abbildung 5.1: Warehouseumgebung

Da das Warehouse und die Quellen wie oben angedeutet räumlich getrennt sind, stellt eine Sichtänderung dann aber keine atomare Operation mehr dar. Jede Änderung (in Abbildung 5.1 mit (1) gekennzeichnet), die von einer Quelle an das Warehouse geschickt wird, kann daher zur Folge haben, daß vom Warehouse weitere Anfragen (2) zu den restlichen Quellen gesendet werden, um die Sicht aktuell zu halten. Probleme treten dann auf, wenn sich eine Anfrage des Warehouses mit einer Änderung der Quelle, die die Anfrage beantworten soll, zeitlich überlappen. Die Sicht des Warehouses zum Zeitpunkt der Anfrage und die Sicht der Quelle geraten in einen inkonsistenten Zustand.

Um diese Inkonsistenzen zu beseitigen bzw. zu vermeiden, betrachten wir im folgenden den ECA sowie den Strobe–Algorithmus. Der Unterschied zwischen den beiden Algorithmen besteht in der Anzahl der Quellen, für die der jeweilige Algorithmus ausgelegt ist. ECA beschäftigt sich nur mit einer Quelle, hingegen wird bei Strobe auf eine Warehouseumgebung mit mehreren Quellen eingegangen.

5.1.2 Beispiele

Um den Mechanismus der Datenaktualisierung mit seinen Änderungen und Anfragen besser zu verstehen, wird zunächst ein Szenario betrachtet, das eine korrekte Sicht beim Warehouse hinterläßt.

Beispiel 1: Angenommen die Quelle hat zwei Relationen $r_1(W, X)$, $r_2(X, Y)$ mit jeweils 2 Attributen, wobei r_2 zu Beginn leer sei. Als Sicht des Data–Warehouse nehmen wir folgenden relationenalgebraischen Ausdruck $V = \Pi_W(r_1 \bowtie r_2)$ an, der eine Verbindung der beiden Relationen mit anschließender Projektion auf das Attribut W beschreibt. Anfangs enthält daher die materialisierte Sicht MV keine Elemente.

r_1 :

W	X
1	2

r_2 :

X	Y

MV :

W

Nun soll an der Quelle eine Änderungsoperation $U_1 = insert(r_2, [2, 4])$ durchgeführt werden.

Ein einfacher Algorithmus zur Sichterhaltung (dieser wird in Abschnitt 1.3.1 genau beschrieben) bewirkt folgendes.

1. Bei der Quelle wird die Änderung U_1 = insert(r_2,[2,4]) durchgeführt und eine Nachricht zum Warehouse gesendet, daß U_1 aufgetreten ist.

2. Nach dem Erhalt von U_1 generiert das Warehouse eine Anfrage $Q_1 = \Pi_W(r_1 \bowtie [2, 4])$ und schickt diese zur Quelle, um alle Tupel aus r_1 zu erhalten, die mit dem neu eingefügten Tupel gejoint werden können.

3. Q_1 wird von der Quelle ausgewertet und das Ergebnis $A_1 = ([1])$ an das Warehouse zurückgeschickt.

4. Das Warehouse fügt das Ergebnistupel A_1 in die Sicht ein. Aus $MV = \emptyset$ wird $MV = ([1])$.

Die Probleme treten, wie schon gesagt, erst auf, wenn sich eine Anfrage vom Warehouse und ein Änderung der Quelle zeitlich überlappen. Das nächste Beispiel soll dies verdeutlichen.

Beispiel 2: Wir betrachten wieder die zwei Basisrelationen r_1, r_2 sowie die Sicht $V = \Pi_W(r_1 \bowtie r_2)$.

r_1 :

W	X
1	2

r_2 :

X	Y

MV :

W

Es werden nun bei der Quelle zwei aufeinanderfolgende Änderungen ausgeführt: $U_1 = insert(r_2, [2, 3])$ und $U_2 = insert(r_1, [4, 2])$.

1. Bei der Quelle wird die Änderung $U_1 = insert(r_2, [2,3])$ durchgeführt und eine Nachricht zum Warehouse geschickt, daß U_1 aufgetreten ist.

2. Nach dem Erhalt von U_1 generiert das Warehouse eine Anfrage $Q_1 = \Pi_W(r_1 \bowtie [2,3])$ und schickt diese zur Quelle.

3. Bei der Quelle wird die Änderung $U_2 = insert(r_1, [4,2])$ durchgeführt und eine Nachricht zum Warehouse geschickt, daß U_2 aufgetreten ist.

4. Das Warehouse empfängt U_2 und schickt $Q_2 = \Pi_W([4,2] \bowtie r_2)$.

5. Die Quelle erhält Q_1 und wertet die Anfrage auf der aktuellen Relation $r_1 = ([1,2],[4,2])$ und $r_2 = ([2,3])$ aus. Die Antwort $A_1 = ([1],[4])$ wird zum Warehouse zurückgeschickt.

6. Das Warehouse evaluiert die Sicht zu $MV \cup A_1 = ([1],[4])$.

7. Bei der Quelle kommt nun Q_2 an und wird auf den gleichen Basisrelationen wie Q_1 ausgewertet. Die Antwort $A_2 = ([4])$ wird zum Warehouse geschickt.

8. Das Warehouse macht wiederum eine Änderung der Sicht und erhält $MV \cup A_2 = ([1],[4],[4])$. [1]

Wie man sieht, ist die Sicht nun inkorrekt. Denn hätte man einen gewöhnlichen Algorithmus zur Sichterhaltung direkt bei der Quelle angewendet, wäre die korrekte Sicht $MV = ([1],[4])$ daraus hervorgegangen. In unserem Fall haben sich Q_1 und U_2 zeitlich überlappt. D.h. Q_1 trifft auf eine Sicht bei der Quelle, die U_2 schon enthält und deshalb tritt im Endergebnis die [4] doppelt auf. Die erste Vermutung, wie dieses Problem gelöst werden kann, ist, daß das Warehouse sich um die Änderungen kümmern muß, die bei ihm eintreffen, nachdem es eine Anfrage losgeschickt hat. Dieses Problem wird beim ECA–Algorithmus mit Hilfe von ausgleichenden Anfragen gelöst. Daher kommt auch der Name des Eager Compensating (Ausgleichend) Algorithmus.

5.2 Notation

Zur Darstellung der Daten und der Sichtendefinition werden wir das relationale Modell bzw. die relationale Algebra verwenden. Eine Sichtendefinition hat dabei folgende Form:

$$V = \Pi_{proj}(\sigma_{cond}(r_1 \times r_2 \times \ldots \times r_n)),$$

wobei *proj* eine Menge von Attributnamen, *cond* ein boolscher Ausdruck und $r_1, \ldots, r_n$ die Basisrelationen der Quellen sind. Eine Änderung kann zwei verschiedene Ausprägungen haben:

- $insert(r,t)$; Füge Tupel t in die Relation r ein, z.B. $insert(r_4, [1,3])$

[1] Bei der Vereinigung werden keine Duplikate entfernt, d.h. es werden Mehrfachmengen betrachtet.

- $delete(r, t)$; Lösche Tupel t aus der Relation r, z.B. $delete(r_2, [2, 4])$

Die Modifikation eines bestehenden Tupels wird durch ein *insert* gefolgt von einem *delete* realisiert.

Die Notation des Algorithmus gründet sich auf sogenannte Ereignis–Aktionspaare, die einerseits bei der Quelle und andererseits beim Warehouse auftreten, wie Abbildung 5.2 illustriert.

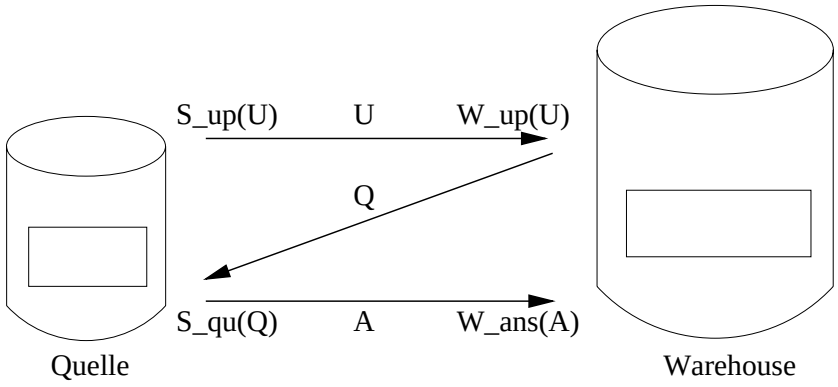


Abbildung 5.2: Ereignis–Aktionspaare

Auf Seite der Quelle gibt es zwei Typen von Ereignis–Aktionspaaren:

- `S_up`(U): Bei der Quelle wird die Änderung U durchgeführt und eine entsprechende Nachricht an das Warehouse geschickt, daß U aufgetreten ist, d.h. das Ereignis `W_up`(U) wird dort ausgelöst.
- `S_qu`(Q): Die Quelle bearbeitet eine Anfrage Q auf den relevanten Basisrelationen und schickt danach die Antwort A an das Warehouse zurück, d.h. `W_ans`(A) wird ausgelöst.

Auf der Seite des Warehouses gibt es ebenfalls zwei Typen von Ereignis–Aktionspaaren:

- `W_up`(U): Das Warehouse erhält eine Änderungsbenachrichtigung U, generiert eine Anfrage Q und schickt diese zur Bearbeitung an die Quelle in Form eines Ereignisses `S_qu`(Q).
- `W_ans`(A): Das Warehouse bekommt die Antwortrelation A für seine Anfrage Q und ändert die Sicht.

5.2.1 Aufbau der Anfragen

Die Anfragen werden in ähnlicher Weise aufgebaut und definiert wie die Sicht, nur daß sie aus mehreren Teiltermen bestehen können. Eine allgemeine Anfrage sieht deshalb wie folgt aus:

$$Q = \sum_i T_i$$

$$\text{mit } T_i = \Pi_{proj}(\sigma_{cond}(\tilde{r}_1 \times \tilde{r}_2 \times \ldots \times \tilde{r}_n)),$$

wobei $\tilde{r}_i$ jeweils eine Relation r_i oder eine Änderung in Form eines Tupels t_i der Relation r_i sein kann.

Weiterhin bezeichne $V\langle U\rangle$ die Sicht V, bei der die Relation, auf die sich die Änderung U bezieht, durch das in U vorkommende Tupel ersetzt ist. Bei einer Sichtendefinition $V = \Pi_W(r_1 \bowtie r_2)$ und Änderung $U_1 = insert(r_1, [4,2])$ ist beispielsweise $V\langle U_1\rangle = \Pi_W([4,2] \bowtie r_2)$.

Da eine allgemeine Anfrage Q meistens aus mehreren Teiltermen besteht, ist $Q\langle U\rangle = \sum_i T_i\langle U\rangle$, d.h. die Ersetzung wird bei jedem Teilterm der Anfrage durchgeführt. $Q\langle U_1, U_2\rangle$ bedeutet die nacheinandergeschaltete Ersetzung, d.h. man wertet Term $(Q\langle U_1\rangle)\langle U_2\rangle$ aus. Falls U_1 und U_2 auf derselben Relation ausgeführt werden, wird per Definition $Q\langle U_1, U_2\rangle = \emptyset$ gesetzt. [2]

Um mit den Änderungen *insert* und *delete* vereinfacht umgehen zu können, führen wir Vorzeichen für die einzelnen Tupel ein. Bestehende oder eingefügte Tupel erhalten ein positives Vorzeichen, gelöschte Tupel hingegen bekommen ein negatives Vorzeichen. Änderung $U_1 = delete(r_2, [2,5])$ hätte dann in einer Anfrage z.B. folgendes Aussehen: $Q_1 = \Pi_W(r_1 \bowtie \Leftrightarrow[2,5])$. Angenommen r_1 besteht nur aus dem Tupel = (+[1,2]), dann ist die Antwort $A_1 = (\Leftrightarrow[1])$. Da sich die Sicht aus $MV + A_1$ neu berechnet, wird also das Tupel von MV abgezogen. Diese Vereinfachung wird nur benötigt, um im Algorithmus *delete* und *insert* einheitlich behandeln zu können.

5.2.2 Korrektheit

Bei der Datenaktualisierung werden eine Menge von Änderungen $U_1, U_2, \ldots, U_n$ durchgeführt. Bei der Quelle treten eine Reihe von Ereignis–Aktionspaaren (source events) $se_1, se_2, \ldots, se_p$ auf mit entsprechenden resultierenden Zuständen (source states) $ss_1, ss_2, \ldots, ss_p$. Entsprechend gibt es beim Warehouse warehouse events $we_1, we_2, \ldots, we_q$ und entsprechende warehouse states $ws_1, ws_2, \ldots, ws_q$. Dabei sei die Zahl von Ereignisse (und damit implizit auch die Anzahl der Änderungen) auf p bzw. q bei Quelle bzw. Warehouse begrenzt.

Der Zustand der materialisierten Sicht beim Warehouse nach Event we_i ist $V[ws_i]$, wobei V die Sichtdefinition ist. Gleichermaßen gilt bei der Quelle, daß $Q[ss_i]$ das Ergebnis der Anfrage Q ist, nachdem Event se_i stattgefunden hat. Wendet man die Sichtendefinition auf einen Zustand bei der Quelle an, so erhält man den Zustand der Sicht nach Ausführung von Event se_i. Um die Korrektheit der Algorithmen nachzuweisen, benötigt man nun einige an die Warehouseumgebung angepasste Korrektheitskriterien. Diese lassen sich in fünf verschiedene Korrektheitsstufen einteilen, die jeweils aufeinander aufbauen.

[2] Die Erklärung dieser Vereinbarung folgt bei der Vorstellung des Algorithmus in Abschnitt 1.3.

- Konvergenz: Für alle abgeschlossenen Ausführungen gilt $V[ws_q] = V[ss_p]$. D.h. nach der letzten Änderung ist die Sicht des Warehouses mit den Relationen bei der Quelle konsistent.

- Schwache Konsistenz: Für alle Ausführungen und für alle ws_i gibt es ein ss_j, sodaß $V[ws_i] = V[ss_j]$ gilt. Zu jeder Sicht beim Warehouse existiert ein gültiger Zustand der Relationen bei der Quelle.

- Konsistenz: Für alle Ausführungen und für jedes Paar $ws_i < ws_j$ gibt es $ss_k \leq ss_l$, sodaß $V[ws_i] = V[ss_k]$ und $V[ws_j] = V[ss_l]$. Zu jedem Zustand beim Warehouse gibt es eine zugehörigen bei der Quelle, und zwar in der gleichen Reihenfolge.

- Strenge Konsistenz: Konsistenz und Konvergenz

- Vollständigkeit: strenge Konsistenz und zu jedem ss_i existiert ein ws_j, sodaß $V[ws_j] = V[ss_i]$. Es gibt eine komplette Reihenfolgeentsprechung zwischen Zuständen der Sicht und Zuständen der Quelle.

5.3 ECA–Algorithmus

5.3.1 Basisalgorithmus

Wir betrachten zu Beginn den Basisalgorithmus, den wir schon bei Beispiel 2 kennengelernt haben. Ausgehend von diesem Algorithmus wird nachher der ECA–Algorithmus entwickelt.

Algorithmus zur Sichterhaltung

Ereignis–Aktionspaare bei der Quelle:

- $\texttt{S\_up}_i(U_i)$: führe Änderung U_i aus;
  löse Ereignis $\texttt{W\_up}_i(U_i)$ aus

- $\texttt{S\_qu}_i(Q_i)$: empfange Anfrage Q_i;
  $A_i = Q_i[ss_i]$; (ss_i ist der Zustand der Relationen)
  löse Ereignis $\texttt{W\_ans}_i(A_i)$ aus

Ereignis–Aktionspaare beim Warehouse:

- $\texttt{W\_up}_i(U_i)$: empfange Änderung U_i;
  $Q_i = V\langle U_i \rangle$;
  löse Ereignis $\texttt{S\_qu}_i(Q_i)$ aus

- $\texttt{W\_ans}_i(A_i)$: empfange A_i;
  Sicht ändern: $MV = MV + A_i$

Das Beispiel 2 hat gezeigt, daß dieser Algorithmus zu Anomalien führt und somit in einer Warehouseumgebung nach den Korrektheitskriterien nicht einmal konvergent ist. Um diesen Anomalien aus dem Weg zu gehen und eine strenge Konsistenz zu erreichen, benützt der folgende ECA–Algorithmus sogenannten kompensierende Anfragen.

5.3.2 Eager Compensating–Algorithmus

Wie sich in Beispiel 2 schon gezeigt hat, bereiten uns Änderungen Probleme, die beim Warehouse ankommen, nachdem eine Anfrage losgeschickt worden ist. Die Konsistenz zwischen Warehouse und Quelle bliebe gewahrt, wenn eine Serialisierung jeder Änderung und den dadurch ausgelösten Ereignissen gegeben wäre, d.h. wenn Ereignisse folgende Reihenfolge hätten : Änderung1, Anfrage1, Antwort1, Sichtänderung1, Änderung2, Anfrage2, Antwort2, Sichtänderung2, usw., was im zentralen Fall durch die Atomizität garantiert wird. Dann gäbe es keinerlei Überlappungen der einzelnen Ereignisse, was somit gleichzusetzen wäre mit dem zentralen Fall. Da aber in einer Warehouseumgebung dieses Szenario nicht die Regel ist, muß man sich um die Menge der UQS (unanswered query set) kümmern, die im folgenden definiert wird.

Definition: Sei we ein Ereignis beim Warehouse. Dann ist die Menge der unbeantworteten Anfragen für we, $UQS(we)$, die Menge der Anfragen, die vom Warehouse gesendet wurden, bevor we eintraf, aber deren Antworten erst nach we dort ankommen.

Algorithmus (ECA)

Die Ereignisse bei der Quelle sind dieselben wie beim Basisalgorithmus. Nur bei den Ereignissen des Warehouse ändert sich etwas. Zu Beginn sei $COLLECT$ als leer initialisiert.

Ereignisse:

- $\texttt{W\_up}_i(U_i)$: empfange Änderung U_i;
  $Q_i = V\langle U_i\rangle \Leftrightarrow \sum_{Q_j \in UQS} Q_j\langle U_i\rangle$;
  löse Ereignis $\texttt{S\_qu}_i(Q_i)$ bei der Quelle aus

- $\texttt{W\_ans}_i(A_i)$: empfange A_i;
  $COLLECT = COLLECT + A_i$;
  wenn $UQS = \emptyset$
  dann $\{MV \leftarrow MV + COLLECT; COLLECT \leftarrow \emptyset\}$
  sonst tue nichts

Beispiel: Gegeben seien die drei Basisrelationen: $r_1(W, X)$, $r_2(X, Y)$, $r_3(Y, Z)$

r_1 :

W	X
1	2

r_2 :

X	Y

r_3 :

Y	Z

Die Sichtdefinition sei $V = \Pi_W(r_1 \bowtie r_2 \bowtie r_3)$. Folglich ist $MV = \emptyset$ und $COLLECT$ als leer initialisiert. Die drei folgenden Änderungen treten nun auf: $U_1 = insert(r_1, [4, 2])$,

$U_2 = insert(r_3, [5,3])$, $U_3 = insert(r_2, [2,5])$. Wir nehmen an, daß alle drei Änderungen vor irgendeiner Antwort beim Warehouse eintreffen.

Im folgenden werden nur die Ereignisse beim Warehouse betrachtet.

1. $\texttt{W\_up}_1(U_1)$: empfange $U_1 = insert(r_1, [4,2])$
   $Q_1 = V\langle U_1\rangle = \Pi_W([4,2] \bowtie r_2 \bowtie r_3)$
   $\texttt{S\_qu}_1(Q_1)$: Warehouse sendet Q_1

2. $\texttt{W\_up}_2(U_2)$: empfange $U_2 = insert(r_3, [5,3])$
   $UQS = \{Q_1\}$

   $Q_2 = V\langle U_2\rangle \Leftrightarrow Q_1\langle U_2\rangle$
   $\quad = \Pi_W(r_1 \bowtie r_2 \bowtie [5,3]) \Leftrightarrow \Pi_W([4,2] \bowtie r_2 \bowtie [5,3])$
   $\texttt{S\_qu}_2(Q_2)$: Warehouse sendet Q_2

3. $\texttt{W\_up}_3(U_3)$: empfange $U_3 = insert(r_2, [2,5])$
   $UQS = \{Q_1, Q_2\}$

   $Q_3 = V\langle U_3\rangle \Leftrightarrow Q_1\langle U_3\rangle \Leftrightarrow Q_2\langle U_3\rangle$
   $\quad = \Pi_W(r_1 \bowtie [2,5] \bowtie r_3) \Leftrightarrow \Pi_W([4,2] \bowtie [2,5] \bowtie r_3)$
   $\quad\quad \Leftrightarrow \Pi_W((r_1 \Leftrightarrow [4,2]) \bowtie [2,5] \bowtie [5,3])$
   $\texttt{S\_qu}_3(Q_3)$: Warehouse sendet Q_3

4. $\texttt{W\_ans}_1(A_1)$: Warehouse empfängt $A_1 = [4]$
   $COLLECT = ([4]), UQS = \{Q_2, Q_3\}$

5. $\texttt{W\_ans}_2(A_2)$: Warehouse empfängt $A_2 = [1]$
   $COLLECT = ([1],[4]), UQS = \{Q_3\}$

6. $\texttt{W\_ans}_3(A_3)$: Warehouse empfängt $A_3 = \emptyset$
   $COLLECT = ([1],[4]), UQS = \emptyset$
   Warehouse macht eine Sichtänderung : $MV = \emptyset + COLLECT = ([1],[4])$

Der ECA–Algorithmus entspricht dem Korrekheitekriterium der strengen Konsistenz, ist aber nicht vollständig. Dazu müßte jeder Zustand der Quelle sich in einem Zustand beim Warehouse niederschlagen. Dieses kann man nicht erreichen, da das Warehouse erst ein paar Antworten sammelt, bevor die Sicht geändert wird. Ein ausführlicher Beweis für die strenge Konsistenz des ECA–Algoritmus ist in [29] gegeben.

5.3.3 ECA–Key–Algorithmus

Ein effizienteren Algorithmus erhält man, wenn man von der Sichtendefinition fordert, daß sie den Schlüssel jeder Basisrelation enthält. Mit dieser Forderung lassen sich *delete*–Änderungen ohne Anfrage an die Quelle direkt beim Warehouse bearbeiten. Mittels der Operation *key–delete* können in der materialisierten Sicht die entsprechenden Tupel gemäß der Schlüsseleigenschaft gelöscht werden. Für *insert*–Änderungen müssen nicht wie beim gewöhnlichen ECA–Algorithmus ausgleichende Anfragen geschickt werden. Der ECA–Key–Algorithmus stellt sich wie folgt dar:

1. $COLLECT$ wird initialisiert mit dem gültigen MV (nicht die leere Menge). Anstatt Modifikationen direkt in MV zu speichern, wird $COLLECT$ als Kopie von MV verwendet.

2. Wenn ein *delete* beim Warehouse ankommt, ist es nicht nötig eine Anfrage zu senden. Die Schlüsseleigenschaft wird ausgenutzt, um aus $COLLECT$ alle Tupel, die sich durch den Schlüssel identifizieren, direkt zu entfernen. Dazu wird die Operation *key–delete* verwendet.

3. Wenn ein *insert* beim Warehouse ankommt, wird wie sonst auch eine Anfrage zur Quelle geschickt. Die Anfrage sieht aber keine kompensierende Teile vor, sondern hat nur die Form $V\langle U\rangle$.

4. Die Antworten werden wie beim ECA–Algorithmus in $COLLECT$ gesammelt, aber es werden keine Duplikate darin aufgenommen. Das ist bedingt durch die Schlüsseleigenschaften der Sicht.

5. Wenn UQS leer ist, wird der MV geändert, d.h. MV wird durch $COLLECT$ ersetzt.

Beispiel: Die folgenden zwei Basisrelationen sind gegeben, wobei W und Y Schlüsselattribute seien.

r_1 :

W	X
1	2

r_2 :

X	Y
2	3

Die Sichtendefinition sei: $V = \Pi_{W,Y}(r_1 \bowtie r_2)$, und demnach ist $MV = ([1,3])$ sowie $COLLECT = MV$. Wir betrachten die drei folgenden Änderungen: $U_1 = insert(r_2,[2,4])$, $U_2 = insert(r_1,[3,2])$, $U_3 = delete(r_1,[1,2])$.

Ereignisse beim Warehouse:

1. $\texttt{W\_up}_1(U_1)$: Warehouse empfängt $U_1 = insert(r_2,[2,4])$
   $Q_1 = V\langle U_1\rangle = \Pi_{W,Y}(r_1 \bowtie [2,4])$
   $\texttt{S\_qu}_1(Q_1)$: Warehouse sendet Q_1

2. $\texttt{W\_up}_2(U_2)$: Warehouse empfängt $U_2 = insert(r_1,[3,2])$
   $Q_2 = V\langle U_2\rangle = \Pi_{W,Y}([3,2] \bowtie r_2)$
   $\texttt{S\_qu}_2(Q_2)$: Warehouse sendet Q_2

3. $\texttt{W\_up}_1(U_1)$: Warehouse empfängt $U_3 = delete(r_1,[1,2])$
   Operation *key–delete*(V,r_1,[1,2]) löscht alle Tupel der Form [1,x]. Daraus ergibt sich dann, daß $COLLECT = COLLECT \Leftrightarrow ([1,3]) = \emptyset$, $UQS = \{Q_1, Q_2\}$

4. $\texttt{W\_ans}_1(A_1)$: Warehouse erhält Antwort $A_1 = ([3,4])$
   $COLLECT = COLLECT + ([3,4]) = ([3,4])$, $UQS = \{Q_2\}$

5. $\texttt{W\_ans}_1(A_1)$: Warehouse erhält Antwort $A_2 = ([3,3],[3,4])$
   Zuerst wird A_2 zu $COLLECT$ dazugezählt, so daß $COLLECT = ([3,3],[3,4])$. Beachtet werden muß dabei, daß Duplikate nicht mit aufgenommen werden. Da $UQS = \emptyset$, wird nun $MV = COLLECT = ([3,3],[3,4])$. $COLLECT$ wird hier nicht wie bei ECA auf Null gesetzt.

5.3.4 Strobe–Algorithmus

Bisher sind wir davon ausgegangen, daß in unserer Warehouseumgebung nur eine Quelle vorhanden ist. Das Hauptziel beim Data–Warehousing ist jedoch die Zusammenfassung und Integration mehrerer solcher Quellen. Das bedeutet für das Warehouse, daß es als Reaktion auf eine Änderung mehrere Teilanfragen an die verschiedenen Quellen losschicken muß. Im folgenden wird anhand eines Beispiels gezeigt, daß dadurch neue Anomalien entstehen können. Danach wird dann der Strobe–Algorithmus vorgestellt, der diese Schwierigkeiten meistern kann.

Beispiel zu ECA–Key mit mehreren Quellen: Gegeben seien wieder die drei Relationen: $r_1(W,X)$, $r_2(X,Y)$, $r_3(Y,Z)$

r_1 :

W	X
2	4

r_2 :

X	Y
4	6

r_3 :

Y	Z
6	8

sowie die Sichtdefinition $V = (r_1 \bowtie r_2 \bowtie r_3)$. Zu Beginn ist $MV = [2,4,6,8] = COLLECT$. Die drei Relationen r_1, r_2, r_3 liegen nun auf drei verschiedenen Quellen A,B,C. Wir betrachten zwei Änderungen: $U_1 = insert(r_3, [6,10])$ und $U_2 = delete(r_2, [4,6])$.

Dann nehmen wir folgenden Ablauf an:

1. Das Warehouse erhält Änderung U_1 und generiert die Anfrage $Q_1 = (r_1 \bowtie r_2 \bowtie [6,10])$. Um diese Anfrage auszuwerten, muß das Warehouse sie in zwei Anfragen aufspalten, da r_1 und r_2 auf verschiedenen Quellen A,B getrennt aufbewahrt werden. Deshalb wird zuerst $Q_1^1 = (r_2 \bowtie [6,10])$ an Quelle B geschickt.

2. Das Warehouse empfängt die Antwort $A_1^1 = [4,6,10]$ von Quelle B. Jetzt wird $Q_1^2 = (r_1 \bowtie [4,6,10])$ an Quelle A geschickt.

3. Während die Antwort zu Q_1^2 noch aussteht, erhält das Warehouse $U_2 = delete(r_2, [4,6])$ von Quelle B. Mit der Operation *key–delete* wird $COLLECT = \emptyset$.

4. Das Warehouse bekommt $A_2^1 = [2,4,6,10]$ von Quelle A zurück und fügt die Antwort zu $COLLECT$ hinzu. $COLLECT = COLLECT + A_2^1 = [2,4,6,10]$. Da jetzt keine Anfrage mehr aussteht, wird $MV = COLLECT = [2,4,6,10]$.

Eigentlich müßte das korrekte Ergebnis $MV = \emptyset$ lauten, da anschaulich betrachtet, Relation r_2 nach den beiden Änderungen leer ist und demnach bei gegebener Sichtendefinition keine Tupel in der Sicht vorhanden sein können.

Strobe–Algorithmus

Wie im Beispiel gezeigt, entstehen die Probleme im verteilten Fall, wenn nachdem bei einer Quelle eine Teilanfrage evaluiert wurde, ein delete bei derselben Quelle auftritt und dieses delete vor der Gesamtantwort beim Warehouse ankommt. Deshalb sammelt

der Strobe–Algorithmus alle *delete*–Änderungen in einer Menge $pending(Q)$, die für jede Gesamtanfrage Q gehalten wird. Somit ist gewährleistet, daß die Gesamtantwort mit Hilfe der Operation *key–delete* um die Tupel bereinigt wird, die vorher beim Warehouse als *delete* ankamen.

Der Strobe–Algorithmus fordert genau wie ECA–Key auch, daß der Schlüssel jeder Relation in der Sichtendefinition vorhanden sein muß. Wir brauchen weiterhin für jede Anfrage Q in UQS (unanswered query set) eine Menge $pending(Q)$. Diese enthält alle Tupel, die gelöscht wurden, nachdem Anfrage Q vom Warehouse generiert wurde. Es hat sich im Beispiel gezeigt, daß diese gesamte Anfrage in mehrere kleinere Teilanfragen zerlegt werden muß, da die Daten auf mehrere Quellen verteilt sind. Der Algorithmus verwendet dazu eine Funktion namens *source_evaluate*, die als Argument die gesamte Anfrage erwartet, die sich aus einer Änderung ergibt. Sie schickt also eine Teilanfrage an eine Quelle, empfängt das Ergebnis wieder und integriert dieses Ergebnis in die Gesamtanfrage, schickt wieder eine Teilanfrage an eine andere Quelle, usw. Schließlich erhält sie von der letzten Quelle das Endergebnis, daß die Gesamtantwort der ursprünglichen Anfrage Q darstellt.

Bei den Quellen treten genau die gleichen Ereignisse auf wie im Basisalgorithmus.

Ereignisse beim Warehouse:

- WC (working copy) wird mit konsistentem MV initialisiert.
- Nach dem Erhalt von Änderung U_i
  - wenn Änderung U_i ein *delete* ist, dann
    - $\forall Q_j \in UQS$ füge U_i $pending(Q_j)$ hinzu;
    - $key \Leftrightarrow delete(WC, U_i)$.
  - wenn Änderung U_i ein *insert* ist, dann
    - $Q_i = V\langle U_i \rangle$ und $pending(Q_i) = \emptyset$;
    - $A_i = source\_evaluate(Q_i)$;
    - $\forall U_j \in pending(Q_i)$ mach ein $key \Leftrightarrow delete(A_i, U_j)$;
    - füge A_i WC hinzu (keine Duplikate hinzufügen).
- wenn $UQS = \emptyset$, dann $MV = WC$.

Beispiel zu Strobe–Algorithmus mit mehreren Quellen: Wir betrachten dasselbe Szenario wie im vorhergehenden Beispiel, nur daß diesmal der Strobe–Algorithmus darauf angewendet wird. Dann ergibt sich folgender Ablauf:

1. Zu Beginn ist $WC = MV = [2, 4, 6, 8]$. Das Warehouse erhält Änderung U_1 und generiert die Anfrage $Q_1 = (r_1 \bowtie r_2 \bowtie [6, 10])$. Um diese Anfrage auszuwerten, muß das Warehouse sie in zwei Anfragen aufspalten, da r_1 und r_2 auf verschiedenen Quellen A,B getrennt aufbewahrt werden. Deshalb wird zuerst $Q_1^1 = (r_2 \bowtie [6, 10])$ an Quelle B geschickt.

2. Das Warehouse empfängt die Antwort $A_1^1 = [4, 6, 10]$ von Quelle B. Jetzt wird $Q_1^2 = (r_1 \bowtie [4, 6, 10])$ an Quelle A geschickt.

3. Während die Antwort zu Q_1^2 noch aussteht, erhält das Warehouse $U_2 = delete(r_2, [4, 6])$ von Quelle B. Dann wird zuerst U_2 in $pending(Q_1)$ eingefügt und danach wird die Operation $key \Leftrightarrow delete(WC, U_2)$ angewendet. Ergebnis ist $WC = \emptyset$.

4. Das Warehouse bekommt $A_1^2 = [2, 4, 6, 10]$ von Quelle A zurück. Weil $pending(Q_1)$ nicht leer ist, macht das Warehouse ein $key \Leftrightarrow delete(A_1^2, U_2)$ und die Antwort $A_1 = \emptyset$. Da keine weiteren Anfragen mehr ausstehen, wird die Sicht des Warehouse jetzt geändert: $MV = WC = \emptyset$. Im Gegensatz zu ECA–Key ist hier die Sicht korrekt.

Durch das Aufbewahren der *delete*–Operationen in den Mengen $pending(Q_i)$ wird verhindert, daß Warehouse und Quellen in einen inkonsistenten Zustand kommen.

Neben dem soeben vorgestelltem Strobe–Algorithmus gibt es noch Erweiterungen davon, die sich mit verschiedenen Transaktionsszenarien beschäftigen. Dort werden dann nicht, so wie wir es betrachtet haben, jeweils einzelne Änderungen an das Warehouse geschickt, sondern mehrere Änderungen zu einer lokalen Transaktion der Quelle zusammengefaßt. Das Ziel ist es, diese lokale Transaktion atomar am Warehouse zu behandeln. Weiterhin werden auch Algorithmen für globale Tansaktionen untersucht, die dann beispielsweise Aktionen enthalten können, die an mehreren Quellen verteilt ablaufen sollen. Ausführlich wird die Strobe–Algorithmusfamilie in [28] beschrieben.

5.4 Aufwandsanalyse

Im folgenden wird eine Aufwandsanalyse des ECA–Algorithmus durchgeführt anhand der Menge der zu übertragenden Daten. Zum Vergleich wird dazu ein weiterer Algorithmus zur Sichterhaltung, der Recomputing the View Algorithmus (RV), kurz vorgestellt. Dieser berechnet die Sicht vollständig neu, d.h. die Antwort der Quelle auf ein Sichtänderung beinhaltet alle Tupel, die der Sichtdefinition entsprechen. Beim ECA–Algorithmus bestand die Antwort auf eine Anfrage nur aus den Tupeln, die mit der jeweiligen Änderung eine Joinverbindung eingingen.

5.4.1 View Recomputing–Algorithmus

Das Warehouse macht erst eine Sichtänderung, wenn es eine Anzahl s an Änderungen von der Quelle erhalten hat. Die Ereignisse bei der Quelle sind dieselben wie im Basisalgorithmus. Beim Warehouse passiert folgendes ($COUNT = 0$):

- $W\_up_i(U_i)$: empfange U_i;
  $COUNT = COUNT + 1$;
  if $COUNT = s$
  then $\{COUNT = 0;\ Q_i = V$;
  sende Q_i zur Quelle;
  starte Ereignis S_qu_i bei der Quelle $\}$

- $W\_ans_i(A_i)$: empfange A_i;
  $MV \leftarrow A_i$

5.4.2 Testumgebung

Da es eine Reihe von Parametern bei diesem Aufwandsvergleich gibt, wird hier ein möglichst repräsentatives Szenario ausgewählt, um den Aufwand der einzelnen Algorithmen zu veranschaulichen. Dieses Szenario sieht wie folgt aus:

Wir betrachten die Basisrelationen $r_1(W,X), r_2(X,Y), r_3(Y,Z)$.
Die Sichtdefinition lautet: $V = \Pi_{W,Z}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie r_3))$.
Änderungen: $U_1 = insert[r_1, t_1]$, $U_2 = insert[r_2, t_2]$, $U_3 = insert[r_3, t_3]$

Weiterhin gelten folgende Voraussetzungen:

1. Die Kardinalität (Anzahl der Tupel) jeder Relation ist eine Konstante C.
2. Die Größe der Attribute, auf die projiziert wird (W,Z) beträgt S Bytes.
3. Der Joinfaktor $J(r_i, a)$ ist die Anzahl der Ergebnistupel, die bei einem Join der Relation r_i über das Attribut a entstehen.
4. Die Selektivität der Bedingung *cond* ist gegeben durch, $0 \leq \sigma \leq 1$.

5.4.3 Berechnung der zu übertragenden Daten

Betrachtet werden nun im Anschluß der günstigste und der ungünstigste Fall bei beiden Algorithmen.

Recomputing the View

Am wenigsten Daten werden natürlich bei RV übertragen, wenn die Sicht erst nach allen Änderungen neu berechnet wird (B_{RVbest}, Anzahl der zu übertragenden Bytes bei RV im günstigsten Fall). Der schlechteste Fall tritt ein, falls die Sicht nach jeder Änderung neu berechnet wird. Hier multipliziert sich B_{RVbest} mit der Anzahl der Änderungen, da ja nach jeder Änderung die neue Sicht zum Warehouse geschickt werden muß, d.h. $B_{RVworst} = B_{RVbest} * Anzahl\ der\ Änderungen$. Beim RV–Algorithmus berechnen sich die zu übertragenen Daten aus der Größe der resultierenden Sicht. Also ist

$$\begin{aligned} B_{RVbest} &= (\text{Größe der Attribute}) * (\text{Anzahl der Tupel der resultierenden Sicht}) \\ &= S * \sigma * (card(r_1) * J(R_2, X) * J(r_3, Y)) \\ &= S\sigma CJ^2 \end{aligned}$$

Für den ungünstigsten Fall ergibt sich bei unserer Testumgebung (drei Änderungen):

$$B_{RVworst} = 3 * B_{RVbest} = 3S\sigma CJ^2$$

Eager Compensating Algorithmus

Bei ECA ist der einfachste Fall der, bei dem keine ausgleichende Anfragen geschickt werden müssen. Eine maximale Anzahl an ausgleichenden Anfragen wird nötig, falls alle Änderungen vor der ersten Antwort beim Warehouse ankommen. Die Ereignisreihenfolge beim ECA–Algorithmus im günstigsten Fall lautet: U_1, Q_1, A_1, U_2, Q_2, A_2, U_3, Q_3, A_3, d. h. es entstehen keine ausgleichende Anfragen. Deshalb ergeben sich drei gleichaufgebaute Anfragen. Beispielweise für $Q_1 = V\langle U_1\rangle = \Pi_{W,Z}(\sigma_{cond}(t_1 \bowtie r_2 \bowtie r_3))$. Demnach ist die Größe der Antwort A_1: $B = S * \sigma * 1 * J(r_2, X) * J(r_3, Y) = S * \sigma * J^2$. Analog geht die Berechnung für A_2, bzw. für A_3, so daß gilt:

$$B_{ECAbest} = 3S\sigma J^2$$

Nun wird der ungünstigste Fall für ECA betrachtet. Die Ereignisreihenfolge ist: U_1, Q_4, U_2, Q_5, U_3, Q_6, A_4, A_5, A_6. Daraus ergeben sich die Anfragen:

$Q_4 = V\langle U_1\rangle = Q_1$
$Q_5 = V\langle U_2\rangle \Leftrightarrow Q_4\langle U_2\rangle = V_2\langle U_2\rangle \Leftrightarrow \Pi_{W,Z}(\sigma(t_1 \bowtie_X t_2 \bowtie_Y r_3))$
$Q_6 = V\langle U_3\rangle \Leftrightarrow Q_4\langle U_3\rangle \Leftrightarrow Q_5\langle U_3\rangle = V\langle U_3\rangle \Leftrightarrow \Pi_{W,Z}(\sigma_{cond}(t_1 \bowtie_X r_2 \bowtie_Y t_3)) \Leftrightarrow \Pi_{W,Z}(\sigma_{cond}(r_1 \bowtie_X t_2 \bowtie_Y t_3)) + \Pi_{W,Z}(\sigma_{cond}(t_1 \bowtie_X t_2 \bowtie_Y t_3))$.
Für den letzten Term in Q_6 entstehen keine Übertragungskosten, da er nur aus einzelnen Tupeln besteht und damit direkt beim Warehouse ausgewertet werden kann. Man sieht, daß die ersten Terme von Q_4, Q_5, und Q_6 mit dem Aufwand für $B_{ECAbest}$ übereinstimmen. Dazu kommen im ungünstigsten Fall noch drei Terme in denen jeweils zwei Tupel und eine Relation enthalten sind. Die Kosten für einen dieser Terme belaufen sich auf $B = S\sigma J$. Somit ergibt sich insgesamt

$$B_{ECAworst} = B_{ECAbest} + 3S\sigma J = 3S\sigma J(J+1).$$

Allgemeine Abschätzung

Es stellt sich also heraus, daß der ECA–Algorithmus bei nur drei Änderungen wesentlich effizienter ist als RV, betrachtet man die Menge der zu übertragenden Daten. Die Formeln für ECA sind von der Größe der Relationen C unabhängig und die Vorteile von ECA gelten auch noch, wenn man den Joinfaktor variiert. Um eine fundierte Aussage über die Effizienz machen zu können, muß man sich allerdings den allgemeineren Fall mit k Änderungen anschauen. Es ergeben sich folgende Formeln für die vier Fälle, die in [29] ausführlich hergeleitet werden:

$B_{RVbest} = S\sigma CJ^2$
$B_{RVworst} = kS\sigma CJ^2$
$B_{ECAbest} = kS\sigma J^2$
$B_{ECAworst} = kS\sigma J^2 + k(k \Leftrightarrow 1)S\sigma J/3$

Bei einer Relationsgröße von $C = 100$ zeigt sich, daß RV_{best} ab einer Menge von 100 Änderungen besser ist als ECA_{best}. Vergleicht man ECA_{worst} mit RV_{best} so verringert

sich die Zahl der Änderungen auf 30 (bis dahin ist ECA besser). Es ist aber sehr unwahrscheinlich, daß alle Änderungen vor irgendeiner Antwort beim Warehouse eintreffen und deshalb viele ausgleichende Anfragen verwendet werden müssen. Wenn man annimmt, daß sich Änderungen und Anfragen bei der Quelle abwechseln, dann liegt die Zahl der Änderungen, bei der ECA_{worst} besser abschneidet als RV_{best}, zwischen 30 und 100. Bei größeren Relationen (≥ 100) wird sich diese Zahl nochmals vergrößern. Und man muß auch in Betracht ziehen, daß bei RV_{best} nur einmal die Sicht geändert wird, was keine den Relationen entsprechende aktuelle Sicht garantiert. Wenn RV häufiger seine Sicht ändert, wird der Algorithmus sehr ineffizient. RV_{worst} ist weitaus schlechter als ECA_{worst}.

Kapitel 6

Materialisierung

Holger Schätzle

Kurzfassung *Die Datenbestände relationaler Datenbanken sind oft unübersichtlich, da sie umfangreich und auf viele Relationen verstreut sein können. Hier bietet OLAP (On-line analytic processing) eine Methode, Daten besser zu verstehen und miteinander zu verknüpfen. Leider sind Benuteranfragen an ein solches System oft sehr komplex und es kann Stunden dauern, bis das System antwortet, wenn solche Anfragen an die Original-Datenmenge gestellt wird. Um diese Wartezeit zu verringern, werden bestimmte Sichten und darauf basierende Indizes vorberechnet. In den folgenden Abschnitten wird eine Strategie vorgestellt, um eine geignete Menge von Sichten und Indexen zu finden.*

6.1 Einleitung

Ein Data-Warehouse sammelt Informationen von vielen verschiedenen Quellen in einer Datenbank. Diese Integration ermöglicht es, komplizierte Anfragen über mehrere Datenbanken hinweg auf einfache und übersichtliche Art zu stellen, ohne dabei auf die Belange der verschiedenen Quellen achten zu müssen. Dies versetzt ein Unternehmen in die Lage geschäftliche Zusammenhänge besser zu verstehen, um daraus Nutzen zu ziehen. Im Unterschied zu der konventionellen OLTP (On-line Transaction Processing) Anwendung, geschieht die Auswertung interaktiv und ist unter der Bezeichnung OLAP (On-Line Analytic Processing) bekannt.

6.1.1 Einführung in OLAP

Als Beispiel betrachte man ein Data-Warehouse von einem Unternehmen mit den Attributen `Teil`, `Lieferant`, `Kunde` und `Umsatz`. Das Unternehmen kauft ein `Teil` von einem `Lieferanten` und verkauft es an einen `Kunden`. Der Anwender eines solchen Data-Warehouse, wünscht sich natürlich die Daten aus verschiedenen Blickwinkeln betrachten zu können, d.h. typische Anfragen an das Data-Warehouse könnten sein:

- Wie hoch ist der `Umsatz` von jedem `Teil`?
- Wie hoch ist der `Umsatz` von jedem `Kunden`?
- ...

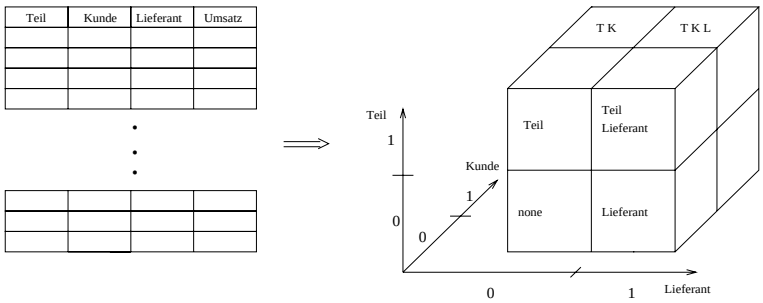


Abbildung 6.1: Relation ⇒ Datenwürfel

Dies bedeutet, daß ein OLAP-System dem Benutzer eine mehrdimensionale Sicht auf die Daten ermöglichen muß. Die verschiedenen Blickwinkel lassen sich systematisch in Form eines *Würfels* darstellen, den Abb. 6.1 für obiges Beispiel zeigt. Die Attribute stellen dabei die Dimension der Achse dar, die Achsen selbst sind in zwei Teile unterteilt. Eine 0 bedeutet, daß das Attribut für die Sicht nicht relevant ist, eine 1 die Relevanz des Attributes. Jede Zelle des Datenwürfels bezieht sich auf eine spezielle Sicht, wobei eine Sicht im Prinzip nichts anderes ist, als das Ergebnis einer Datenbankanfrage. Der Datenwürfel stellt also eine Art logische Sicht auf die zugrundeliegende Tabelle dar[1]. Die Anfrage: Wie hoch ist der Umsatz von jedem Teil wird dabei durch den linken oberen Würfel repräsentiert.

Um einen interaktiven Betrieb mit den vielen Daten zu ermöglichen, müssen die daten der entsprechende *Zelle* schnell zu ermitteln sein. Daraus ergibt sich für einen Warehouse Administrator (WHA) die Frage ob die Anfragen an die verschiedenen Datenbanken direkt gestellt werden sollen, was sich enorm zeitintensiv gestaltet, oder ob all die verschiedenen Sichten im Warehouse vorberechnet werden sollen, was allerdings einen enormen Speicherbedarf im Warehouse erfordert. Hierfür gibt es zwei verschiedene Strategien:

- MD-OLAP (multidimensional OLAP): Bei dieser Strategie werden alle möglichen Sichten im Warehouse gespeichert, was eine schnelle Antwortzeit ermöglicht, doch eine lange Vorberechnung mit sich bringt und zusätzlich enorme Speicheranforderungen an das Warehouse stellt.
- ROLAP (relational OLAP): Diese Strategie dagegen berechnet nur wenige Sichten voraus. Für bestimmte Anfragen an das Warehouse, müssen dann die Quelldaten aus den verschiedenen Datenbanken herangezogen werden, um die Anfrage zu beantworten. Diese Möglichkeit spart Vorberechnungszeit und Speicherplatz im Warehouse, verlangsamt auf der anderen Seite aber die Antwortzeit. Jedoch wird sich später zeigen, daß durch geschickte Wahl der Sichten und insbesondere der Verwendung von Indexen, sich die Antwortzeit enorm reduzieren läßt.

[1] Die Auswahl der Zellen im obigen Beispiel, werden in Abschnitt 6.2.1 erklärt

Die MD-OLAP Methode ist nur bei sehr kleinen Datenbeständen anwendbar, da die Vorberechnungszeit exponentiell mit der Anzahl der zugrundeliegenden Attributen anwächst (was natürlich auch für den Speicherbedarf im Warehouse gilt)[2]. Wir werden daher im folgenden die zweite Methode ROLAP betrachten.

Die Vorberechnung bei einem ROLAP System wird üblicherweise in einem zweischrittigen Prozeß durchgeführt. Im ersten Schritt werden die gewünschten Sichten berechnet. Im zweiten Schritt, werden die erforderlichen Indexe für die Sichten erzeugt.Dabei gehen wir davon aus, daß die Speicherkapazität beschränkt ist. Das Problem ist daher den Verfügbaren Speicherplatz zwischen diesen zwei Schritten aufzuteilen, was sich nicht einfach gestaltet. Verwendet man zuviel Speicher für die Indexe, bleibt zuwenig Platz, um ein ausreichendes Maß an Sichten anzulegen, verwendet man hingegen zuviel Platz für die Sichten ist nicht mehr genug Speicher für Indexe frei. In beiden Fällen kann dies zu sehr hohen Leistungseinbußen führen, d.h. die Antwortzeiten können sich drastisch erhöhen.

Eine effizientere Methode ist es, die Suche der Sichten und die Berechnung der passenden Indexe in einem Prozeßschritt durchzuführen. Diese Methode führt zu einer effizienten Ausnützung des gegebenen Speichers. Im folgenden werden wir einen Algorithmus vorstellen, der nach diesem Prinzip arbeitet [30].

6.2 Grundbegriffe und Definiton

6.2.1 Sichten

Die Auswahl der Sichten und somit die Materialisierung des Würfels, kann auf vielfältige Weise geschehen. Beispielsweise könnte jeden einzelnen Zelle des Würfels materialisiert werden oder aber nur eine Sicht welche die gesamte Tabelle beinhalten würde. Doch sind beide Extreme keine gute Wahl, denn die erste Auswahl würde zu einer unüberschaubaren Menge von Zellen führen und die zweite Aufteilung ist ja nichts anderes, als ein Abbild der Quelle, also eine unnötige Kopie der Daten.

Jede Zelle (Sicht) entspricht in SQL-Notation einer **group by** Anfrage. Allgemein ergeben sich be k Attributen 2^k Sichten, welche im Beispiel folgende sind:

1. `Teil, Lieferant, Kunde` (6M, d.h. 6 Millionen Datensätze)
2. `Teil, Kunde` (6M)
3. `Teil, Lieferant` (0.8M)
4. `Lieferant, Kunde` (6M)
5. `Teil` (0.2M)
6. `Lieferant` (0.01M)

[2]Da der Speicherbedarf und die Vorberechnungszeit zusammenhängen, wird im weiteren nur eines der beiden Worte erwähnt

7. `Kunde` (0.1M)
8. `none` (1)

`none` bezeichnet die leere Menge, d.h. die Anfrage bei der keine Attribute in der **group by** Klausel stehen. Die Sicht Teil-Kunde als Beispiel ist also eine Tabelle die durch folgende SQL-Anfrage entsteht:

```
SELECT Teil, Kunde, SUM(Umsatz) AS Gesamtumsatz
FROM R
GROUP BY Teil, Kunde;
```

Da wir dieses Szenario als durgehendes Beispiel verwenden, wollen wir aus Vereinfachungsgründen die Attribute Teil mit T, Lieferant mit L und Kunde mit K abkürzen.

6.2.2 Anfragen

Um Anfragen etwas kürzer zu formulieren werden wir eine an die Relationenalgebra angelehnte Schreibweise verwenden. So wird die SQL-Anfrage Q:

```
SELECT Kunde, SUM(Umsatz) AS Gesamtumsatz
FROM R
WHERE Teil=Konstante
GROUP BY Kunde;
```

folgendermaßen abgekürzt: $\gamma_K \sigma_T$. D.h. γ bestimmt die Attribute der **group by** Klausel während σ die Selektionsattribute der Anfrage bestimmt. Der Umsatz als Ausgabe wird nicht extra erwähnt, da dieser immer abgefragt wird.Die obige Anfrage „schneidet“ ein Stück aus der Sicht `Teil-Kunde`. Deßhalb nennt man diese Art einer Anfrage eine *Slice Query*. Man kann solche Anfragen der Art $\gamma_{G_1,\ldots,G_k}\sigma_{S_1,\ldots,S_l}$ mit Hilfe der Sicht $G_1,\ldots,G_k$, $S_1,\ldots,S_l$ entsprechend beantworten, wobei diese Sicht die kleinstmögliche ist, die diese Anfrage beantworten kann. Eine Anfrage, die als Ergebnis eine ganze Sicht hat ist dabei eine spezielle Art einer Slice Query, deren Menge von Selektionsattributen (where Klausel) leer ist, d.h. alle Anfragen Slice Queries. Eine r-dimensionale Sicht (Anzahl der Attribute = r) kann somit von 2^r Slice Queries „abgefragt“ werden. Ein n-dimensionaler Würfel hat $\binom{n}{r}$ r-dimensionale Sichten. Somit errechnet sich die gesamte Anzahl der Slice Queries die an einen n-dimensionalen Würfel gestellt werden können durch $\sum_{r=0}^{n}\binom{n}{r}2^r$, was beinahe 3^n entspricht.

Als Kosten für eine Anfrage, betrachten wir die Anzahl der Datensätze, auf die zugegriffen werden muß.

Die Anfrage Q_1: $\gamma_T\sigma_L$, kann beispielsweise mit der Sicht TL mit den Kosten von 0.8M (Datensätzen) beantwortet werden oder mit der Sicht TLK mit den Kosten von 6M (Datensätzen).

6.2.3 Indexe

B-Baum Indexe können die Antwortzeit einer Anfrage enorm beschleunigen. Man kann verschiedene Indexe für eine Sicht erstellen so gibt es z.B. für die Sicht TL die vier Indexe

$$I_T(TL), I_L(TL), I_{TL}(TL), I_{LT}(TL)$$

In der Klammer steht auf welcher Sicht der Index erstellt wurde. Die Attribute nach denen der Index-Baum sortiert ist, sind dem *Index* zu entnehmen, wobei die Reihenfolge der Attribute eine wichtige Rolle spielt. Hat man z.B. den Index I_{TL}, so unterstützt (d.h. die Antwortzeit verkürzt sich) dieser sowohl eine Anfrage, bei der nach einem bestimmten Wert von Tgesucht wird, als auch eine Anfrage bei der nach T und L gesucht wird. Hingegen müssen alle Blätter von dem Index-Baum durchsucht werden, wenn nach einem bestimmten Wert für L selektiert wird. Dies bedeutet im allgemeinen, daß ein Index $I_{X_1,\ldots,X_k}$ nur dann sinnvoll ist, wenn in der Selektionsklausel der Anfrage ein Präfix $X_1, \ldots, X_k$ auftritt.

Die mögliche Zahl der Indexe für eine gegebene Sicht mit m Attributen errechnet sich durch $\sum_{r=1}^{m} \binom{m}{r} r!$ was für große m durch (e - 1)m! genähert werden kann. Somit errechnet sich die Gesamtzahl der Indexe für einen n-dimensionalen Würfel näherungsweise mit $(e \Leftrightarrow 1)^2 n! \approx 3n!$ ist. Betrachtet man nur sogenannte *fette Indexe*[3] , so ist die mögliche Zahl der fetten Indexe in einem n-dimensionalen Datenwürfel $(e \Leftrightarrow 1)n!$, bzw. näherungsweise $2n!$.

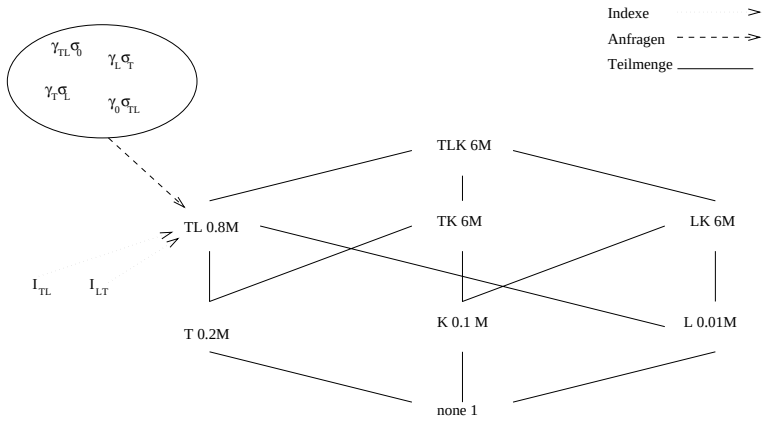


Abbildung 6.2: Sichten, Anfragen und Indexe einer Datenbank

6.2.4 Zusammenfassung

Abbildung 6.2 zeigt nochmals die verschiedenen Sichten des Würfels in einer etwas anderen Form. In dieser Darstellung wird die Teilmengenbeziehung der verschiedenen Zellen deutlich. Zusammenfassend gilt also für unsere Aufteilung des Datenwürfels. Ein n-dimensionaler Würfel hat:

[3] Ein fetter Index ist ein Index, der kein Präfix eines anderen Indexes ist. So ist z.B. $I_T(TL)$ kein fetter Index, da das Attribut T in dem Index $I_{TL}(TL)$ als Präfix enthalen ist.

- 2^n Sichten
- 3^n Slice Queries und
- ca. $3n!$ mögliche Indexe, davon sind $2n!$ fette Indexe

6.2.5 Sichten- und Indexsuche im Zwei-Schritt-Verfahren

Zurück zum eigentlichen Problem: Welche der vielen Sichten und Indexe soll vorberechnet werden, um eine schnellstmögliche Antwortzeit zu erhalten? Zur Vereinfachung sei angenommen, daß alle möglichen Anfragen gleichwahrscheinlich sind. Um alle Sichten und Indexe zu materialisieren, d.h. zu berechnen und im Data Warehouse abzuspeichern, wäre ein Platz von ca. 80M Datensätzen notwendig. In den meisten praktischen Fällen ist dieser nicht vorhanden und so daß nicht alles vorberechnet werden kann. In diesem Beispiel sei angenommen, daß ein Speicherplatz für rund 25M Datensätzen zur Verfügung steht.

Die im ersten Abschnitt erwähnte Zweischritt-Strategie, teilt den vorhandenen Speicher für die Sichten und für die Indexe. Im ersten Schritt werden die Sichten, die materialisiert werden sollen bestimmt und danach die für notwendig gehaltenen Indexe. Indexe können natürlich nur für vorher gewählte Sichten erzeugt werden. Ein großes Problem ist hierbei die Einteilung des Speichers, d.h. man muß vor der Wahl der Sichten wissen, wieviel Platz jeweils für Sichten bzw. Indexe zur Verfügung gestellt werden soll. In diesem Beispiel wird der Speicher halbiert, so daß den Sichten eine Hälfte und den Indexe die andere Hälfte zur Verfügung steht. Es wird ein Greedy-Algorithmus verwendet, auf den nicht näher eingegangen wird, der im ersten Schritt folgende Sichten findet:

TLK, TL, K, L, T, none, LK

und im folgenden Schritt die Indexe:

$$I_{KLT}, I_{TKL}$$

Diese Auswahl führt zu durchschnittlichen Anfragekosten von 1.18M Datensätzen.

Wir werden später einen Greedy-Algorithmus vorstellen, der die beiden Schritte vereint und somit folgende Ergebnis liefert:

$$TLK, I_{KLT}, TL, I_{TKL}, I_{LTK}, K, L, T, none$$

Diese Auswahl kommt auf durchschnittliche Anfragekosten von nur 0.74M Datensätzen, was im Gegensatz zur obigen Methode eine Ersparnis von fast 40 Prozent ausmacht. Der Grund der Ersparnis ist, daß der Zweischrittalgorithmus den verfügbaren Speicherplatz halbiert, was sich aber später als Fehler herausstellt, da es in diesem Fall besser ist nur ein viertel des Platzes den Sichten und drei viertel den Indexen zur Verfügung zu stellen. Wieviel Platz man für die Indexe nehmen sollte, hängt von mehreren Faktoren ab wie z. B. der Größe der bereits gewählten Sichten und Indexe und ist vorab schwer zu bestimmen.

Interessant ist noch, daß die zusätzliche Materialisierung der nichtgewählten Sichten und Indexe, die einen Speicherplatz von rund 55M Datensätzen benötigen, keinen großen Einfluß auf die Antwortzeit hätte.

6.3 Kostenmodell

In diesem Abschnitt wird ein Formel vorgestellt, um den Kostenaufwand einer Anfrage zu berechnen. Danach werden wir Methoden vorstellen, mit Hilfe derer die Größe einer Sicht bzw. eines Indexes abgeschätzt werden kannn, ohne diese berechnen zu müssen.

Der im nächsten Abschnitt folgende Algorithmus benötigt folgende Informationen:

1. Für jedes (Anfrage, Sicht, Index) Tripel, die Kosten, die Anfrage mit der gegeben Sicht und dem gegebenen Index zu beantworten.
2. Die Größe von jeder Sicht
3. Die Größe von jedem Index

In Abschnitt 6.3.1 wird gezeigt, wie Punkt 1 mit gegebenen Punkten 2 und 3 berechnet werden kann.

6.3.1 Berechnung der Kosten

Es geht darum, eine Anfrage Q durch eine Sicht S und einen Index J zu beantworten.

- Sei Q eine Anfrage $\gamma_{\bar{A}}\sigma_{\bar{B}}$, wobei $\bar{A}$[4] und $\bar{B}$ Mengen von Attributen sind. Wenn $\bar{B} = \emptyset$, so handelt es sich um eine Anfrage über die gesamte Sicht.
- Sei $\bar{C}_S$ die Menge der Attibute der Sicht S. Kann Q durch S beantwortet werden, so muß $\bar{A} \cup \bar{B} \subseteq \bar{C}_S$ sein. Bei der Sicht **none** gilt dann: $\bar{C}_{none} = \emptyset$
- Sei J der Index $I_{\vec{D}}(S)$. $\vec{D}$ soll hervorheben, daß die Reihenfolge der Attribute hier eine Rolle spielt; $\vec{D}$ ist also keine Menge, sondern eine Folge. $\vec{D} = \langle\rangle$ (die leere Folge) bezeichnet somit den Fall, bei dem kein Index benutzt wird.

Sei $\bar{E}$ die größte Untermenge von $\bar{B}$, so daß die Attribute in $\bar{E}$ einen Präfix von $\vec{D}$ bilden sowie T die Sicht für die gilt $\bar{C}_T = \bar{E}$. Die Kosten um Q mit S in Verbindung mit J zu beantworten errechnen sich durch:

$$c(Q, S, J) = \frac{|S|}{|T|}$$

Betrachten wir noch einmal das Beispiel; die Sicht S = TLK mit einer Größe von 6M Datensätzen, die Anfrage $Q = \gamma_K \sigma_{TL}$ und den Index $J = I_{LKT}$. In diesem Fall ist $\bar{C} = TLK$ und $\bar{E} = s$, wobei die größte Untermenge der Attribute TL die einen Präfix in LKT bilden, s ist. Aus dem Beispiel ist bekannt, daß $|TLK| = 6M$ und $|L| = 0.01M$ ist. Damit sind die Kosten:

$$c(Q, S, J) = \frac{6M}{0.01M} = 600$$

[4] $\bar{X}$ soll bedeuten, daß X eine Menge ist

Die obige Formel funktioniert in allen Fällen. Auch in dem Fall wenn $\bar{E} = \emptyset$. Dieser Fall trifft ein, wenn es keinen bzw. keinen passenden Index für S gibt, oder weil Q eine Abfrage einer gesamten Sicht ist. In diesen Fällen müssen alle Datensätze der Sicht durchsuchtwerden, also $c(Q, S, J) = |S|$. Doch auch hier gibt die Formel die richtige Antwort, denn $|\emptyset| = 1$.

6.3.2 Abschätzung der Sichtgröße

Es gibt viele Möglichkeiten die Größe der Sichten zu bestimmen, ohne sie berechnen zu müssen. Dazu zählen analytische Methoden mit denen man die Größe der einzelnen Sichten bestimmen kann, indem man nur die größte Sicht (d.h. die Sicht, die alle Attribute enthält) berechnet. Die Größe einer Sicht ist dann in unserem Beispiel die Anzahl der verschiedenen Werte der Attribute, nach denen gruppiert worden ist.

6.3.3 Abschätzung der Indexgröße

Ist die Größe der Sicht bekannt, kann man die Indexgröße bestimmen. Für Indexe gilt ein ähnliches Kostenmodell wie bei den Sichten. Die Größe von jedem Index (B-Baum) ist die Anzahl der Knoten und Blätter. Diese Anzahl ist jedoch annähernd gleichgroß wie die Anzahl der Datensätze des unterliegenden Indexes. Damit folgt, daß

- die Größe eines Index auf einer Sicht S ist gleich der Größe der Sicht S.

Diese Erkenntnis hat eine wichtige Konsequenz. Betrachtet man zwei Indexe $J_1 = I_{\vec{A}}(S)$ und $J_2 = I_{\vec{B}}(S)$ auf derselben Sicht S. Ist $\vec{B}$ ein Präfix von $\vec{A}$, dann gilt sicherlich $c(Q, S, J_1) \leq c(Q, S, J_2)$ für eine beliebige Anfrage Q, nach dem Kostenmodell aus Abschnitt 6.3.1, weiter sind die Speicheranforderungen von J_1 und J_2 fast gleich. Daraus folgt, daß man den Index J_2 weglassen kann, wenn man den Index J_1 berechnet hat. Allgemein heißt dies, man braucht nur sogenannte fette Indexe[5] Sei S die Sicht mit der Attributmenge $\bar{C}_S$, so ist die Menge der Indexe gegeben durch:

$$\{I_{\vec{D}}(V) | \vec{D} \quad \text{ist ein Permutation von} \quad \bar{C}\}$$

Man kann zeigen, daß diese Beobachtung zu einer Reduzierung der Indexe um einen Faktor von nahezu $e \Leftrightarrow 1$ beiträgt, wobei e die eulersche Zahl ist.

6.4 Materialiserung von Sichten und Indexen

In diesem Abschnitt wird ein Algorithmus vorgestellt, der Sichten und Indexe aussucht, die nicht mehr als eine gegeben Größe an Speicher S belegen, um damit eine gegebene Menge von Anfragen zu beantworten. Die Suche nach der optimalen Lösung ist NP-vollständig, doch weicht der vorgestellte Algorithmus nur wenig von einer optimalen Lösung ab.

[5] Zur Erinnerung: Ein fetter Index ist ein Index, dessen Attribute keinen Präfix eines andreren Indexes derselben Sicht bilden.

6.4.1 Problemdefinition

Betrachte einen bipartiten Graph, $G = (S \cup Q, E)$, ein sogenannter *Anfrage-Sichten-Graph* (siehe Abb. 6.3). S sei die Menge der Sichten und Q die Menge der Anfragen.

- Zu jeder Sicht $s_i \in S$ gehört ein Tupel (P_i, I_i) (das in Abb. 6.3 nicht angegeben ist, da pro Index bzw. Sicht ein Platzbedarf von 1 angenommen wird), wobei

  P_i den Speicherbedarf (Platz) angibt, der durch die Sicht verbraucht wird, und

  I_i ist die Menge der Indexe auf der Sicht. I_{ik} bezeichnet den k-ten Index von s_i.

- Mit jeder Anfrage $q_i \in Q$ sind die Kosten T_i verbunden. T_i bezeichnet die Kosten eine Anfrage q_i zu beantworten, ohne daß dabei eine spezielle Sicht oder ein Index verwendet wird (In Abb. 6.3 wird T_i als konstant = 100 angenommen).

- Jede Kante (q_i, s_j) hat eine Markierung (k, t_{ijk}), wobei t_{ijk} die Kosten sind, die Anfrage q_i mit der Sicht s_j und deren k-ten Index zu beantworten. Wenn $k = 0$, sind t_{ijk} die Kosten, um q_i nur mit der Sicht s_i zu beantworten.

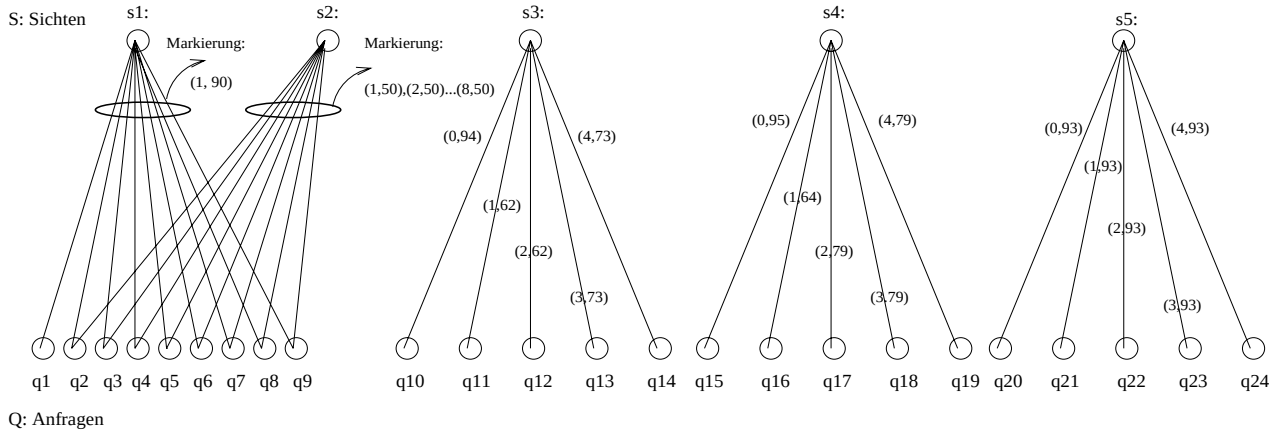


Abbildung 6.3: Ein Anfrage-Sichten-Graph

In Abb. 6.3 beispielsweise besitzt die Sicht 1 den Index I_{11}, die Sicht 4 die Indexe $I_{41}, \ldots, I_{44}$. Das Ziel ist es nun, mit einer gegebenen Menge von Sichten S und einer Menge von Anfragen Q, eine Menge $M \subseteq S$ zu finden, die Sichten und Indexe beinhaltet, die die gegebenen Anfragen Q in schnellstmöglicher Zeit beantworten, wobei M einen gegebenen Speicherplatz P nicht überschreiten darf. Formal bedeutet dies:

$$\tau(G, M) = \sum_{i=1}^{|Q|} \mathbf{min}(T_i, \min_{s_i, I_{j,k} \in M} t_{ijk})$$

6.4.2 Gewinn

Sei C eine beliebige Menge von Sichten und Indexen in einem Anfrage-Sichten-Graph G. $P(C)$ bezeichnet dann den gesamten Speicherplatz, der von den Stukturen[6] in C belegt

[6] Eine *Struktur* ist ein Sicht oder ein Index

wird. Der *Gewinn* durch C unter Berücksichtigung von M wird durch $B(C, M)$ bezeichnet und wird definiert durch $(\tau(G, M) \Leftrightarrow \tau(G, M \cup C))$, wobei τ die oben definierte Funktion ist. $B(C, \emptyset)$ bezeichnet den *absoluten Gewinn* der Menge C. Gewinn von C pro Speichereinheit mit Berücksichtigung von M ist $B(C, M)/P(C)$.

6.4.3 Der r-Greedy–Algorithmus

Der r-Greedy Algorithmus beinhaltet mehrere Schleifendurchläufe, wobei er während jedem Durchlauf eine Teilmenge C bestimmt, die maximal r Strukturen beinhaltet. Diese Teilmenge besteht entweder aus

1. Einer Sicht und einigen Indexen der Sicht, oder
2. Einem einzelnen Index, dessen Sicht in einer vorherigen Phase bereits selektiert wurde.

Angenommen es gibt s Sichten und jede Sicht hat höchstens i Indexe. So muß der r-Greedy-Algorithmus in jedem Durchlauf den Gewinn von maximal $si + s\binom{i}{r-1}$ möglichen Mengen berechnen. Somit ergibt sich eine Komplexität des Algorithmus von $O(km^r)$, wobei m die Anzahl der Strukturen im gegebenen Anfrage-Sichten-Graph ist und k die Anzahl der vom Algorithmus ausgewählten Strukturen, die höchstens P_{max} beträgt.

r-Greedy Algorithmus

Gegeben: G, ein Anfrage-Sichten-Graph und P_{max} der verfügbare Speicher
BEGIN
  $M = \emptyset$ /* Anzahl der bereits gesammelten Strukturen */
  while $(P(M) < P_{max})$
    Durchsuche alle möglichen Mengen von Sichten und Indexen der folgenden Form:
    - $\{s_i, I_{ij_1}, I_{ij_2}, \ldots, I_{ij_p}\}$, so daß $s_i \notin M, I_{ij_l} \notin M$, $1 \leq l \leq p$, und $0 \leq p < r$, oder
    - $\{I_{ij}\}$, so daß die Sicht s_i des Index I_{ij} in M ist und $I_{ij} \notin M$.

    Unter diesen Mengen soll C die Menge sein, welche am meisten Nutzen pro Speichereinheit hat, d.h. $\max(B(C, M)/P(C))$.
    $M = M \cup C$;
  end while;
  return M;
END.

Funktionsweise des Algorithmus

Als Beispiel verwenden wir den Anfrage-Sichten-Graph in Abb. 6.3. Zur Vereinfachung, sei der Speicherbedarf pro Index bzw. pro Sicht eine Speichereinheit. Die Kosten eine Anfrage ohne Benutzung einer Sicht oder eines Indexes betragen 100 Speichereinheiten, d.h es

müssen also 100 Speichereinheiten durchlaufen werden um die Anfrag zu beantworten. Der verfügbare Speicher sei auf $P_{max} = 7$ beschränkt. Die in Klammer angegebenen Werte beziehen sich auf die Kosten einer Sicht ((0, Kosten)) oder eines Index ((x, Kosten) $x = 1, \ldots, n$).

Im folgenden wird aufgezeigt, wie der r-Greedy Algorithmus für verschiedene Werte von r arbeitet.

1-Greedy: Im ersten Schritt werden alle Sichten der Reihe nach auf ihren Gewinn hin untersucht. So kann die Anfrage q_{10} beispielsweise durch die Sicht s3 mit Kosten von 94 beantwortet werden, was einen Gewinn von $B(s3, \emptyset) = 100 \Leftrightarrow 94 = 6$ ergibt. Den meisten Nutzen bringt jedoch die Sicht s5 mit einem Gewinn von 7 ($100 \Leftrightarrow 93 = 7$). Somit sucht sich der Algorithmus also die Sicht S_5 aus.

Nun werden zu der Menge M der Reihe nach die Indexe $I_{5,1}, I_{5,2}, I_{5,3}, I_{5,4}$ hinzugefügt.

Da $P(M) = 5 < P_{max}$, wird jetzt im zweiten Schritt die Sicht S_3 ausgewählt. Da diese von den noch nicht gewählten Sichten am meisten Nutzen (6) hat.

Nun wird noch der Index $I_{3,1}$ hinzugefügt und so gibt der 1-Greedy-Algorithmus als Lösung die Menge $M = \{S_5, I_{5,1}, I_{5,2}, I_{5,3}, I_{5,4}, S_3, I_{3,1}\}$, mit einem Gewinn von 79, aus.

2-Greedy: Im ersten Schritt selektiert der Algorithmus $C = \{S_1, I_{1,1}\}$, mit einem Gewinn von 10 x 9 = 90. Der Gewinn von $\{S_2, I_{2,1}\}$ währe 40 mit Berücksichtigung von C (d. h. 20 pro Speichereinheit). Jedoch wird im zweiten Schritt $\{S_4, I_{4,1}\}$ gewählt, da der Gewinn 41 (also 20,5 pro Speichereinheit) beträgt. Im nächsten Schritt werden die Indexe von S_4 selektiert, was zu einer Lösung von $M = \{S_1, I_{1,1}, S_4, I_{4,1}, I_{4,2}, I_{4,3}, I_{4,4}\}$ mit einem Gesamtgewinn von 194.

3-Greedy: Sowie bei 2-Greedy, wird zuert $C = \{S_1, I_{1,1}\}$ mit einem Gewinn von 90 ausgewählt. Im zweiten Schritt wählt der Algorithmus die Menge $\{S_3, I_{3,1}, I_{3,2}\}$ mit einem Gewinn von 82, also 27.3 pro Speichereinheit. Später werden die Indexe $I_{3,3}$ und $I_{3,4}$ hinzugefügt, was zu der Menge $M = \{S_1, I_{1,1}, S_3, I_{3,1}, I_{3,2}, I_{3,3}, I_{3,4}\}$ mit einem Gesamtgewinn von 226 führt.

Optimale Lösung: Es ist nicht schwer zu erkennen, daß die optimale Lösung für das gegebene Beispiel $M = \{S_2, I_{2,1}, I_{2,2}, I_{2,3}, I_{2,4}, I_{2,5}, I_{2,6}\}$ mit einem Gesamtgewinn von 300 ist.

Der vorgegebene Algorithmus garantiert, daß der Gewinn von M mindestens ($1 \Leftrightarrow 1/e^{(r-1)/r}$) mal dem Gewinn der optimalen Lösung ist, die mit demselben Speicherplatz auskommt.

Kapitel 7

Sichtenevolution

Jürgen Enge

Kurzfassung *Bei Sichten, die in Relationen materialisiert sind, ist es bei einer Änderung der Sicht oft aufwendig, die komplette Relation neu zu erstellen. Dieser Text stellt Möglichkeiten zur Anpassung von materialisierten Sichten vor.*

7.1 Einführung

Anwendungen, die über Sichten auf große Datenbestände zugreifen, haben oft das Problem, daß der Zugriff auf die Daten über die Sicht direkt nicht schnell genug (komplexe Sicht), oder aus anderen Gründen nicht möglich ist. Dieses Problem wird durch eine Materialisierung der Sicht, das heißt, daß alle Daten, die in der Sicht zu finden sind, in eine Relation eingefügt werden, gelöst. Auf diese Art und Weise greift die Anwendung dann nicht mehr auf die Sicht (**VIEW**), sondern auf deren Materialisierung zu.
Bei Änderungen in der Sicht ist es dann oft aufwendiger, die neue Sicht komplett zu materialisieren, als die alte Sicht anzupassen (`Adaption, Evolution`).
Anwendungen, in denen materialisierte Sichten verwendet werden, sind zum Beispiel Visualisierungsprogramme, die bei einer Sichtänderung schnell auf die neuen Daten zugreifen können müssen, um interaktiv zu bleiben, oder aber Anwendungen zur Datenarchäologie, in denen der Archäologe versucht, Regeln über die Daten zu finden, indem er Abfragen formuliert, die Ergebnisse betrachtet und dann, mit wachsendem Verständnis der Daten, die Abfragen ändert.

7.1.1 Beispiel zur Motivation

Alle vorkommenden Beispiele werden sich auf die folgenden Relationen A(rbeiter), Z(eiten) und B(austelle) beziehen:

- A(*Ang#*, Name, Adresse, Alter, Lohn)

- Z(*Ang#*, *Bau#*, Stunden)
- B(*Bau#*, BauName, Leiter#, Ort, Gesamtaufwand)

Der Schlüssel jeder Relation ist unterstrichen. Angenommen, eine Anwendung will Daten visualisieren, die es aus den Relationen mittels SELECT, FROM, WHERE, GROUPBY und anderen SQL-Konstrukten erhält.
Zum Beispiel könnte die folgende Sicht von der Abfrage A_1 definiert werden.

CREATE VIEW V **AS**
SELECT $\underline{Ang\#}$, $\underline{Bau\#}$, $Lohn$
FROM $A \bowtie Z$
WHERE $Lohn > 10000$ **AND** $Stunden > 40$

Aus Gründen einfacherer Darstellung wird die natürliche Verbindung zwischen den Relationen A und Z auf dem Attribut $\underline{Ang\#}$ im **FROM**-Ausdruck durch das '$\bowtie$'-Zeichen dargestellt.
Die Abfrage A_1 könnte zum Beispiel von einer Anwendung generiert werden, die die Daten grafisch in einer Tabelle darstellt und einen Schieberegler für den *Lohn* und einen für die *Stunden* besitzt. Werden nun die Regler verschoben, so muß möglichst schnell die Tabelle aktualisiert werden.
Wird der Regler für den *Lohn* auf $Lohn > 15000$ eingestellt, so kann die Antwort auf diese neue Abfrage schnell aus der Antwort der alten Abfrage berechnet werden. Es müssen lediglich alle Tupel, in denen der *Lohn* größer 10000 und kleiner 15000 ist, entfernt werden. Wenn aber der Regler auf > 5000 geschoben wird, so ist der Vorgang nicht mehr ganz so einfach. Die Alten Daten können jedoch weiterhin angezeigt werden, es müssen aber neue Tupel, nämlich die, bei denen der *Lohn* größer 5000 und kleiner 10000 ist, hinzugefügt werden. Obwohl die neue Abfrage nicht mit den Daten der Alten beantwortet werden kann, ist es doch möglich den Aufwand der Neuberechnung stark zu reduzieren, indem die alten Daten weiterverwendet werden.
Will der Benutzer die Abfrage A_1 dahingehend ändern, daß er sie mit der Relation B verbindet und eine Aggregierung berechnet, so ist die Neuberechnung der Sicht nicht mehr so einfach.

CREATE VIEW V **AS**
SELECT $\underline{Bau\#}$, Ort, **SUM**($Lohn$)
FROM $A \bowtie Z \bowtie B$
WHERE $Lohn > 10000$ **AND** $Stunden > 40$
GROUPBY $\underline{Bau\#}$, Ort

In A_2 wird also A_1 mit B über das Attribut $\underline{Bau\#}$ verbunden und die resultierende Sicht über die Attribute $\underline{Bau\#}$ und *Ort* gruppiert. Da nun der Schlüssel der Relation B, $\underline{Bau\#}$, schon im Ergebnis der Abfrage A_1 vorkommt, muß zur Berechnung von A_2 nur noch das Attribut *Ort* aus B geholt werden und in die materialisierte Sicht eingetragen werden. Will man verhindern, daß A_2 getrennt von A_1 materialisiert werden muß, so kann man in A_1 schon im voraus Platz für zukünftige Attribute lassen.$\diamond$

Der Schwerpunkt wird nun auf Änderungen von materialisierten Sichten und der Neuberechnung der Materialisierung unter Verwendung der alten Sichten liegen.

7.1.2 Ergebnis

Der Prozess, eine materialisierte Sicht neu zu definieren, wird als eine Folge von lokalen Änderungen der Sicht definiert. Die Adaption wird als eine zusätzliche Abfrage oder eine Änderung (Update) auf der alten Sicht und der Basisrelation, die benötigt wird, um die Sicht an die Änderung anzupassen, gesehen. Dabei wird eine Grundmenge von lokalen Änderungen so gewählt, daß eine Folge von lokalen Änderungen durch einfaches Aneinanderhängen durchgeführt werden kann. Meistens kann dieses Aneinanderhängen geschehen, ohne daß Zwischenergebnisse materialisiert werden müssen, was eine einzige Anpassungsmethode für beliebige Änderungen der Sichtendefinition hervorbringt.
Es werden verschiedene Typen für lokale Änderungen und Algorithmen zur Aufrechterhaltung der Sichten in Bezug auf die Änderungen vorgestellt. Diese Algorithmen können in kosten-basierende Abfrage-Optimierer integriert werden, die dann die Kosten der alternativen Methode der Sichtenanpassung mit denen der Neuberechnung vergleichen und dann die günstigste Methode auswählen.
Es wird auch gezeigt, daß es oft sinnvoll ist, zusätzliche Informationen in materialisierten Sichten unterzubringen, um damit Änderungen an der Sichtendefinition viel effizienter durchführen zu können [31].

7.1.3 Ähnliche Problematiken

Das Problem, eine materialisierte Sicht neu zu definieren, ähnelt dem Problem, eine beliebige Abfrage unter der Bedingung, daß die Datenbank eine Sicht V materialisiert hat, zu optimieren. Die Abfrage kann als Neudefinition der Sicht V angesehen werden, und man kann sie neuberechnen, indem die Materialisierung von V geändert wird. Aber es gibt einen wichtigen Unterschied. Angenommen eine Abfrage liefert alle Tupel einer Sicht, außer einem. Wird das ganze als eine Abfrageoptimierung betrachtet, so ist die Komplexität, die Sicht zu verwenden, $O(|V|)$ ($|V|$ ist die Kardinalität von V). Bei der Betrachtung als ein Problem der Sichtenanpassung ist die Komplexität des Prozesses der Aufrechterhaltung der Sicht nur $O(log(|V|))$. Ferner geht bei der Sichtenanpassung die alte Sicht verloren, während sie bei der Abfrage erhalten bleibt.
Die Sichtenanpassung unterscheidet sich von dem Problem, eine materialisierte Sicht zur Beantwortung von Abfragen zu verwenden, auch darin, daß bei der Anpassung die neue Sicht nahe an der Alten liegen muß, was bedeutet, daß die neue Sicht sich nur durch eine kleine Menge von lokalen Änderungen von der Alten unterscheidet. Es gibt im Abfrage-Antwort Problem jedoch keinen Ansatz, der die Sichtenanpassung berücksichtigt, so daß ein Abfrage Compiler/Optimierer viel Zeit darauf verwenden muß, die vorhandenen Sichten zu verwenden, um die Abfrage *richtig* zu beantworten. So betrachtet die Anpassung einen kleineren Suchraum und liefert eine kleinere, aber effizientere Menge von Standard-Techniken, die einfach in relationale Systeme eingegliedert werden können.

7.2 Das Modell

7.2.1 Bedingungen

Die minimale Vorbedingung ist, daß sich die Neudefinition der Sicht durch eine Folge von primitiven lokalen Änderungen ausdrücken läßt. Jede lokale Änderung ist eine kleine Änderung der Sichtendefinition, wie zum Beispiel das Entfernen oder Ändern eines Auswahlprädikats, Hinzufügen eines Ergebnisattributes, Änderungen in der Gruppierung und Hinzufügen einer Verbindung.
Hat man eine neu definierbare Sicht, so muß sich der Administrator zuerst darüber klar werden, (a) ob die Sicht durch zusätzliche Informationen aufgebläht werden soll, die bei späteren Anpassungen nützlich sind, (b) wie die materialisierte Sicht gespeichert werden soll (eventuell noch Platz lassen, damit die Tupel wachsen können), und (c) ob die materialisierte Sicht einen Index erhalten soll.
Das Aufblähen einer Sicht kann nur durch Hinzufügen von Attributen oder Tupeln geschehen, woraus folgt, daß die Sicht eine Projektion der aufgeblähten Sicht sein muß. Die Zusätzlichen Attribute können bei der Anpassung der Sicht, bei Änderungen der Auswahl, Projektion, Gruppierung oder Vereinigung nützlich sein.
Nachdem der Benutzer nun eine Sicht neu definiert hat, wird die Neudefinition in eine Folge primitiver Änderungen übersetzt, und das System muß die aufgeblähte Sicht und die Änderungen analysieren, um festzustellen, (1) ob die aufgeblähte Sicht angepaßt werden kann und (2) welche Algorithmen für die Änderungen verwendet werden können.

7.2.2 Einfache Änderungen

Die folgenden Änderungen werden als primitive Änderungen an Sichten unterstützt.

- Hinzufügen oder Löschen von Attributen in **SELECT**.
- Hinzufügen, Löschen oder Ändern von Prädikaten in **WHERE**.
- Hinzufügen oder Löschen von Verbindungs-Operanden (in **JOIN**) mit den zugehörgigen Prädikaten der natürlichen Verbindung *equijoin* in **SELECT**.
- Hinzufügen oder Löschen von Attributen in **GROUPBY**.
- Hinzufügen oder Löschen von Aggregierungsfunktionen in Sichten mit Gruppierungen.
- Hinzufügen oder Löschen von Operanden der **UNION** und **EXCEPT** Operanden.
- Hinzufügen oder Löschen des **DISTINCT** Operators.

7.2.3 In-place Anpassung

Soll die Sicht V zu der Sicht V' umdefiniert werden, so muß zuerst die neue Sicht materialisiert, die Alte gelöscht und danach die neue Materialisierung V' in V umbenannt werden. Der Anpassungsprozess kann nun Versuchen so viel wie möglich der Materialisierung der alten Sicht zu verwenden, um möglichst wenig Tupel kopieren zu müssen. Deshalb sollte so viel wie möglich direkt geändert werden. Dies kann mit den SQL-Befehlen **INSERT**, **UPDATE**und **DELETE**geschehen. Für Updates wird die folgende erweiterte Syntax verwendet:

UPDATE v **IN** V
SET A (**SELECT** B
FROM $R_1 \bowtie \ldots \bowtie R_m$
WHERE C_1 **AND** ... **AND** C_k).

Die Bedingungen im **SELECT**-Abschnitt in der Unterabfrage können sich auch auf die Variable v beziehen, die geändert werden soll beziehen. Die Unterabfrage soll jedoch nur einen Wert liefern. Es wäre auch möglich, daß A nicht in der ursprünglichen Sicht V ist und in der Neudefinition hinzugegügt wird. Dann muß aber noch Raum für das neue Attribut in der materialisierten Sicht vorhanden sein.

7.3 SELECT-FROM-WHERE Sichten

Dieser Abschnitt beschäftigt sich mit der **SELECT-FROM-WHERE**-Abfrage und Neudefinitionen, die Änderungen in den **SELECT**, **FROM** und/oder **WHERE** Abschnitten vornehmen.
Ein allgemeine Sicht dieses Typs sieht folgendermaßen aus:

CREATE VIEW V **AS**
SELECT $A_1, \ldots, A_n$
FROM $R_1 \bowtie \ldots \bowtie R_m$
WHERE C_1 **AND** ... **AND** C_k

Wie am Anfang schon gesagt, taucht die natürliche Verbindung im **FROM**-Abschnitt der Abfrage auf, weshalb Änderungen darin auch in diesem Abschnitt auftauchen.

7.3.1 Änderungen im SELECT-Abschnitt

Entfernen von Attributen

Beim Entfernen von Attributen kann man entweder einfach die Attribute aus der Materialisierung löschen, oder aber man läßt die alte Sicht unverändert und sorgt am Ende einer Abfrage für eine Projektion, die die entfernten Attribute auslöscht.

Hinzufügen von Attributen

Das Hinzufügen von Attributen ist nicht ganz so einfach. Wenn man in einer aufgeblähten Relation W mehr Attribute, als in V benötigt, hat und bei der Verwendung von V diese ausblendet (Projektion) und diese zusätzlichen Attribute die Hinzuzufügenden sind, so hat man keine Probleme.
Die obige Lösung ist aber nur für wenige Attribute zu gebrauchen, da bei vielen Attributen und Relationen die Datenmenge zu groß wird. Dann ist es besser einfach einen Schlüssel der Basisrelation als zusätzliches Attribut mit in die Relation zu nehmen.

Beispiel 1: Die Datenbank besteht aus den schon bekannten Relationen A, Z und B. Eine Sicht wird wie folgt definiert:

CREATE VIEW V **AS**
SELECT *Name, BauName*
FROM $A \bowtie Z \bowtie B$
WHERE Ort = Karlsruhe

Würde man alle Attribute in einer aufgeblähten Relation halten, so hätte man zehn überflüssige Attribute. Als Alternative könnte man einfach nur $\underline{Ang\#}$ und $\underline{Bau\#}$ in einer aufgeblähten Relation (G) halten.
Angenommen, man will das Attribut *Adresse* der Sicht hinzufügen, so kann dies dann in einem einzigen Durchlauf durch die Relation G und indizierte Abfragen in A basierend auf $\underline{Ang\#}$ geschehen.

UPDATE g **IN** G
SET *Adresse* = (**SELECT** Adresse
FROM A
WHERE $A.\underline{Ang\#} = g.\underline{Ang\#}$)

Das Update kann entweder direkt geschehen, oder man kann das Ergebnis in eine neue Version von G kopieren. Ein Abfrage-Optimierer könnte jedoch auch die **UPDATE**-Anweisung in eine Verbindung zwischen A und G ($A \bowtie G$) umwandeln und alle Tupel von G ändern, die in der Verbindung auftauchen.$\diamond$

Oft besitzt auch die Original-Sicht V einen Schlüssel der Basisrelation R. Wenn eine Neudefinition zusätzliche Attribute aus R benötigt, dann kann die Sicht einfach mit Hilfe des Schlüsseln angepaßt werden.

Ändern des DISTINCT-Schalters

Soll eine Sicht, die vorher keine **DISTINCT** hatte, ein **DISTINCT** erhalten, so müssen nur alle doppelten Tupel aus der materialisierten Sicht gelöscht werden. Durch ein einfaches **SELECT DISTINCT** auf die alte Sicht kann dann die neue Sicht erzeugt werden.

Sehr aufwendig ist das Entfernen des **DISTINCT**-Schalters, da nicht klar ist, wieviele Duplikate in der Sicht vorhanden sind. Einfach zu lösen ist dieses Problem mit einer aufgeblähten Sicht, die alle Tupel besitzt, bei der nur die Projektion, die alle doppelten Tupel ausgeblendet hat, entfernt werden muß.

7.3.2 Änderungen im WHERE-Abschnitt

Im **WHERE**-Abschnitt wird nicht zwischen Bedingungen auf eine Relation und Bedingungen auf mehrere Relationen unterschieden.
Sei C_1' die neue Bedingung, die aus C_1 entstehen soll. Nun soll die neue Sicht V', die folgendermaßen definiert sein könnte

$$\begin{array}{l}\textbf{CREATE VIEW } V' \textbf{ AS}\\ \textbf{SELECT } A_1, \ldots, A_n\\ \textbf{FROM} \quad R_1 \bowtie \ldots \bowtie R_m\\ \textbf{WHERE } C_1' \textbf{ AND} \ldots \textbf{AND } C_k\end{array}$$

unter Berücksichtigung der Tatsache, daß V bereits materialisiert wurde, materialisiert werden.
Algebraisch bedeutet das, $V' = V \cup V^+ \Leftrightarrow V^-$ wobei

$$V^+ = \begin{array}{l}\textbf{SELECT } A_1, \ldots, A_n\\ \textbf{FROM} \quad R_1 \bowtie \ldots \bowtie R_m\\ \textbf{WHERE } C_1' \textbf{ AND NOT } C_1 \textbf{ AND} \ldots \textbf{AND } C_k\end{array}$$

$$V^- = \begin{array}{l}\textbf{SELECT } A_1, \ldots, A_n\\ \textbf{FROM} \quad R_1 \bowtie \ldots \bowtie R_m\\ \textbf{WHERE NOT } C_1' \textbf{ AND } C_1 \textbf{ AND} \ldots \textbf{AND } C_k\end{array}$$

Wenn die Attribute, die in C_1' vorkommen, eine Untermenge von $\{A_1, \ldots, A_n\}$ sind, dann gilt:

$$V^- = \textbf{SELECT } A_1, \ldots, A_n \textbf{ FROM } V \textbf{ WHERE NOT } C_1'$$

oder

$$V \Leftrightarrow V' = \textbf{SELECT } A_1, \ldots, A_n \textbf{ FROM } V \textbf{ WHERE } C_1'$$

V kann dann wie folgt angepaßt werden:

$$\textbf{DELETE FROM } V \textbf{ WHERE NOT } C_1'$$

$$\begin{array}{l}\textbf{INSERT INTO } V \ (\textbf{SELECT } A_1, \ldots, A_n\\ \qquad\qquad\qquad \textbf{FROM} \quad R_1, \ldots, R_m\\ \qquad\qquad\qquad \textbf{WHERE } C_1' \textbf{ AND NOT } C_1 \textbf{ AND} \ldots \textbf{AND } C_k)\end{array}$$

Sind die Attribute von C_1' nicht in der Sicht enthalten, so hat eventuell der Anpassungsalgorithmis der **SELECT**-Anweisung die Attribute in einer aufgeblähten Relation G, oder die benötigten Attribute werden mittels einer Verbindung aus der Basisrelation, die sie enthält, geholt.
Nun sieht man, daß die Kosten einer Anpassung von V in einem der obigen Fälle eine Auswahl auf V (oder G), um V nach $V \Leftrightarrow V^-$ zu bringen, plus die Kosten für die Berechnung von V^+, die in V eingefügt werden muß, betragen.

Beispiel 2: Sei A und Z definiert wie oben. Anahme eine Sicht sei definiert als

CREATE VIEW V **AS**
SELECT * **FROM** $A \bowtie Z$ **WHERE** $Lohn > 20000$

Nun soll V folgendermaßen angepaßt werden:

SELECT * **FROM** $A \bowtie Z$ **WHERE** $Lohn > 25000$

Das sei nun V'. Bezugnehmend auf die obige Ausdrucksweide ist nun C_1 „$Lohn > 20000$" und C_1' ist „$Lohn > 25000$". Daraus folgt, daß V^- und V^+ folgendermaßen definiert werden können:

$$V^- = \begin{array}{l} \textbf{SELECT } * \\ \textbf{FROM } \quad V \\ \textbf{WHERE } Lohn \leq 25000 \textbf{ AND } Lohn > 20000 \end{array}$$

$$V^+ = \begin{array}{l} \textbf{SELECT } * \\ \textbf{FROM } \quad V \\ \textbf{WHERE } Lohn > 25000 \textbf{ AND } Lohn \leq 20000 \end{array}$$

V^+ ist leer, da die Bedingungen im **WHERE**-Abschnitt einander ausschließen. Deshalb betragen die Kosten, um die Sicht neuzuberechnen, nur einen Durchlauf durch V. Wäre V' nun folgendermaßen definiert

SELECT * **FROM** $A \bowtie Z$ **WHERE** $Lohn > 19000$

dann ist V^- leer und V^+ kann durch

SELECT * **FROM** $A \bowtie Z$ **WHERE** $Lohn > 19000$ **AND** $Lohn \leq 20000$

ausgedrückt werden. Ist das Attribut $Lohn$ in A indiziert, so ist die Berechnung von $V \cup V^+$ viel effizienter, als wenn V' von Grund auf neu berechnet werden würde. $\diamond$

Hinzufügen oder Löschen einer Bedingung

Das Hinzufügen einer Bedingung kann man beschreiben, indem man der ursprünglichen Sicht V eine tautologisch wahre Bedingung hinzufügt und diese dann zu V' Ändert. Das hat dann zur Folge, daß V^+ leer ist und die neue Sicht $V' = V \Leftrightarrow V'$.
Das Löschen einer Bedingung kann als das Ersetzen der zu löschenden Bedingung durch eine Tautologie angesehen werden. Dabei ist dann V^- leer und $V' = V \cup V^+$. Ein Optimierer muß nun nur noch die Kosten zur Berechnung von V^+ gegen die Kosten einer kompletten Neuberechnung von V' abwägen.

7.3.3 Ändern des FROM-Abschnittes

Wird eine natürliche Verbindung geändert, so ist es unwahrscheinlich, daß V^+ effizient berechnet werden kann. Diese Annahme kommt aus der intuitiven Ansicht, daß eine solche Änderung große Änderungen in den Daten zur Folge hat und daher mit der alten Sicht V nicht viel anzufangen ist. Es ist jedoch relativ unwahrscheinlich, daß ein Benutzer eine natürliche Verbindung ändern will.
Trotzdem gibt es Situationen, in denen es möglich ist, eine neue Sicht V', in der eine Verbindung entfernt oder hinzugefügt wurde, effizient mit Hilfe der alten Sicht berechnet werden kann.

Hinzufügen einer Verbindung

Angenommen, eine Relation R_{m+1} wird dem **FROM**-Abschnitt hinzugefügt, bei der eine natürliche Verbindung zwischen einem Attribut A von R_{m+1} und einem Attribut B aus einer Relation R_i, $1 \leq i \leq m$, die schon in der Abfrage vorhanden ist, besteht. Es sollen zusätzlich die Attribute $D_1, ..., D_j$ aus R_{m+1} zu der Sicht hinzugefügt werden.
Ist B schon Teil der Sicht, so kann die neue Sicht durch

$$\begin{array}{ll} \textbf{SELECT} & A_1, \ldots, A_n, D_1, \ldots, D_j \\ \textbf{FROM} & V, R_{m+1} \\ \textbf{WHERE} & A = B \end{array}$$

berechnet werden.
Ist das Verbindungattribut A ein Schlüssel, oder ist sichergestellt, daß A eindeutig ist, so kann die Anpassung als ein Update durchgeführt werden. (Die SQL-Syntax wird dahingehend etwas erweitert, daß eine Liste von Werten von einer Abfrage, die genau ein Tupel liefert, zugewiesen werden kann)

$$\begin{array}{lll} \textbf{UPDATE} & v \textbf{ IN } V & \\ \textbf{SET} & D_1, \ldots, D_j = & (\textbf{SELECT } D_1, \ldots, D_j \\ & & \quad \textbf{FROM } R_{m+1} \\ & & \quad \textbf{WHERE } R_{m+1}.A = v.B) \end{array}$$

Ist B nicht in V enthalten, aber ein Schlüssel K von R_i, so kann die Anpassung durch eine Verbindung mit R_i zu berechnen, wenn A Schlüssel in R_{m+1} ist.

UPDATE v **IN** V
SET $D_1, \ldots, D_j$ = (**SELECT** $D_1, \ldots, D_j$
FROM R_{m+1}, R_i
WHERE $R_{m+1}.A = R_i.B$ **AND** $v.K = R_i.K$)

Ist A nicht eindeutig in R_{m+1}, so ist ein Update nicht mehr möglich.

SELECT $A_1, \ldots, A_n, D_1, \ldots, D_j$
FROM V, R_{m+1}, R_i
WHERE $A = B$ **AND** $V.K = R_i.K$)

Beispiel 3: Soll zum Beispiel einer materialisierten Sicht, die Kundennamen und deren Telefonvorwahlen und Nummern enthält die Namen der Städte hinzugefügt werden, so kann dies durch eine natürliche Verbindung zwischen der Relation, die den Telefonnummern die Städtenamen zuordnet, und der Sicht geschehen. $\diamond$

Löschen einer Verbindung

Wird eine Verbindung gelöscht, so muß erhöhte Vorsicht bei der Bestimmung der Anzahl der Duplikate angewendet werden. Wird zum Beispiel aus $R \bowtie S \bowtie T$ die Relation T entfernt, so muß das System zuerst alle Tupel aus $R \bowtie S$ finden, die nicht mit T verbunden werden können, und dann Vielfachheit von Tupeln aus der neuen Sicht. Das Erste kann man getrost ignorieren, wenn die Verbindung mit T verlustfrei ist. Das Letzte kann ignoriert werden, wenn die Sicht keinen Wert auf Duplikate legt (**SELECT DISTINCT**) oder wenn T über seine Schlüsselattribute verbunden wurde und diese in der alten Sicht vorhanden sind.

7.3.4 Zusammenfassung: SELECT-FROM-WHERE-Sichten

In Tabelle 7.2 werden alle Anpassungstechniken für **SELECT-FROM-WHERE**-Sichten zusammengefaßt. Es wird von einer Abfrage, wie sie am Anfang von Sektion 7.3 angegeben ist, ausgegangen. Für jede mögliche Neudefinition wird eine mögliche Anpassung mit den zugehörigen Vorbedingungen angegeben. Diese Vorbedingungen sind in Tabelle 7.1 zu finden. Diese Tabelle kann auf mehrere Arten genutzt werden. Einmal kann ein Optimierer sie verwenden, um die Technik der Anpassung der vollständigen Neuberechnung gegenüberzustellen. Zum Anderen kann ein Datenbank-Betreuer sie dazu verwenden, materialisierte Sichten in Hinblick auf eine leichte Anpassung an Änderungen zu erstellen (aufgeblähte Sichten).

1. Attribut A ist in Relation S und der Schlüssel K für S ist in der Sicht V.
2. Es wird eine aufgeblähte Sicht verwendet, die eine gewisse Anzahl von Ableitungen (Duplikate bzgl. der Sicht) für die Tupel besitzt.
3. Die Attribute der Bedingung sind entweder Teil der Sicht, oder aber in einer aufgeblähten Relation zu finden.
4. $D_1, \ldots, D_j$ and A sind Attribute von R_{m+1} und die Verknüpfungsbedingung ist $A = B$.
5. B ist ein Attribut von V.
6. A ist Schlüssel der Relation R_{m+1}.
7. B ist Attribut von R_i, K ist Schlüssel von R_i und K ist Attribut von V.
8. Eine Verbindung mit R_m ist verlustfrei.
9. V enthält entweder ein **SELECT DISTINCT** oder die Verbindung mit R_m basiert auf einem Schlüsselattribut, das auch in V enthalten ist.

Tabelle 7.1: Vorbedingungen für die Anpassungstechniken in Tabelle 7.2

Hinweise zu aufgeblähten Materialisierungen

- Schlüssel von Relationen, die evtl. eingefügt werden, sollten schon in der aufgeblähten Sicht vorhanden sein.
- In der materialisierten Sicht sollte Platz für Erweiterungen sein.
- Attribute, die in Auswahl-Kriterien vorkommen, sollten in der Materialisierung vorhanden sein (oder zumindest die Schlüssel der Relationen, in denen sie stehen).
- Die Anzahl von abgeleiteten (doppelten) Tupeln sollte in der Materialisierung stehen.

7.4 Sichten mit Aggregierung

Beispiel 4: Es liegen wieder die schon bekannten Relationen A, Z und B zugrunde. Aus der folgenden Sicht sind die Kosten eines Projektes zu ersehen. Dabei gilt, daß ein Arbeiter im Schnitt eine 35-Stunden-Woche hat, und bei Über- oder Unterstunden Zuschläge oder Abzüge erhält.

$$\begin{array}{ll}
\textbf{CREATE} & \textbf{VIEW}\ V(\underline{Bau\#}, Ort, Baukosten)\ \textbf{AS} \\
\textbf{SELECT} & \underline{Bau\#}, Ort, \textbf{SUM}(\frac{Lohn \times Stunden}{35}) \\
\textbf{FROM} & A \bowtie Z \bowtie B \\
\textbf{GROUPBY} & \underline{Bau\#}, Ort
\end{array}$$

Interessieren jetzt nur noch Kosten für einzelne Orte und nicht mehr für Projekte, so entstpricht die dem Entfernen des Attributes $\underline{Bau\#}$ aus dem **GROUPBY**-Abschnitt. Dies ergibt dann die folgende Sicht:

$$\begin{array}{ll}
\textbf{CREATE} & \textbf{VIEW}\ V'(Ort, Baukosten)\ \textbf{AS} \\
\textbf{SELECT} & Ort, \textbf{SUM}(\frac{Lohn \times Stunden}{35}) \\
\textbf{FROM} & A \bowtie Z \bowtie B \\
\textbf{GROUPBY} & Ort
\end{array}$$

Unter Ausnutzung der Kommutativität der Summenfunktion kann nun ein Optimierer die neue Sicht V' aus der alten Sicht V berechnen.

$$\begin{array}{ll}
\textbf{SELECT} & Ort, \textbf{SUM}(Baukosten) \\
\textbf{FROM} & V \\
\textbf{GROUPBY} & Ort
\end{array}$$

Diese Methode ist viel effizienter, als die vollständige Neuberechnung der Sicht.
Soll nun die Summe der Löhne für jede Baustelle ausgegeben werden, so muß nur das Attribut Ort aus V entfernt werden, da $\underline{Bau\#}$ der Schlüssel von B ist und daher die Gruppen identisch sind. $\diamond$

Neudefinierte Sicht	Anpassungstechnik	Bedingungen
SELECT $A, A_1, \ldots, A_n$ **FROM** $R_1 \bowtie \ldots \bowtie R_m$ **WHERE** C_1 **AND** ... **AND** C_k	**UPDATE** v **IN** V **SET** $A = ($**SELECT** A **FROM** S **WHERE** $S.K = v.K$	(1)
SELECT $A_2, \ldots, A_n$ **FROM** $R_1 \bowtie \ldots \bowtie R_m$ **WHERE** C_1 **AND** ... **AND** C_k	**UPDATE** v **IN** V **DROP** A_1	
SELECT DISTINCT $A_1, \ldots, A_n$ **FROM** $R_1 \bowtie \ldots \bowtie R_m$ **WHERE** C_1 **AND** ... **AND** C_k	**INSERT INTO** V' **SELECT DISTINCT** * **FROM** V	
SELECT DISTINCT $A_1, \ldots, A_n$ **FROM** $R_1 \bowtie \ldots \bowtie R_m$ **WHERE** C_1 **AND** ... **AND** C_k	Die Sicht als **DISTINCT** definieren	(2)
SELECT $A_1, \ldots, A_n$ **FROM** $R_1 \bowtie \ldots \bowtie R_m$ **WHERE** C'_1 **AND** ... **AND** C_k	**DELETE** **FROM** V **WHERE NOT** C'_1	$C'_1 \Rightarrow C_1$, (3)
SELECT $A_1, \ldots, A_n$ **FROM** $R_1 \bowtie \ldots \bowtie R_m$ **WHERE** C'_1 **AND** ... **AND** C_k	**DELETE** **FROM** V **WHERE NOT** C'_1 **INSERT INTO** V' **SELECT** $A_1, \ldots, A_n$ **FROM** $R_1 \bowtie \ldots \bowtie R_m$ **WHERE** C'_1 **AND NOT** C_1 **AND** ... **AND** C_k	$C'_1 \not\Rightarrow C_1$, (3)
SELECT $A_1, \ldots, A_n$ **FROM** $R_1 \bowtie \ldots \bowtie R_m$ **WHERE** C_0 **AND** C_1 **AND** ... **AND** C_k	**DELETE** **FROM** V **WHERE NOT** C_0	(3)
SELECT $A_1, \ldots, A_n$ **FROM** $R_1 \bowtie \ldots \bowtie R_m$ **WHERE** C_2 **AND** ... **AND** C_k	**INSERT INTO** V' **SELECT** $A_1, \ldots, A_n$ **FROM** $R_1 \bowtie \ldots \bowtie R_m$ **WHERE NOT** C_1 **AND** C_2 **AND** ... **AND** C_k	(3)

Tabelle 2a: Anpassungstechniken für **SELECT-FROM-WHERE**-Sichten

Neudefinierte Sicht	Anpassungstechnik	Bedingungen
SELECT $A_1, \ldots, A_n, D_1, \ldots, D_j$ **FROM** $R_1 \bowtie \ldots \bowtie R_m \bowtie R_{m+1}$ **WHERE** C_1 **AND** $\ldots$ **AND** C_k	**UPDATE** v **IN** V **SET** $D_1, \ldots, D_j =$ (**SELECT** $D_1, \ldots, D_j$ **FROM** R_{m+1} **WHERE** $R_{m+1}.A = v.B$)	(4, 5, 6)
SELECT $A_1, \ldots, A_n, D_1, \ldots, D_j$ **FROM** $R_1 \bowtie \ldots \bowtie R_m \bowtie R_{m+1}$ **WHERE** C_2 **AND** $\ldots$ **AND** C_k	**INSERT INTO** V' **SELECT** $A_1, \ldots, A_n, D_1, \ldots, D_j$ **FROM** V, R_{m+1} **WHERE** $A = B$	(4, 5)
SELECT $A_1, \ldots, A_n, D_1, \ldots, D_j$ **FROM** $R_1 \bowtie \ldots \bowtie R_m \bowtie R_{m+1}$ **WHERE** C_1 **AND** $\ldots$ **AND** C_k	**UPDATE** v **IN** V **SET** $D_1, \ldots, D_j =$ (**SELECT** $D_1, \ldots, D_j$ **FROM** R_{m+1}, R_i **WHERE** $R_{m+1}.A = R_i.B$ **AND** $v.K = R_i.K$)	(4, 6, 7)
SELECT $A_1, \ldots, A_n, D_1, \ldots, D_j$ **FROM** $R_1 \bowtie \ldots \bowtie R_m \bowtie R_{m+1}$ **WHERE** C_2 **AND** $\ldots$ **AND** C_k	**INSERT INTO** V' **SELECT** $A_1, \ldots, A_n, D_1, \ldots, D_j$ **FROM** V, R_{m+1}, R_i **WHERE** $A = B$ **AND** $V.K = R_i.K$	(4, 5)
SELECT $A_1, \ldots, A_n$ **FROM** $R_1 \bowtie \ldots \bowtie R_{m-1}$ **WHERE** C_1 **AND** $\ldots$ **AND** C_k	Anpassung nicht nötig	(8, 9)
SELECT $A_1, \ldots, A_j$ **FROM** $R_1 \bowtie \ldots \bowtie R_{m-1}$ **WHERE** C_1 **AND** $\ldots$ **AND** C_k	**UPDATE**v **IN** V **DROP** $A_{j+1}, \ldots, A_n$	(8, 9)

Tabelle 2b: Anpassungstechniken für **SELECT-FROM-WHERE**-Sichten

7.4.1 Entfernen von GROUPBY-Spalten

In einer Sicht mit Aggregierung wird jede Menge von Tupeln, die die gleichen Werte in den gruppierten Attributen besitzen, eine Gruppe genannt. In Beispiel 4 ist also jedes Tupel, das die gleichen Werte im Paar ($\underline{Bau\#}$, Ort) besitzt in der gleichen Gruppe. In V' ist jedes Tupel mit dem gleichen Wert für Ort in der gleichen Gruppe.
Wird ein Gruppierungsattribut entfernt, so kann man versuchen die Anpassung durch die Kombination der Aggregationswerte von den ursprünglichen Gruppen durchzuführen. In Beispiel 4 wurde die Entfernung von $\underline{Bau\#}$ aus der Gruppierung dadurch durchgeführt, daß die Sicht V über dem Ort gruppiert wurde und dann die $Baukosten$ addiert wurden. Beim Entfernen von Ort wurde keine neue Aggregierung benötigt.
Unter folgenden Bedingungen kann eine materialisierte Sicht beim Entfernen eines Gruppierungattributs angepaßt werden:

- Das entfernte Gruppierungsattribut ist funktional bestimmt durch die zurückbleibenden Gruppenspalten.
- Die Aggregierungsfunktionen in der neudefinierten Sicht können durch Berechnungsfunktionen über die ursprünglichen Aggregierungsfunktionen bestimmt werden. In Tabelle 7.3 sind Aggregierungsfunktionen angegeben, die so berechnet werden können.

Tabelle 7.3 ist nicht vollständig, sondern skizziert nur die Möglichkeiten.

Hinzufügen von GROUPBY-Spalten

Im allgemeinen ist es beim Hinzufügen von **GROUPBY**-Spalten nötig, alles von Grund auf neuzuberechnen, da die Körnigkeit in den Aggregierungsfunktionen kleiner wird. Ist das neuhinzugekommene Attribut schon durch die ursprünglichen Gruppierung bestimmt, so genügt es einfach das neue Attribut in die Sicht zu projizieren.
Wenn die ursprüngliche Sicht noch keine Gruppierung enthält und alle Attribute, die für die Gruppierung nötig sind, schon in der Sicht enthalten sind, so kann die neue Sicht einfach durch Hinzufügen der **GROUPBY**-Spalte aus der Alten berechnet werden. Sind die benötigten Attribute noch nicht vorhanden, so konnen diese, wie in vorigen Abschnitten beschrieben, hinzugefügt werden.

Hinzufügen und Entfernen von Aggregierungsfunktionen

Das Entfernen einer Aggregierungsfunktion entstpricht exakt dem Ausblenden eines Attributes. Es ist nur selten möglich Aggregierungsfunktionen hinzuzufügen, es sei denn, die Funktion kann durch schon vorhandene Funktionen ausgedrückt werden, oder aber die Sicht ist aufgebläht. Ein Beispiel dafür ist das Aufblähen mit allen Schlüsselattributen, die in den Relationen, die in der Sicht verwendet werden, vorkommen.

Hinweis für die Aufblähung: Aus Tabelle 7.3 ist zu erkennen, daß es äußerst sinnvoll ist, in eine materialisisierte Sicht mit Aggregierungen, ein zusätzliches Attribut mit der Aggregierungsfunktion $COUNT(*)$ aufzunehmen.

Neudefinierte Aggr.	**Anpassung mit der Originalsicht**
$MIN(X)$	$MIN(M)$, wobei $M = MIN(X)$ eine ursprüngliche Aggregierungsspalte war.
$MAX(X)$	$MAX(M)$, wobei $M = MAX(X)$ eine ursprüngliche Aggregierungsspalte war.
$MIN(X)$	$MIN(X)$, wobei X eine ursprüngliche Gruppierungsspalte war.
$MAX(X)$	$MAX(X)$, wobei X eine ursprüngliche Aggregierungsspalte war.
$SUM(X)$	$SUM(S)$, wobei $S = SUM(X)$ eine ursprüngliche Aggregierungsspalte war.
$SUM(X)$	$SUM(X \times C)$, wobei $C = COUNT(*)$ eine ursprüngliche Aggregierungsspalte und X eine Gruppierungsspalte war.
$COUNT(*)$	$SUM(C)$, wobei $C = COUNT(*)$ eine ursprüngliche Aggregierungsspalte war.
$AVG(X)$	$SUM(A \times C)/SUM(C)$, wobei $C = COUNT(*)$ und $A = AVG(X)$ ursprüngliche Aggregierungsspalten waren.
$AVG(X)$	$SUM(X \times C)/SUM(C)$, wobei $C = COUNT(*)$ eine ursprüngliche Aggregierungsspalte und X eine Gruppierungsspalte war.

Tabelle 7.3: Anpassungen von Aggregierungsfunktionen

7.4.2 Sichten über Vereinigungen und Differenzen

UNION

Sei eine Sicht V definiert als eine Vereinigung von Abfragen V_1 und V_2. Ändert sich die Definition von V in entweder V_1 oder V_2 (nicht in Beiden), so ist es angebracht, eine der zuvor vorgestellten Techniken zu verwenden, um die Materialisierung von V_1 oder V_2 anzupassen.
Um dies zu tun, muss man wissen, welche Tupel zu V_1 und welche zu V_2 gehören. Mit diesem Wissen ist es einfach möglich, die einen Tupel zu belassen und die Anderen anzupassen. Deshalb ist es angebracht zu jedem Tupel noch die Abfrage (V_1 oder V_2), zu der er gehört, mit abzuspeichern. Es wäre auch möglich Materialisierungen von V_1 und V_2 getrennt zu speichern und die Vereinigung nur bei der Verwendung von V direkt zu erstellen.

Beispiel 5: Angenommen man will entweder die Namen aller Arbeiter, die an einer Baustelle in Karlsruhe arbeiten, oder die Namen derjenigen, die Bauleiter einer Baustelle in Karlsruhe sind. Diese Sicht V kann als Vereinigung zweier Unterabfragen V_1 und V_2 geschrieben werden.

$$V_1 = \begin{array}{ll} \textbf{SELECT} & Name, SubQ = "V_1" \\ \textbf{FROM} & A \bowtie Z \bowtie B \\ \textbf{WHERE} & Ort = Karlsruhe \end{array}$$

$$V_2 = \begin{array}{ll} \textbf{SELECT} & Name, SubQ = "V_2" \\ \textbf{FROM} & A, B[A.\underline{Ang\#} = B.Leiter\#] \\ \textbf{WHERE} & Ort = Karlsruhe \end{array}$$

Das Attribut $SubQ$ sollte nicht dem Benutzer gezeigt, aber in einer aufgeblähten Relation gespeichert werden. Soll nun V_1 geändert werden, so können alle Tupel mit $SubQ = "V_1"$ mit Techniken der vorigen Abschnitte bearbeitet werden, während die anderen unverändert bleiben. $\diamond$
Das Löschen eines **UNION**-Operanden ist dann einfach, wenn bekannt ist, aus welcher Unterabfrage welcher Tupel stammt. Dann müssen einfach alle in Frage kommenden Tupel gelöscht werden.
Das Hinzufügen eines **UNION**-Operanden ist genauso einfach. Die alte Relation bleibt unverändert und es wird einfach die hinzugekommene Unterabfrage ausgewertet, um die hinzuzufügenden Tupel zu generieren.

EXCEPT

Beispiel 6: Will man alle Arbeiter, die auf einer Baustelle in Karlsruhe arbeiten, aber keine Baustellenleiter sind, so kann die Sicht V als Differenz von V_1 und V_2 (V_1 und V_2 aus Beispiel 5) ausgedrückt werden:

$V = V_1$ **EXCEPT** V_2

◇

Anders als bei den Vereinigungen ist es weitaus sinnvoller V aufzublähen, als die beiden Unterabfragen V_1 und V_2. Aus diesem Grund kann man nicht davon ausgehen, daß man bei jedem Tupel weiß aus welcher Unterabfrage er stammt.
In zwei Fällen kann man trotzdem die alte Sicht weiterverwendet werden, um die neue Sicht effizienter zu berechnen.

1. Wird V_2 durch eine Sicht V_2', die garantiert mehr Tupel enthält, ersetzt, dann gilt, daß V_2^- leer ist und $V' = V$ **EXCEPT** V_2^+.

2. Wenn V_1 durch eine Sicht V_1' ersetzt, die garantiert weniger Tupel enthält, dann ist V_1^+ leer und $V' = V$ **EXCEPT** V_1^-

Soll eine Unterabfrage V_2 von einer materialisierten Sicht V differenziert werden, so läßt sich die neue Sicht V' effizient mit dem obigen Punkt 1 berechnen, für die neue Sicht gilt also: $V' = V$ **EXCEPT** V_2.
Im allgemeinen Fall gibt es für den Optimierer eine weiter Möglichkeit V' zu berechnen. Angenommen V_2 ändert sich und V_2^- und V_2^+ sind nicht leer, dann gilt, daß $V' = V$ **EXCEPT** V_2^+ **UNION** U, wobei $U = V_1 \cap V_2^-$. Selbst wenn V_1 nicht materialisiert ist, ist es möglich U zu berechnen, indem für alle Tupel aus V_2^- überprüft wird, ob sie den Bedingungen aus V_1 genügen. Diese Methode ist vor allem dem kompletten Neuberechnen von V vorzuziehen, wenn V_2^- und V_2^+ klein sind.

Hinweis für die Aufblähung: Es sollte zu jedem Tupel ein Attribut gespeichert werden, in dem steht, aus welcher Unterabfrage er kommt.

7.5 Mehrere Änderungen auf einer Sichtendefinition

Bis jetzt wurden nur einzelne Änderungen auf einer Sichtendefinition betrachtet. Es kann jedoch auch vorkommen, daß ein Benutzer gleichzeitig mehrere lokale Änderung auf einer Sicht durchführen will. Dabei kann die neue Sicht einfach durch Aneinanderhängen der einzelnen Änderungen berechnet werden. Der Nachteil dabei ist jedoch, daß dabei viele Zwischenergebnisse materialisiert werden, was nicht unbedingt notwendig sein muß.
Sollen zum Beispiel mehrere Bedingungen im **WHERE**-Abschnitt gleichzeitig geändert werden, so kann man dies mit Hilfe von 7.3.2 erledigen, wenn man sich C_1 und C_1' als Konjunktion von Bedingungen vorstellt. Genauso könne mehrere Attribute im **SELECT**-Abschnitt gelöscht oder hinzugefügt werden, wenn die Techniken von 7.3.1 verwendet werden. Mehrere Relationen können zum **FROM**-Abschnitt hinzugefügt werden, wenn man 7.3.3 verwendet. Dabei werden jeweils keine Zwischenergebnisse materialisiert.

Sollen verschieden Typen von Änderungen gleichzeitig angewendet werden, so kann man das Materialisieren von Zwischenergebnissen dadurch verhindern, daß das Ergebnis einer Änderung an die nächste Änderung direkt weitergepiped wird. Pipelining kann bei allen Änderungen verwendet werden, die in einem Durchgang ablaufen können (one pass). Das bedeutet, daß bis auf eine Ausnahme alle hier aufgeführten Änderungen gepiped werden können. Die Ausnahme ist, daß eine zuvor materialisierte Sicht V in einer Agregierung, die über ein Attribut gruppiert wird, das kein Sortierattribut für V ist, verwendet wird. Ein Optimierer muß folglich aus den folgenden drei Möglichkeiten die Beste auswählen:

- Anwenden von aufeinanderfolgenden 'in place' Änderungen
- Verschachteln der Änderungen, wie oben beschrieben
- Alles komplett neu berechnen

7.6 Zusammenfassung

Bei der Änderung einer materialisierten Sicht muß die Materialisierung auf den neuesten Stand gebracht werden. Dieser Text setzt dabei sein Hauptaugenmerk auf die Anpassung der Materialisierung. Die Alternative dazu wäre die völlige Neuberechnung der Sicht. Es ist sehr oft viel effizienter die Sicht anzupassen, anstatt sie neuzuberechnen.
Eine große Anzahl von Applikationen, wie in der Daten-Archäologie und Datenvisualisierungsprogramme benötigen schnelle Reaktionszeiten, bei Änderungen der Definition einer materialisierten Sicht.
Es wurde hier eine umfassende Liste von Anpassungstechniken für Grund-Änderungen vorgestellt, die, ausgedrückt als eine SQL-Abfrage oder -Update, immer von der alten materialisierten Sicht Gebrauch machen. Ein Abfrage-Optimierer kann mit einer Standard-Kostenrechnung die Kosten für die Anpassung berechnen und mit denen der kompletten Neuberechnung vergleichen.
Die grundlegenden Anpassungstechniken entsprechen lokalen Änderungen in der Sichtendefinition. Es wird auch beschrieben, wie mehrere solche Anpassungen gleichzeitig durchgeführt werden können. Dabei können alle Techniken, bis auf eine, in einer Pipeline ausgeführt werden, damit keine Zwischenergebnisse materialisiert werden müssen.
Es ist oft einfacher, eine Sicht anzupassen, wenn bestimmte zusätzliche Attribute in einer aufgeblähten Materialisierung gespeichert werden. Das sind zum Beispiel Schlüsselattribute der zugrundeliegenden Relationen, Attribute, die in Auswahlkriterien vorkommen, die Anzahl der zusammengefaßten Tupel bei Aggregierungen, zusätzliche Aggregierungsfunktionen und Identifikationen, mit denen sich bei einer Vereinigung feststellen läßt, aus welcher Unterabfrage die Tupel kommen. Zusätzlich kann es nützlich sein, wenn zusätzlicher Platz in einer Materialisierung freigehalten wird, damit neue Tupel hinzugefügt werden können.
Aus diesen Anpassungstechniken wurden Tabellen entwickelt, die auf verschiedene Arten genutzt werden können:

- Ein Abfrageoptimierer kann die Tabellen nutzen, um die richtige Anpassungstechnik aus den gegebenen Eigenschaften des Schemas, gegenüber den Vorbedingungen aus den Tabellen, zu finden.

- Der Datenbank Administrator oder Benutzer kann die Vorbedingungen aus den Tabellen dazu verwenden bei der Definition der Sicht dafür zu sorgen, daß spätere Anpassungen leicht möglich sind.

- Der Datenbank Administrator kann den Abfrageoptimierer so beeinflussen, daß Zugriffsmethoden und Indizes auf den Grundrelationen und Sichten so aufgebaut sind, daß eine Anpassung möglichst einfach ist.

Kapitel 8

Konzepte des Online Analytical Processing (OLAP)

Rainer Ruggaber

Kurzfassung *Die Analyse von Unternehmensdaten ist eine zentrale Voraussetzung um Entscheidungen im Unternehmen zu treffen. Die großen Datenmengen und die trotzdem vom Benutzer erwarteten kurzen Antwortzeiten, sowie die Darstellung der Daten in mehrdimensionalen Datenwürfel sind die Anforderungen an solche entscheidungsunterstützenden Systeme. Der mehrdimensionale Datenwürfel ist die zentrale Datenstruktur im Online Analytical Processing (OLAP). Alle Operationen sollten direkt auf dieser Struktur ausgeführt werden können. Um der mehrdimensionalen Datenanalyse eine theoretische Grundlage zu geben, wird eine Algebra für mehrdimensionale Daten vorgestellt. Zuerst wird die Eignung von mehrdimensionalen Datenbanken für das OLAP untersucht. Diese speichern intern die Daten in einer mehrdimensionalen Struktur. Auch relationale Datenbanken, die hauptsächlich zur Transaktionsverarbeitung genutzt werden, werden auf ihre Eignung für das OLAP untersucht. Relationale Datenbanken speichern Daten intern jedoch in Tabellen. Hauptaugenmerk lag bei der Untersuchung auf der Eignung von SQL der Standardanfragesprache für relationale Datenbanken, für MOLAP. Einige Firmen erweitern SQL, um einige der Schwächen, die SQL bei Aggregationsoperationen besitzt, zu beseitigen. Auch diese Erweiterungen werden vorgestellt.*

8.1 Einleitung

Eine schnelle Reaktionszeit auf neue Marktanforderungen wird in Zeiten des immer größer werdenden Konkurrenzdruckes immer wichtiger. Das Konsumverhalten der Kunden und die zur Herstellung von Produkten eingesetzten Verfahren ändern sich immer schneller. Die Produktzyklen werden immer kürzer. Deshalb müssen immer öfter wichtige Entscheidungen im Unternehmen getroffen werden. Gleichzeitig werden in den Unternehmen mehr und mehr Daten gesammelt. Diese Daten liegen unter Umständen in verschiedenen Datenbanken an unterschiedlichen Orten. Diese Datenflut in Unternehmen macht es für den, der Entscheidungen treffen muß immer schwieriger, die für ihn wichtigen Informationen zu finden [32].

Ein Data Warehouse ist dabei eine der Möglichkeiten, dieses Dilemma zu lösen. In einem Data Warehouse werden die für Unternehmensentscheidungen wichtigen Informationen bereitgestellt. Diese Informationen können aus verschiedenen Datenbanken, die innerhalb oder außerhalb des Unternehmens liegen, stammen. Die Informationen werden so aufbereitet, daß entscheidungsunterstützende Systeme diese nutzen können. Beispiele für solche Systeme, die auf ein Data Warehouse zugreifen, sind Management-Informationssyteme, Online Analytical Processing (OLAP) Syteme und Data-Mining Syteme, wobei die ersten beiden Arten von Systemen einen passiven Charakter haben. Sie stellen nur, wenn auch auf komfortable Weise, die im Data Warehouse gespeicherten Daten dar. Das Data-Mining, auf der anderen Seite, versucht die in der Datenbank implizit vohandenen Muster, und Zusammenhänge weitgehend automatisch, zu erkennen.

OLAP und die mehrdimensionale Darstellung von Unternehmensdaten sind heute zu Synonymen geworden. Die mehrdimensionale Darstellung ist für den Benutzer anschaulich und dadurch intuitiv zu bedienen. Die Basis dieser mehrdimensionalen Darstellung können mehrdimensionale Datenbanken sein.

Im zweiten Abschnitt wird beschrieben, was unter OLAP zu verstehen ist, und welche Anforderungen an ein OLAP-System gestellt werden. Wichtig sind hier die Art der darstellbaren Daten und die Operationen, die auf diese Daten angewand werden können. Im dritten Abschnitt werden die mehrdimensionalen Datenbanken vorgestellt. Der Einsatz relationaler Datenbanken für OLAP-Systeme wird im vierten Abschnitt diskutiert. Dabei wird untersucht, welche Funktionen SQL für OLAP bereitstellt, und welche proprietären SQL-Erweiterungen existieren. Im fünften Abschnitt werden die vorgestellten Varianten bewertet und verglichen. Im letzten Abschnitt folgt eine Zusammenfassung.

8.2 OLAP (Online Analytical Processing)

8.2.1 Einleitung

OLAP-Systeme dienen zur komfortablen Analyse, im Unternehmen vorhandener Daten. Ein Beispiel für diese Art von Daten sind die Verkaufszahlen mehrerer Produkte, die an verschiedenen Tagen, in verschiedenen Geschäften verkauft wurden. Ein intuitives Vorgehen, um einen Überblick über die Verkaufszahlen zu erhalten wäre, zuerst die Produkte zu Produktgruppen und die Geschäfte in Regionen zusammenzufassen. Anschließend könnte man sich Produkte aus Produktgruppen, die in bestimmten Regionen auffällig sind, genauer betrachten.

Dieses Vorgehen ist typisch bei der Analyse. Man beginnt im Allgemeinen und untersucht nur auffällige Bereiche genauer. Dieses Vorgehen wird drill-down genannt. Die Anforderung an das OLAP-System ist in diesem Fall die detaillierten Daten schnell bereitzustellen, da sonst der Analysevorgang beim Anwender stark behindert wird. Ein anderes Vorgehen wäre, sich Verkaufszahlen von Produktgruppen im Zeitverlauf anzeigen zu lassen, um so Vorhersagen für zukünftige Verkaufszahlen machen zu können, oder bei unbefriedigenden Verkaufszahlen oder Tendenzen, Strategien zur Förderung bestimmter Produkte frühzeitig einzuleiten. In diesem Fall sollte das OLAP-System spezielle Funktionen für Zeitreihen bereithalten, wie sie unten noch vorgestellt werden.

Das Online Analytical Processing (OLAP) beschreibt eine Menge von Funktionen zur Analyse und Verarbeitung von Daten. Im Gegensatz dazu dienen OLTP-Systeme (Online Transaction Processing) der Transaktionsverarbeitung. Für OLTP-Anwendungen werden hauptsächlich relationale Datenbanken eingesetzt. Typische OLTP-Anwendungen erzeugen, ändern oder suchen einzelne Datensätze einer Datenbank. Relationale Datenbanksysteme sind auf diese Art von Zugriffen optimiert. Demgegenüber wird in einer OLAP-Anwendung eine große Anzahl Datensätze ausgewählt und bestimmte Operationen auf diese Auswahl angewand. Da relationale Datenbanksysteme nicht auf diese Art von Anfragen optimiert sind, haben sie extrem lange Laufzeiten. Im Gegensatz zu OLTP-Anwendungen benötigen die OLAP-Anwendungen keinen schreibenden Zugriff auf die Daten, so daß eine aufwendige Zugriffskontrolle innerhalb der Datenbank entfallen kann. Die Daten werden meist in Zeiten niedriger Last aus den OLTP-Datenbanken in das Data Warehouse übertragen. Dabei können die Daten aus verschiedenen, auch örtlich verteilten, Datenbanken stammen.

8.2.2 Mehrdimensionale Darstellung von Daten

Bei einem Autohändler liegen zum Beispiel die Verkaufszahlen für PKW's in der folgenden Tabelle vor.

Fahrzeugtyp	Farbe	Anzahl
Van	Blau	6
Van	Rot	5
Van	Weiß	4
Sportwagen	Blau	3
Sportwagen	Rot	5
Sportwagen	Weiß	5
Kombi	Blau	4
Kombi	Rot	3
Kombi	Weiß	2

Die Darstellung der PKW-Verkäufe als Tabelle ist nicht übersichtlich. Um beispielsweise feststellen zu können wieviel blaue PKW's verkauft wurden, müssen zunächst alle Einträge der Tabelle durchsucht werden. Handelt es sich um einen Eintrag für einen blauen PKW, wird dieser zur Auswahl hinzugenommen. Anschließend werden die Verkäufe der Einträge in der Auswahl summiert. Eine andere typische OLAP-Anfrage wäre, für jeden Fahrzeugtyp die Summe der verkauften PKW's zu bestimmen. Der Benutzer müßte hierzu zuerst alle unterschiedlichen Fahrzeugtypen bestimmen, um anschließend die Summe für jeden der gefundenen Fahrzeugtyp zu bestimmen. Eine andere Möglichkeit wäre die Tabelle als Matrix darzustellen. Das Ergebnis der Matrixdarstellung sieht man in Abb. 8.1.

Ein Vorteil dieser Darstellungsart ist, daß der Benutzer sofort die Anzahl der in der Datenbank vorhandenen Ausprägungen einer Dimension (hier: Farbe und Fahrzeugtyp) erkennt. In diesem Beispiel gibt es für Farbe drei Ausprägungen (Rot, Blau und Weiß). Ein weiterer Vorteil ist, daß die Werte für eine bestimmt Ausprägung in einer Zeile oder Spalte stehen. In obiger Matrix stehen zum Beispiel die Verkaufszahlen aller roten Modelle in der ersten Spalte. Um die Summe der verkauften roten PKW's zu bestimmen, muß nur die Spalte für die Farbe rot gesucht werden um die Werte dort zu summieren. Um die

Fahrzeugtyp

	Rot	Weiß	Blau
Van	5	4	6
Sportwagen	5	5	3
Kombi	3	2	4

Farbe

Abbildung 8.1: Die Verkaufszahlen des Autohändlers in Matrixdarstellung

Summe der Verkaufszahlen für jede Farbe zu bestimmen, muß in der Matrixdarstellung nur die Dimension für Fahrzeugtypen ausgewählt werden, und für jedes Element der Dimension die Zeilensumme bestimmt werden. Die beiden Beispielanfragen lassen sich in der Matrixdarstellung also auf Spaltensummen und Zeilensummen zurückführen.

Die Achsen in dieser Matrix werden *Dimension* genannt, im obigen Beispiel Fahrzeugtyp und Farbe. Die diskreten Einteilungen einer Dimension wird *position* oder *member* genannt, im obigen Beispiel sind das Blau, Rot und Weiß für Farbe. Die eigentlichen Daten befinden sich im Innern der Matrix und werden *cells* genannt, das sind im Beispiel die Verkaufszahlen.

Die Darstellung von Daten als Matrix läßt sich bezüglich der Dimensionen verallgemeinern. Der Großhändler der unseren Autohändler beliefert, hat eine ähnliche Tabelle. Diese Tabelle hat eine vierte Spalte in welcher der belieferte Händler vermerkt ist. Die aus dieser Tabelle resultierende mehrdimensionale Darstellung hat drei Dimensionen. Der Fahrzeugtyp und die Farbe sind, wie bisher, Dimensionen, und zusätzlich existiert noch eine Dimension für den Händler. Mit diesen drei Dimensionen lassen sich nun auch Fragen nach allen Verkäufen eines Autohändlers beantworten. In diesem Fall müßen die *cells* der gesamten Ebene des gewählten Händlers addiert werden. Den data cube sieht man in Abb. 8.2

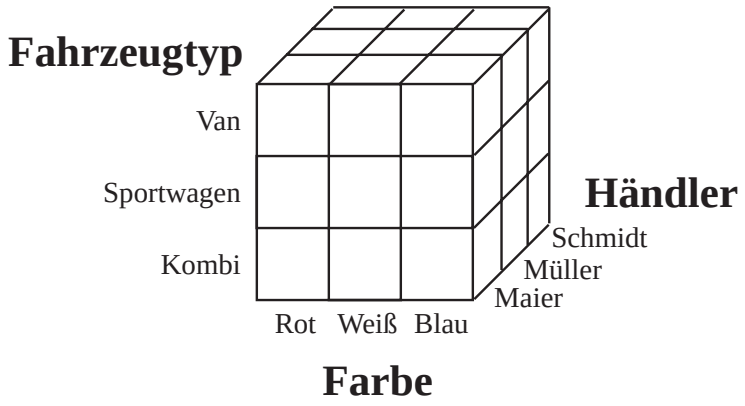


Abbildung 8.2: Die Verkaufszahlen des Großhändlers als Data Cube

Falls der Großhändler in seiner Tabelle auch noch die Woche, in der die Verkäufe stattgefunden haben, vermerkt, wird die graphische mehrdimensionale Darstellung schwierig.

Diese hat nun vier Dimensionen, und ist nur dann graphisch darstellbar wenn auf eine der Dimensionen bei der visuellen Darstellung verzichtet wird. Dieses Verzichten auf eine Dimension kann auf zweierlei Weise geschehen. Einerseits kann man die Werte entlang dieser Dimension zu einem neuen Wert zusammenfassen oder andererseits für jede *position* dieser Dimension einen eigenen data cube darstellen. Die Zusammenfassung von Werten entlang einer Dimension kann verschieden erfolgen. Im Falle des Großhändlers addiert man die Verkaufszahlen auf, bei anderen Arten von Werten kann auch der Mittelwert oder das Maximum die Zusammenfassung sein. Um die Verkaufsverläufe der Händler beobachten zu können, verzichtet man auf eine eigene Händlerdimension. Es existiert nun für jeden der Händler ein eigener Kubus mit den Dimensionen Fahrzeugtyp, Farbe, und Woche. Diese Festlegung ist nicht statisch, und sollte dem Benutzer überlassen werden.

Es gibt jedoch auch Informationen, bei denen die mehrdimensionale Darstellung nicht von Vorteil ist. Dazu soll folgendes Beispiel betrachtet werden. Die Personaltabelle des Autohändlers enthält die Personalnummer, den Namen und die Telefonnummer des Mitarbeiters. Auch diese Tabelle hat mehrere Spalten, wie das vorhergehende Beispiel.

Personalnummer	Name	Telefon
1	Müller	643822
2	Schneider	9732331
3	Maier	0743/32215
4	Schmidt	753328

Sie eignet sich jedoch trotzdem nicht für eine mehrdimensionale Darstellung. Wählt man zum Beispiel Name und Personalnummer als Dimensionen sähe die Matrix wie in Abb. 8.3 aus.

Name

Müller	**643822**			
Schneider		**9732331**		
Maier			**0743/32215**	
Schmidt				**753328**
	1	2	3	4

Personalnummer

Abbildung 8.3: Die Personaldaten des Autohändlers als Matrix

Eine Telefonnummer ist einem Namen eindeutig zugeordnet. Durch die eindeutige Zuordnung von Telefon zu Namen hat die mehrdimensionale Darstellung dieser Tabelle keine Vorteile. Sollten in der Tabelle jedoch zwei Mitarbeiter den gleichen Namen haben, würde in der entsprechenden Zeile zwei Telefonnummern stehen. Unabhängig davon, welche Dimensionen man wählt, erhält man stets eine eindeutige Zuordnung. Die Daten in dieser Tabelle sind nur eindimensional.

8.2.3 Funktionen mehrdimensionaler OLAP-Systeme

Rotation oder data slicing

Die Drehung des mehrdimensionalen data cubes um 90 Grad wird als *rotation* oder *data slicing* bezeichnet. Ein Beispiel sieht man in Abb. 8.4.

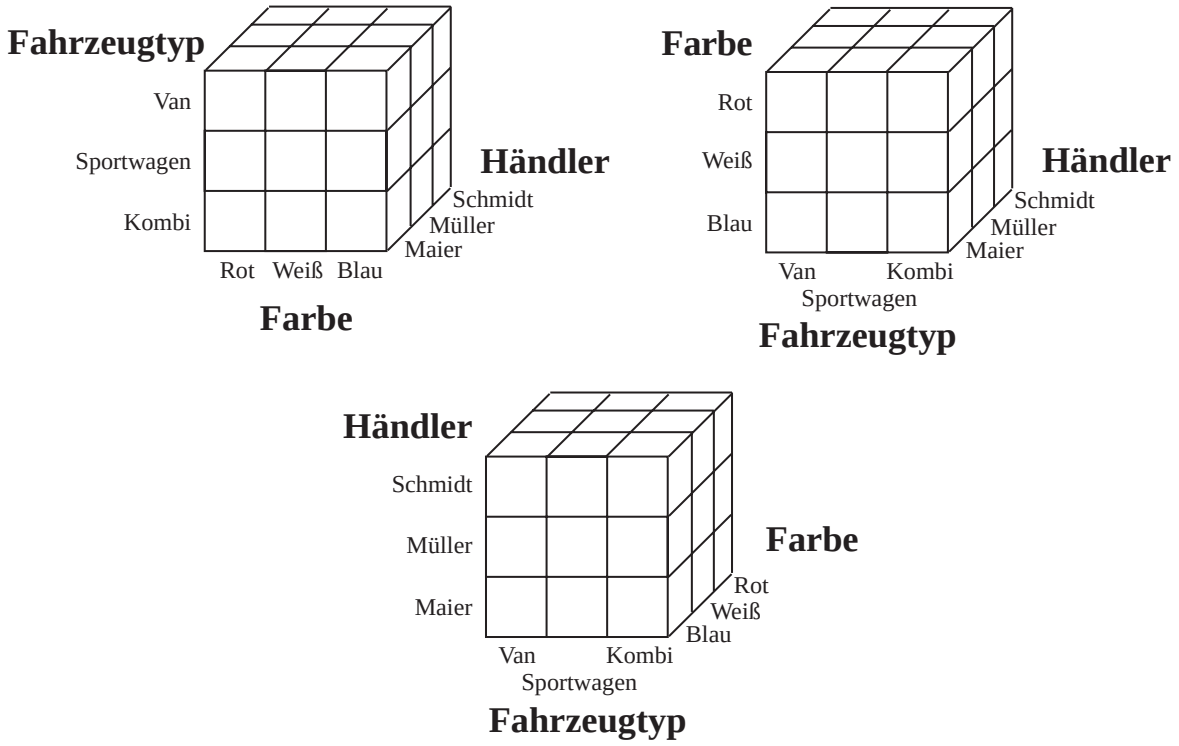


Abbildung 8.4: Rotationen des Großhändler data cubes

Die Drehung der Daten erlaubt es jedem Benutzer, sich die Daten so darstellen zu lassen wie es für seine Bedürfnisse am geeignetsten ist.

Die Anzahl der, durch Drehung erreichbaren, Sichten der Daten, steigt exponentiell mit der Anzahl der Dimensionen der Daten. Bei drei Dimensionen existieren sechs Sichten, und bei fünf Dimensionen existieren 120 Sichten, also allgemein bei n Dimensionen n! Sichten.

Ranging oder data dicing

Das Einschränken einzelner Dimensionen auf bestimmte *positions* wird als *ranging* oder *data dicing* bezeichnet. Diese Operation dient zur Einschränkung der angezeigten Daten auf die, die für den Benutzer von Interesse sind. Ein Beispiel sieht man in Abb. 8.5.

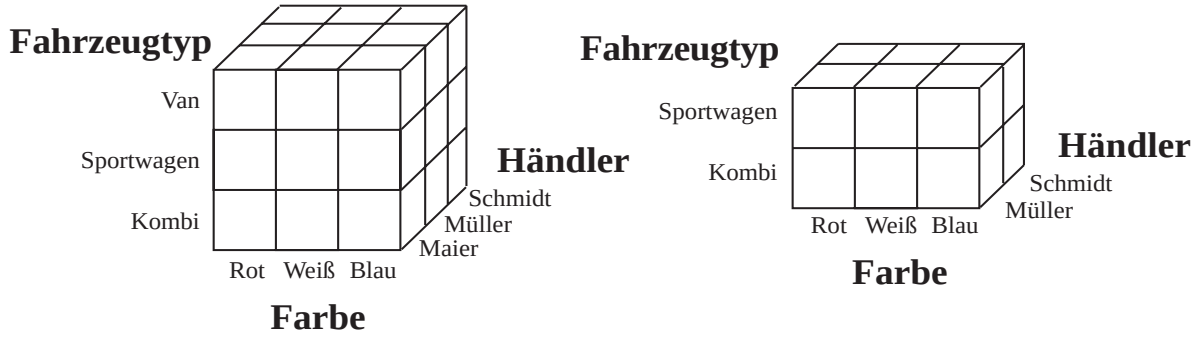


Abbildung 8.5: Ranging des Großhändler data cubes

Auf den resultierenden data cube können, wie auf den Ausgangswürfel, alle Operationen angewandt werden.

Hierarchien, roll-up und drill-down

Für jede Dimension kann eine Hierarchie definiert werden. Mit einer Hierarchie können mehrere *positions* einer Dimension zu einer neuen *position* zusammengefaßt werden, die dann an Stelle der bisherigen *positions* steht. Die Hierarchien können mehrere Ebenen umfassen. Eine höhere Hierarchiestufe bedeutet dabei einen höheren Aggregationsgrad der dargestellten Daten. Je tiefer man in die Hierarchie hinabsteigt, desto detaillierter werden die Informationen. Das auf und ab innerhalb einer Hierarchie wird als *roll-up* und *drill-down* bezeichnet.

Im Großhändlerbeispiel können die Händler zum Beispiel zu Regionen zusammengefasst werden. *Einfache Hierarchien* sind dadurch gekennzeichnet, daß jede *position* einer Dimension zu genau einer *position* der darüberliegenden Hierarchiestufe zusammengefasst werden kann. Ein Beispiel für eine einfache Hierarchie sieht man in Abb. 8.6.

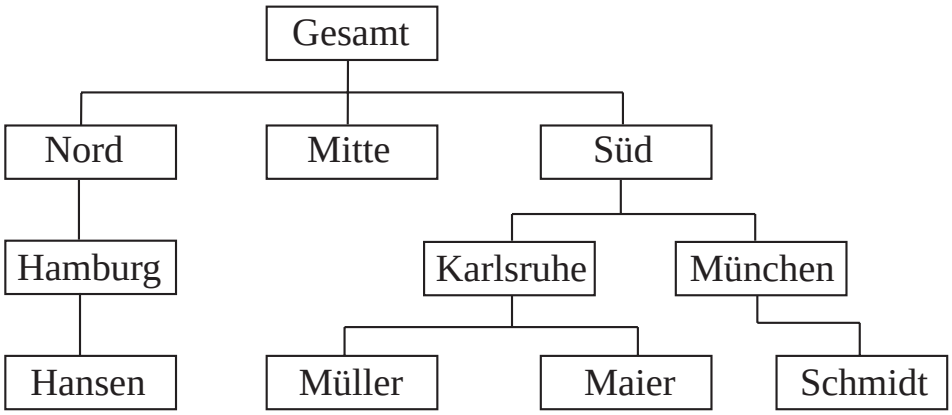


Abbildung 8.6: Einfache Hierarchie für Händler

Für bestimmte Daten wäre es wünschenswert, wenn man verschiedene Hierarchien auf einer Dimension aufbauen könnte. Im Händlerbeispiel hätte man die Autotypen einerseits gerne zu Herstellerfirmen zusammengefasst, andererseits auch zu verschiedenen Autoklassen (z.B. Geländewagen). Diese Problem läßt sich durch *mehrfache Hierarchien* lösen. In

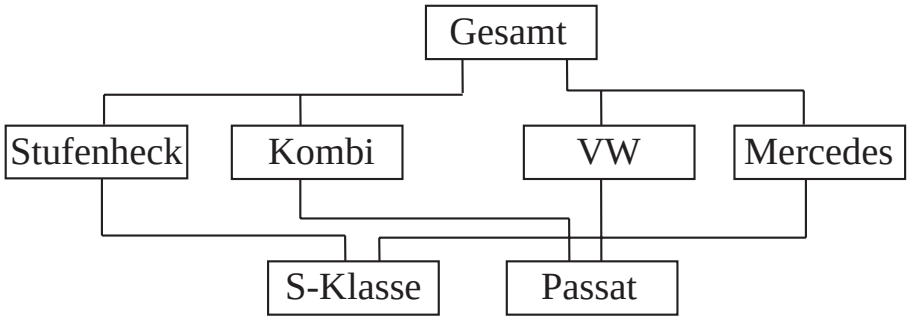


Abbildung 8.7: Eine mehrfache Hierarchie für Fahrzeugtypen

einer mehrfachen Hierarchie kann eine *position* zu verschiedenen *positions* einer höheren Hierarchiestufe gehören. Eine mehrfache Hierarchie sieht man in Abb. 8.7.

Für jede Hierarchie muß natürlich eine Aggregationsfunktion angegeben werden, mit Hilfe derer das OLAP-System die Daten beim Übergang zu einer höheren Hierarchiestufe aggregieren kann.

Die Dimension Zeit

Für viele Analyse ist es nötig den Verlauf von Geschäftsdaten über der Zeit zu beobachten. Die Zeit ist in nahezu allen zu analysierenden Daten enthalten. Die Zeit hat eine natürliche Einteilung in Tage, Wochen, Monate und Jahre. Ein Unternehmen betrachtet jedoch möglicherweise ein Geschäftsjahr und Quartale, die nicht mit einem Kalenderjahr übereinstimmen. Weil die Zeit also besonders wichtig ist, wird sie in einigen OLAP-Systemen auch speziell unterstützt. Es gibt zwei Arten, wie ein OLAP-System die Zeit unterstützen kann. Entweder wird die Zeit durch eine spezielle Zeit-Dimension oder durch einen speziellen Zeit-Datentyp modelliert.

Unterstützt das benutzte OLAP-System eine spezielle Zeit-Dimension, bedeutet dies für den Anwender, daß er für die Zeit eine Dimension benutzen kann, in der vordefinierte Hierarchien und Aggregierungsfunktionen existieren.

Unterstützt das benutzte OLAP-System jedoch einen Zeit-Datentyp, so wird für die Zeit keine eigene Dimension angelegt. Die zeitabhängigen Daten werden in einer *cell* gespeichert. Das heißt, daß in einer *cell* nun mehrere Werte stehen. Man könnte also in einer *cell* die täglichen Verkaufszahlen eines Produktes für die letzten zehn Jahre speichern. In [35] wird eine Reihe von Attributen zur Konfiguration des Zeit-Datentyps angegeben. Neben der Angabe von Hierarchien, Aggregationsfunktionen und Datentyp der gespeicherten Daten, muß auch die Liste der Daten die in der *cell* gespeichert werden sollen angegeben werden.

Beide Möglichkeiten haben den Vorteil, daß durch die vordefinierten, und für die eigenen Bedürfnisse konfigurierbaren, Funktionen viel Programmierarbeit beim Anwender gespart werden kann. Zusätzlich wird dem OLAP-System durch die vordefinierten Funktionen und Datentypen Wissen über die im System gespeicherten Daten mitgeteilt. Dieses kann, durch das vorhanden Wissen, Operationen optimieren und so die Antwortzeiten des Systems verbessern.

8.2.4 Algebra für mehrdimensionale Daten

In [36] wird eine Algebra für mehrdimensionale Datenbanken definiert. Die Funktionalität des mehrdimensionalen OLAP wurde an zwei Stellen erweitert. Erstens sollen Dimensionen und Werte in den *cells* gleich behandelt werden. Es muß also möglich sein, Gruppierungs- und Aggregierungsfunktionen nicht nur auf Dimensionen, sondern auch auf den Werten in den *cells*, zu definieren. Dies führt dazu, daß die Algebra einen Operator zur Verfügung stellt, der die Werte in den *cells* in eine Dimension umwandelt. Zweitens, sollten die Operatoren so definiert sein, daß eine Schachtelung dieser möglich ist, ähnlich wie in der relationalen Algebra. Bei den aktuellen mehrdimensionalen OLAP-Systemen hat der Benutzer einen Kubus, führt eine Operation auf diesem Kubus aus und erhält einen neuen Kubus auf den er erneut eine Operation anwenden kann. Hätte man wohldefinierte Operatoren, könnten diese geschachtelt werden. So könnte man komplexe Anfragen zusammensetzen, die optimiert und damit schneller ausgeführt werden könnten.

Die vorgestellten Operatoren sind minimal. Das bedeutet, keiner der Operatoren läßt sich durch einen, oder mehrere, der anderen darstellen.

Bei der Definition der Algebra wurde versucht, so nah wie möglich an der relationale Algebra zu bleiben. Zusätzlich sollten die definierten Operatoren in die relationale Algebra übersetzbar sein.

Das Datenmodell

In diesem Modell sind die Daten in einem oder mehreren data cubes gespeichert. Ein data cube hat folgende Eigenschaften: Er hat k Dimensionen, und eine Domäne aus der die Werte dieser Dimension sind. Die Elemente im data cube sind definiert durch eine Abbildung der einzelnen Domänen auf ein n-Tupel, eine 0 oder eine 1. Ist ein Element im Datenwürfel 0, so bedeudet das, daß dieser Wert in der Datenbasis nicht vorhanden ist. Hat ein Element den Wert 1, so ist diese Kombination in der Datenbank enthalten. Ein n-Tupel bedeutet, daß für diese Kombination zusätzliche Informationen verfügbar sind. Falls ein Element eines Datenwürfels eine 1 ist, enthält dieser keine n-Tupel, und umgekehrt. Den Händlerkubus mit Verkaufsdimension sieht man in Abb. 8.8.

Eine 1 bei Art=Van, Farbe=blau und Anzahl=6 bedeutet, daß diese Kombination existiert, also sechs blaue Vans verkauft wurden. Eine 0 bei Art=Van, Farbe=blau und Anzahl=5 bedeutet, daß keine fünf blauen Vans verkauft wurden.

Die Operatoren

Die folgenden Operatoren sind in der Algebra definiert:

- *push.* Der push-Operator erweitert alle Elemente eines data cubes, die nicht null sind, um den Wert der zugehörigen Dimension. Falls das Element eine eins ist, wird es in ein 1-Tupel umgewandelt. Eingabe für diesen Operator ist ein data cube und die hinzuzufügende Dimension. Dieser Operator dient zur Vorbereitung des data cubes auf eine Dimensionsreduzierung. Die übergebene Dimension wird jedoch noch nicht aus dem data cube entfernt.

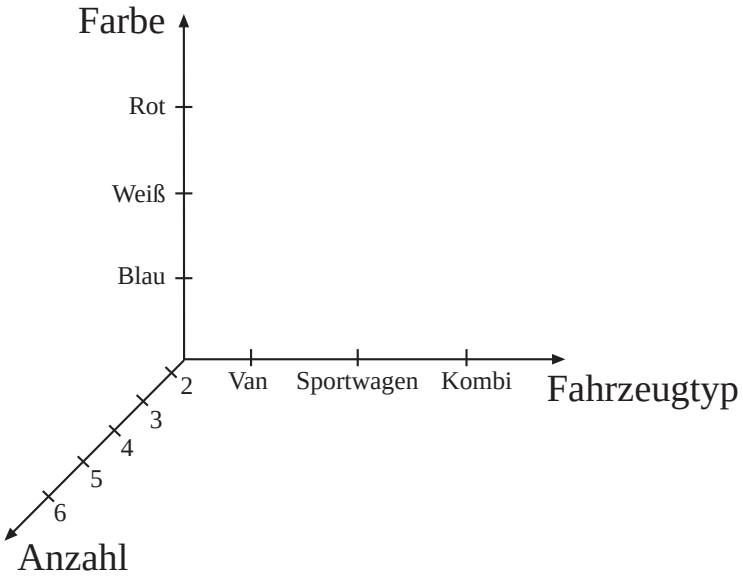


Abbildung 8.8: Data cube mit Verkaufsdimension

- *pull*. Der pull-Operator erzeugt eine neue Dimension. Dabei wird aus jedem n-Tupel des data cubes das k-te Element entfernt. Aus der Menge dieser Werte wird die Domäne der neuen Dimension bestimmt. Dieser Operator dient dazu Elemente des data cubes in Dimensionen umzuwandeln, die dann zu Gruppierungs- und Aggregationsfunktionen herangezogen werden können.

- *destroy*. Der destroy-Operator dient zum entfernen einer Dimension aus dem data cube. Dabei können nur Dimensionen entfernt werden die eine einelementige Domäne besitzen. Soll eine Dimension entfernt werden deren Domäne mehr Elemente enthält, so muß diese zuerst mit dem merge-Operator auf ein Element verkleinert werden.

- *restrict*. Der restrict-Operator entfernt aus einer Dimension alle *positions* die ein gegebenes Prädikat nicht erfüllen. Das Prädikat wird nicht auf jede *position* einzeln angewant, sondern auf auf alle *positions* gemeinsam. Damit sind zum Beispiel auch Fragen nach den fünf größten Werten einer Dimension beantwortbar.

- *join*. Der join-Operator dient zum Verknüpfen von Daten aus zwei data cubes. Dabei können mehrere Dimensionen gleichzeitig verknüpft werden, jedoch wird immer eine Dimension mit genau einer anderen Dimension verknüpft. Das Ergebnis der Verknüpfung, eines n-dimensionalen data cubes mit einem m-dimensionalen über k-Dimensionen, hat n+m-k Dimensionen. Alle Elemente mit dem gleichen Wert in der Verknüpfungsdimension werden zusammengefasst. Ist eines der Elemente gleich null, ist auch das Ergebnis null. Anschließend wird auf die beiden verknüpften Werte eine Funktion zur Berechenung des zugehörigen Elements im Ergebnis angewand.

- *merge*. Der merge-Operator dient zum Aggregieren von *positions* einer Dimension. Dazu muß einerseits angegeben werden welche *positions* zu einer neuen zusammengefasst werden sollen, andererseits muß noch eine Funktion angegeben werden die festlegt wie die Elemente des data cubes, die in das selbe Aggregat fallen, zu einem zusammengefasst werden.

Der merge-Operator gehört nicht zur minimalen Menge von Operatoren, da er durch einen join des Datenwürfles mit sich selbst beschrieben werden kann. Er wurde trotzdem in die Menge aufgenommen, da er im Gegensatz zum join-Operator, ein unärer Operator ist.

Beschreibung der OLAP-Funktionalität durch die Operatoren

Die oben beschriebenen Operationen mehrdimensionaler Datenanalyse lassen sich mit Hilfe der hier beschriebenen Operatoren darstellen:

- *ranging* oder *data-dicing*. Die ranging Operation wird direkt in den restrict-Operator umgesetzt.
- *roll-up*. Die roll-up Operation kann direkt in den merge-Operator umgewandelt werden.
- *drill-down*. Beim drill-down von einem aggregierten Wert auf die n Werte aus denen er besteht, ist es nötig die n unterliegenden Werte zu kennen. Sind beide data cubes, der mit den aggregierten Daten und der mit den detaillierten Daten, bekannt, muß ein join zwischen diesen beiden durchgeführt werden, um die ursprünglichen Daten zu bekommen.

In [36] wird auch eine Umsetzung der mehrdimensionalen Operatoren in SQL angegeben, für den Fall, daß eine relationale Datenbank zugrundeliegt.

8.3 OLAP auf der Basis mehrdimensionaler Datenbanken

Mehrdimensionale Datenbanken sind für den Einsatz im OLAP optimiert. Sie bieten eine direkte Unterstützung der im OLAP benötigten Funktionen. Dadurch sind sie bei dieser Art der Datenverarbeitung besonders schnell. Erreicht wird diese Optimierung der Datenbank durch spezielle Zugriffspfade. Die genaue Struktur und Funktionsweise der mehrdimensionalen Datenbanken wird von den Anbietern jedoch in der Regel nicht veröffentlicht.

8.4 OLAP auf der Basis relationaler Datenbanken

In diesem Abschnitt wird untersucht, wie sich relationale Datenbanken zur analytischen Datenverarbeitung eignen. Im ersten Abschnitt wird die Struktur von relationalen Systemen vorgestellt. Die Aggregations- und Gruppierungsfunktionen von SQL werden im zweiten Abschnitt vorgestellt. Da diese Funktionen von SQL nicht mächtig sind, haben einige Firmen proprietäre Erweiterungen in ihre Produkte eingebaut. Diese Erweiterungen werden im letzten Abschnitt vorgestellt.

8.4.1 Struktur relationaler Datenbankysteme

Eine anschauliche Darstellung von Daten in einem relationalen System ist die Tabelle. Diese Tabelle wird Relation genannt. Dabei entspricht ein Datensatz, der Tupel genannt wird, einer Zeile in der Tabelle. Um Redundanz in einer relationalen Datenbank zu vermeiden, werden sogenannte Normalformen definiert. Ausgehend von einer Relation die alle Informationen enthält, wird diese immer weiter zerlegt. Dabei werden Informationen, die für mehrere Datensätze gelten, in einer neuen Relation zusammengefasst. Dadurch wird erreicht, daß jede Information nur einmal in der Datenbank enthalten ist. Um jedoch wieder die ursprüngliche Information zu erhalten, müssen Informationen aus mehrere Tabellen zusammengesetzt werden. Diese sogenannte Verbindungsoperation (join) die dazu dient, Informationen aus verschiedenen Tabellen zu verknüpfen, ist sehr aufwendig.

Die Art, in der die Daten in einem relationalen System gespeichert sind, führt dazu, daß einzelne Datensätze schnell gefunden und bearbeitet werden können. Die Verarbeitung großer Datenmengen aus mehreren Tabellen benötigt jedoch viele Verbindungsoperationen, die sehr lange dauern. Relationale Systeme wurden ursprünglich für das Online Transaction Processing (OLTP) entwickelt und auch für diese Art der Datenverarbeitung optimiert.

8.4.2 SQL-Funktionen für OLAP

Die Gruppierung und die Aggregierung von Daten sind zwei der wichtigsten Aufgaben in OLAP-Anwendungen. SQL stellt für diese Aufgaben mehrere Operatoren bereit. Für die Gruppierung den `group by` Operator, und als Aggregationsfunktionen `count()`, `sum()`, `min()`, `max()` und `avg()`. Bei der Aggregation wird der entsprechende Operator auf die gesamte Relation angewand. Der `group by` Operator teilt die Zeilen einer Tabelle in mehrere disjukte Teilmengen ein, wobei auf jede der Teilmengen anschließend ein Aggregationsoperator angewand wird. Das Ergebnis der Aggregationsoperationen ist ohne Gruppierung nulldimensional, und mit einer vorgeschaltenen Gruppierung eindimensional.

Drei typische Funktionen der Datenanalyse lassen sich jedoch nicht direkt oder nur umständlich durchführen. Mit dem `group by` Operator ist es zum Beispiel nicht möglich, Histogramme direkt zu erzeugen. Eine Aggregation über berechnete Kategorien wird als Histogramm bezeichnet. Der `group by` Operator in SQL läßt jedoch nur Aggregationen über vorhandenen Feldern zu. Um diese Einschränkung zu umgehen muß zuerst eine Zwischenrelation mit den berechneten Werten erstellt werden. Auf diese Zwischenrelation kann nun die Gruppierung angewand werden, weil die berechneten Kategorien, über die gruppiert werden soll, nun als Feld vorliegen. Weitere häufig benötigte Funktionen sind das *roll-up* und das *drill-down*. Beim *roll-up* werden die Daten schrittweise zu immer größeren Einheiten, die einen Überblick über die Daten geben, mit Hilfe von Aggregationsfunktionen, zusammengefasst. Dabei muß in jedem Schritt eine Gruppierung auf der aktuellen Relation, mit berecheneten Kategorien, die sich aus der Definition der benutzten Hierarchie ergeben, bestimmt werden. Das *drill-down* ist das dem *roll-up* entgegengesetzte Vorgehen. Da es keine Möglichkeit gibt, ein Aggregat wieder in seine Teile zu zerlegen, müssen beim *drill-down* alle unteren Hierarchiestufen, mit ihren Gruppierungen und Aggregaten, neu gebildet werden. Eine Möglichkeit die Neuberechnung der Gruppierungen

und Aggregate zu umgehen ist die Vorausberechnung derselben. Dabei ist zu beachten, daß auch für diese Vorausberechnung eine Menge Gruppierungs- und Aggregationsoperationen vorab durchzuführen sind. Zusätzlich muß eine große Anzahl Datensätze in die Datenbank eingefügt werden, um die vorausberechneten Aggregate zu speichern.

8.4.3 Erweiterungen von SQL für OLAP

Viele Hersteller von relationalen Datenbanken sind sich der Schwächen von Gruppierung und Aggregation bewußt, und erweitern SQL um eigene, mächtigere Funktionen.

Illustra

Illustra bietet die Möglichkeit neue, selbst definierte und programmierte, Aggregationsfunktionen in das System einzufügen. Dazu wird ein Programm mit drei callback Routinen dem System hinzugefügt.

- `Init (&handle)`: Reserviert den Speicher für handle, und initialisiert die Aggregationsfunktion.
- `Iter (&handle, value)`: Fügt den aktuellen Wert zum Aggregat hinzu.
- `value = Final(&handle)`: Berechnet den Wert des Aggregats mit Hilfe der Daten die im handle gespeichert sind, und gibt das Ergebnis zurück. Anschließend wird der Speicher wieder freigegeben.

Um den Mittelwert einer Menge von Datensätzen zu berechnen, muß im handle die Anzahl und die Summe der Werte gespeichert werden. Beide Werte werden bei der Initialisierung auf null gesetzt. Bei jedem Aufruf von `Iter()` wird die Anzahl inkrementiert, und der aktuelle Wert zur Summe addiert. Beim Aufruf von `Final()` wird die Summe durch die Anzahl der Werte geteilt, und das Ergebnis zurückgegeben.

Die so definierten Aggregationsfunktionen können auch zusammen mit einer Gruppierung verwendet werden.

Red Brick

Red Brick definiert einige Aggregationsfunktionen, welche den `group by`-Operator erweitern.

- `Rank (expression)`: Berechnet für jeden Datensatz der Tabelle `expression` und gibt den Rang des Ergebnisses zurück. Falls n Datensätze in der Menge sind, und der aktulle Wert der größte ist, gibt diese Funktion n zurück.
- `N_tile (expression, n)`: Der Wertebereich von `expression` über die gesamte Tabelle wird berechnet, und anschließend in n Bereiche, mit annähernd gleicher Elementanzahl, geteilt. Die Funktion gibt den Rang des Bereichs an, in dem der aktuelle Datensatz liegt.

- `Raion_To_Total(expression)`: Berechnet die Summe der Ergebnisse von `expression` für alle Datensätze und teilte dieses durch die Anzahl der Datensätze.
- `Cummulative (expression)`: Addiert alle bisherigen Werte in einer geordneten Liste.
- `Running_Sum (expression, n)`: Addiert die letzten n Werte einer geordneten Liste.
- `Running_Average (expresion, n)`: Bildet den Durchschnitt der letzten n Werte.

Der data cube Operator

Microsoft erweitert SQL an mehreren Stellen [37]. Zum Beispiel werden berechnete Kategorien im `group by`-Operator erlaubt. Die nächste Erweiterung betrifft die Attribute die im `select` Teil einer SQL-Anfrage stehen dürfen. Normalerweise sind nur Attribute erlaubt nach denen gruppiert wurde oder solche auf die eine Aggregationsfunktion angewand wird. Dies sind aber zu starke Einschränkungen. Falls ein Attribut funktional von einem der Gruppierungsattribute abhängt, ist dieses für die gesamte Gruppe eindeutig definiert, und kann als "Verschönerung" des Ergebnisses mit ausgegeben werden. Diese Erweiterungen der SQL-Syntax sind vom nachfolgend definierten Cube Operator unabhängig.

Der data cube Operator ist die Erweiterung des `group by`-Operators auf mehrere Dimensionen. Der data cube Operator erzeugt alle Aggregate einer gegebenen Tabelle. Dazu wird die Potenzmenge aller Spalten, über die gruppiert werden soll, gebildet. Für jedes Element dieser Potenzmenge, wird das Aggregat über die Spalten, die dieses Element enthält, bestimmt. Der Inhalt jeder Spalte, die nach `group by` folgt, nach der aber nicht gruppiert wurde, wird durch ein `ALL` ersetzt. Für die Händlertabelle sieht das Ergebnis wie folgt aus:

Art	Farbe	Anzahl
Van	Blau	6
Van	Rot	5
Van	Weiß	4
Van	ALL	15
Sportwagen	Blau	3
Sportwagen	Rot	5
Sportwagen	Weiß	5
Sportwagen	ALL	13
Kombi	Blau	4
Kombi	Rot	3
Kombi	Weiß	2
Kombi	ALL	9
ALL	Blau	13
ALL	Rot	13
ALL	Weiß	11
ALL	ALL	37

Die Umwandlung der Händlertabelle in die Matrixdarstellung sieht man in Abb. 8.9.

Fahrzeugtyp

	Rot	Weiß	Blau	ALL
ALL	**13**	**11**	**13**	**37**
Van	**5**	**4**	**6**	**15**
Sportwagen	**5**	**5**	**3**	**13**
Kombi	**3**	**2**	**4**	**9**

Farbe

Abbildung 8.9: Die Händlertabelle nach Anwendung des data cube-Operators in Matrixdarstellung

Die zugehörende Anfrage ist die folgende:

```
SELECT Fahrzeugtyp, Farbe, SUM(Anzahl) AS Anzahl
FROM Verkäufe
GROUP BY Fahrzeugtyp, Farbe WITH CUBE;
```

Falls die Anzahl der verschiedenen Ausprägungen der n Spalten C_1, C_2,...,C_n ist, so ist die Größe der Ergebnistabelle $\prod_1^n (C_i + 1)$.

Jeder der `ALL`-Werte steht für die Menge über die das entsprechende Aggregat gebildet wurde. Ein `ALL` in der Spalte Art bedeutet, daß in der Anzahl dieser Zeile die Verkaufszahlen von Van, Sportwagen und Kombi enthalten sind.

Um das selbe Ergebnis in SQL zu erhalten ist die folgende Anfrage nötig:

```
SELECT Fahrzeugtyp, ALL, SUM(Anzahl)
FROM Verkäufe
GROUP BY Fahrzeugtyp
UNION
SELECT ALL, Farbe, SUM(Anzahl)
FROM Verkäufe
GROUP BY Farbe
UNION
SELECT Fahrzeugtyp, Farbe, SUM(Anzahl)
FROM Verkäufe
GROUP BY Fahrzeugtyp, Farbe
UNION
SELECT ALL, ALL, SUM(Anzahl)
FROM Verkäufe;
```

Die relationale Datenbank muß zur Auswertung dieser Anfrage vier Vereinigungen und vier Gruppierungsoperationen mit anschließender Aggregation bestimmen. Die meisten relationalen Datenbanken führen jede dieser Operationen explizit aus, da sie nicht über alle Anfragen gemeinsam optimieren können. Dadurch ist diese Anfrage sehr langsam. Die Vorteile des data cube Operators liegen in der leichteren Formulierung der Anfrage und der verbesserten Optimierungsmöglichkeit der relationalen Datenbank.

8.5 Vergleich und Bewertung

Die Benutzung mehrdimensionaler Datenbanken für OLAP hat mehrere Vorteile. Mehrdimensionale Datenbanken sind um Größenordnungen schneller bei der Analyse und Verarbeitung großer Datenmengen. Schon die Suche nach einem einzelnen Element ist in einer mehrdimensionalen Datenbank schneller als in einer relationalen Datenbank. Um in dem 10*10*10 Einträge großen Datenwürfel des Großhändlers, die Verkäufe eines bestimmten Händlers, für eine bestimmte Farbe, eines bestimmtes Modells zu bestimmen, müssen im relationalen System alle 1000 Datensätze durchsucht werden. Im mehrdimensionalen System hingegen müssen auf jeder der Dimensionen maximal 10 Einträge durchsucht werden. Auch Aggregierungsfunktionen lassen sich in mehrdimensionalen Datenbanken meist auf Zeilen- oder Spaltenoperationen zurückführen, und können somit schnell ausgeführt werden.

Relationale Datenbanken sind für OLAP nur bedingt geeignet. Werden die Daten in normalisierter Form in der Datenbank gespeichert, müssen diese zuerst miteinander verbunden werden, um die Zusammenhänge zwischen den einzelnen Relationen wiederherzustellen. Da Verbindungsoperationen jedes relationale System stark beanspruchen, könnte man diese Daten auch nicht normalisiert in der Datenbank ablegen. Dies vergrößert die Datenbank, aufgrund der vohandenen Redundanz, jedoch beträchtlich, was starke Leistungseinbußen zur Folge hat. So muß bei Suchvorgängen eine, um das mehrfache, größere Datenmenge durchsucht werden. Mit steigender Datenbankgröße muß immer öfter auf die Festplatte zugegriffen werden, wodurch die Anfragen immer länger dauern. Diese Geschwindigkeitsnachteile relationaler Systeme können mit Hilfe von Zugriffspfaden teilweise beseitigt werden. Jedoch können Optimierungen nicht alle Anwendungen berücksichtigen, so daß die Möglichkeit Spontananfragen zu stellen, und schnell beantwortet zu bekommen, eingeschränkt ist. Es ist auch zu beachten, daß Indexe viel Speicher benötigen, oft sind sie zusammen größer als die eigentlichen Daten. Da diese Möglichkeiten gegeläufige Effekte haben, besteht das Problem darin, den für die eigene Anwendung idealen Weg zwischen normalisierten und nicht normalisierten Relationen mit zugehörigen Zugriffspfaden zu finden.

Relationale Datenbanksysteme können keine mehrdimensionalen Daten verwalten. Dies ist nur über den Umweg der Aufspaltung der Daten in mehrere Tabellen möglich. Durch diese Normalisierung der Daten in einem relationalen System, sind viele Informationen, die gemeinsam betrachtet werden sollen auf verschiedene Tabellen verteilt. Für einen Endanwender ist es schwierig, selbst wenn er die Zusammenhänge zwischen den Tabellen kennt, komplexe Anfragen über mehrere Tabellen zu stellen. Um auch einem Anwender der nicht über große SQL Kenntnisse verfügt Zugang zur Datenbank zu verschaffen, muß eine Anwendung entwickelt werden, die die Anfragen für den Anwender übernimmt. Um die Ausgaben der relationalen Datenbank in die, für die Analyse typische, Darstellung, nämlich einen Datenwürfel, zu überführen, und die üblichen Analysefunktionen zu bieten, muß viel Programmierarbeit geleistet werden.

Relationale Datenbanksysteme und SQL als Anfragesprache für relationale Datenbanksysteme, bieten keine der OLAP Funktionen, wie drill-down, slicing und dicing, direkt an. Diese Funktionen müssen durch komplexe, meist langlaufende, Anfragen simuliert werden. SQL unterstützt nur wenige Aggregationsfunktionen (SUM, AVG, MIN, MAX,...),

und bietet keine Möglichkeit, bestimmte Anfragen, wie zum Beispiel die nach den zehn größten Autohändler, zu formulieren. Einige Firmen haben Erweiterungen von SQL vorgeschlagen um diese Einschränkungen zu beseitigen, diese sind jedoch firmenspezifisch, und nicht in den SQL-Standard übernommen worden.

Die angeführten Nachteile relationaler Datenbanksysteme im OLAP-Bereich bedeuten jedoch nicht, daß mehrdimensionale Datenbanken den relationalen Datenbanken in ihrem angestammten Bereich, den OLTP-Anwendungen, Konkurrenz machen können. Mehrdimensionale Datenbanken haben Schwächen bei der Speicherung und Verarbeitung nicht mehrdimensionaler Daten. Die relationale Datenbank für OLTP und die mehrdimensionale Datenbank für OLAP sollte deshalb getrennt werden. Zu beachten sind noch die typischen Lastverteilungen von OLTP und OLAP Anwendungen. OLTP-Systeme haben im Normalfall eine gleichmäßig hohe Last über lange Zeiträume, wohingegen die Last von OLAP Anwendungen relativ niedrig ist, jedoch bei Anfragen der Benutzer sprunghaft auf das Maximum ansteigt. Wenn beide Anwendungen auf dem selben System betrieben werden, wäre die OLTP-Anwendung stark von der OLAP-Anwendung behindert. Deshalb sollte die OLAP und die OLTP Anwendung auf verschiedenen Rechnern betrieben werden.

8.6 Zusammenfassung

Im ersten Abschnitt wurden die Anforderungen an OLAP-Systeme vorgestellt. Dabei war vor allem die intuitive Bedienung und die schnelle Beantwortung aller Anfragen durch das System wichtig. Zu den Eigenschaften von OLAP-Anwendungen gehören die großen zu bearbeitenden Datenmengen, und vielfältige Analysefunktionen. Im nächsten Abschnitt wurde die Eignung von mehrdimensionalen Datenbanken für OLAP-Anwendungen untersucht. Dabei zeigte sich, daß diese in ganz natürlicher Weise die OLAP-Anwendungen unterstützten. Bei der Untersuchung relationalen Systemen für OLAP-Anwendungen zeigte sich, daß diese nicht besonders gut für OLAP-Anwendungen geeignet sind. Gößtes Problem relationaler Datenbanken ist die mangelhafte Möglichkeit typische Analysefunktionen natürlich zu formulieren. Meist müssen diese Anfragen umständlich formuliert werden, und haben lange Laufzeiten, da die Art der Datenspeicherung im relationalen System und die vom Benutzer gewünschte Ausgabe der Daten unterschiedlich sind. Die Laufzeiten relationaler Systeme im Vergleich zu mehrdimensionalen liegt bei OLAP-Anwendungen meist um Gößenordnungen höher.

Relationale Systeme sind im OLTP-Bereich nicht zu ersetzen, haben jedoch Schwächen bei OLAP-Anwendungen. Mehrdimensionale Datenbanken sind jedoch auf natürliche Weise für OLAP-Anwendungen geeignet. Die Verarbeitung der immer stärker anwachsenden Datenmengen in Unternehmen, ist nur möglich, wenn für jede der benötigten Anwendungen spezialisierte Systeme bereitstehen. Der Einsatz von mehrdimensionalen Datenbanken im OLAP-Bereich ist deshalb unumgänglich.

Kapitel 9

Optimierungsmethoden für OLAP

Stefan Weitland

Kurzfassung *Dieses Kapitel befaßt sich mit den verschiedenen Wegen zur Implementierung von OLAP–Servern. Zuerst werden die Unterschiede zwischen OLTP und OLAP diskutiert. Im weiteren Verlauf wird eine Gitternetznotation vorgestellt, die es erleichtert, die Anforderungen an einen OLAP–Server zu veranschaulichen. Anhand dieses Gitternetzes kann man die Implementierung der Datenstruktur des OLAP–Servers strukturieren. Im weiteren Verlauf wird mit den vorgestellten Algorithmen „PipeSort“ und „HashSort“ zwei Verfahren vorgestellt, die die Implementierungsreihenfolge der Anfragen optimieren. Als Grundlage für die beiden Algorithmen werden fünf Optimierungsmethoden vorgestellt. Beide Algorithmen versuchen die Optimierungsmethoden so zu kombinieren, daß die Implementierung der Anfragen besonders effektiv ist.*

9.1 Einleitung

OLAP ist die Abkürzung für on-line analytical processing. Dieser Begriff bezeichnet einen Anwendungsbereich für Datenbanken, in dem die Anforderungen an die Anfrage- und Antworteigenschaften im Vergleich zum bisherigen Prinzip der transaktionsorientierten Verarbeitung (OLTP) sehr hoch sind. Ein Datenbanksystem eines heutigen Unternehmens besteht aus einer großen Anzahl von Servern und einer noch größeren Anzahl von Clients. Die Daten sind hierbei auf die speziellen Server der einzelnen Unternehmensbereiche verteilt. Dieser hohe Grad an verteilter Datenhaltung ließ früh die bisherigen Auswertungsmöglichkeiten von Datenbanken an ihre Grenzen stoßen. Ein Grund dafür sind die erhöhten Anforderungen an die Analyse der Datenbestände. Beispielsweise muß ein Manager eines Unternehmens zur Planung einer neuen Verkaufsstrategie eines Produktes sofort auf eine Analyse der aktuellsten Daten zu diesem Produkt zugreifen können. Dabei stießen bisherige Anfragemöglichkeiten schnell an ihre Grenzen.

9.2 Prinzipien der analytischen Verarbeitung

Bisher verwendete OLTP Anwendungen unterscheiden sich stark von OLAP–Anwendungen [33]. OLTP Anwendungen bestehen aus einer großen Anzahl relativ einfacher Transaktionen. Diese Transaktionen verändern meistens nur eine geringe Anzahl von Datensätzen in einzelnen Tabellen, deren Beziehungen untereinander relativ einfach sind. Ein typisches Beispiel für eine OLTP Anwendung ist das Aufnehmen einer Bestellung eines neuen Kunden. Hierbei müssen nur in zwei verschiedenen Tabellen Daten eingefügt werden: in die Kundentabelle und in die Tabelle für die Bestellungen.

OLAP–Anwendungen greifen auf große Datenbanken mit komplexen Anfragen zu. Bei diesen Zugriffen müssen große Mengen von Daten berechnet, analysiert und verschoben werden. Eine typische OLAP–Anfrage läßt sich anhand der folgenden Punkte charakterisieren:

- Zugriff auf eine große Menge von Daten (historische und aktuelle Daten)
- Analyse der Beziehungen verschiedener Geschäftselemente (z.B. Verkäufe, Produkte, Regionen)
- Berechnung von Daten (z.B. Verkaufsvolumen, Ausgaben)
- Vergleich berechneter Daten über verschiedene Zeiträume
- Präsentation der Daten in verschiedenen Darstellungen (z.B. Diagrammen)
- Komplexe Vorausberechnungen und Prognosen
- Schnelle Antwortzeit

9.3 Datenbanken für OLAP

OLAP–Server benutzen mehrdimensionale Strukturen, um Daten und deren Beziehungen zu speichern. Diese mehrdimensionale Struktur kann am besten durch einen Würfel wiedergegeben werden. Jede Seite des Würfels stellt eine Dimension dar (Abbildung 9.1).

Jede Dimension repräsentiert eine Kategorie wie z.B. Produkttyp, Region, Verkaufswege und Zeit. Jede Zelle innerhalb dieser mehrdimensionalen Struktur enthält berechnete Daten, die sich auf Elemente entlang jeder Dimension beziehen. Mehrdimensionale Datenbanken sind kompakt und leicht zu verstehende Gebilde um Daten zu visualisieren und zu manipulieren, die komplex miteinander verbunden sind.

Zusätzlich zu den Standardoperationen von Datenbanken unterstützen OLAP–Server weitere Operationen:

A. Verallgemeinerung (Roll-Up) Diese Operation verdichtet die berechneten Daten. Dies kann einfaches Zusammenzählen bedeuten oder das Berechnen komplexer Ausdrücke kompliziert verbundener Daten. Als Beispiel können Verkäufe eines Produktes bezogen auf einen Landkreis berechnet werden und dann bezogen auf das Bundesland durch das Zusammenzählen der Landkreise.

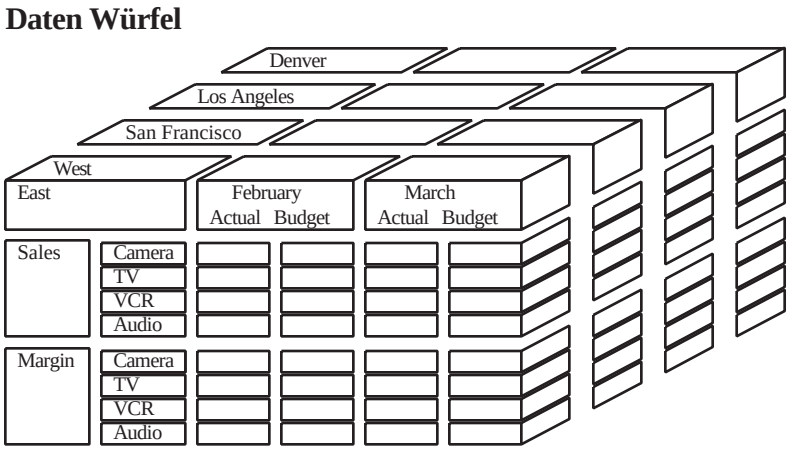


Abbildung 9.1: Beispiel für ein Datenwürfel

B. Verfeinerung (Drill-Down) OLAP–Server können auch in die andere Richtung gehen. Dies bedeutet die Berechnung detaillierterer Darstellungen aggregierter Daten. Verfeinerung und Drill-Down sind komplementäre Operationen.

C. Schneiden und Würfeln (Slicing and Dicing) Slicing and Dicing bezeichnet die Fähigkeit, Daten unter verschiedenen Aspekten oder Gesichtspunkten zu betrachten. Beispielsweise kann man alle Verkäufe nach Verkaufsweg innerhalb einer Region betrachten oder alle Verkäufe nach Verkaufsweg für jedes Produkt. Um Trends zu analysieren oder Schemata zu entdecken, wird diese Operation hauptsächlich entlang der Zeitachse durchgeführt.

D. Andere Eigenschaften OLAP–Server können mehrdimensionale Datenbanken komprimiert speichern. Dies wird durch die dynamische Auswahl des Hintergrundspeichers und durch Kompressionstechniken erreicht. Zudem werden Datenbereiche je nach Dichte der darin enthaltenen Daten separat gespeichert.

Die wichtigste Eigenschaft eines OLAP–Servers im Unterschied zu anderen Servern ist die logische Organisation der Daten in mehrere Dimensionen. Dies ermöglicht Benutzern schnell und einfach komplexe Datenstrukturen zu analysieren. Physikalisch ist die Datenbank so organisiert, daß über mehrere Dimensionen verbundene Daten schnell gefunden werden.

9.4 OLAP auf der Basis relationaler Datenbanken

Relationale Datenbanksysteme und die dazugehörigen Applikationen sind für OLAP–Anfragen meist zu stark eingeschränkt [33].

- Für ein RDBMS bedeutet die Darstellung einer mehrdimensionalen Datenstruktur die Projektion dieser auf eine zweidimensionale Struktur. Für die Projektion

einer dreidimensionalen Struktur bedeutet es z.B., diese Struktur in eine Vielzahl von Schichten zu zerlegen und durch entsprechende Beziehungen zu verknüpfen. Die Rekonstruktion der dreidimensionalen Struktur gestaltet sich kompliziert und ressourcenintensiv.

- Für den Endbenutzer ist es schwer die komplizierte zweidimensionale Struktur einer mehrdimensionalen Datenbank zu verstehen. Um Anfragen zu entwerfen, muß er wissen welchen Daten in welchen Tabellen gespeichert sind und wie sie untereinander in Beziehung stehen.
- Bei mehreren Benutzern und mehreren OLAP–Anfragen kann das RDBMS durch den Zugriff auf sehr große und kompliziert verbundene Datenbestände schnell an seine Leistungsgrenze stoßen. Dies führte zu dem Versuch die Daten anders zu speichern.

In OLTP–Systemen werden die Daten in der Regel normalisiert gespeichert (Abbildung 9.2). Diese Art der Speicherung ist für den Endbenutzer schwer verständlich und führt bei komplizierten Anfragen zu Leistungsverlusten, da mehrere Verbindungen der Relationen über Schlüssel viel Ressourcen beanspruchen. Deshalb wird von den meisten Datenbankherstellern vorgeschlagen, daß zur Datenanalyse die Tabellen denormalisiert gespeichert werden.

Table 1 — **1,000,000 Rows**

Sales Office	Product ID	Sales
BOS	1	1,000,000
SF	1	300,000
SF	2	500,000

Table 2 — **1,000 Rows**

Product ID	Description
1	Widget
1	Super Widget

Table 3 — **100 Rows**

Sales Office	District
BOS	A
SF	B

Table 4 — **10 Rows**

District	Region
A	East
B	West

Abbildung 9.2: Normalisierte Daten

Die Speicherung in denormalisierten Tabellen hat den Vorteil, daß die Verbindungen über Relationen über Schlüssel nicht mehr vorgenommen werden müssen, und somit Zeit gespart wird (Abbildung 9.3).

Sales Office	Product ID	Description	Sales	District	Region
BOS	1	Widget	1,000,000	A	East
SF	1	Widget	300,000	B	West
SF	2	Super Widget	500,000	B	West

Abbildung 9.3: Denormalisierte Daten

In Bezug auf die Systemressourcen entstehen durch die Denormalisierung große Tabellen, da Daten redundant in jedem Datensatz gespeichert werden. Dadurch sind OLAP–Anfragen dazu gezwungen, sehr große Datenbanken abzusuchen, was ebenso aufwendig

sein kann wie die entsprechende Anfrage auf die normalisierte Datenbank. Da denormalisierte Datenbanken sehr groß sind, erfordern diese bei Anfragen eine sehr große Anzahl von Zugriffen auf den Hintergrundspeicher. Zudem benötigt die Datensuche viel Speicherleistung was wiederum andere Anwendungen behindert.

Für den Benutzer ist es schwer, sich zwischen der denormalisierten- oder normalisierten relationalen Lösung zu entscheiden, da die Leistungen bezüglich der Anfragezeiten nahezu gleich sind. Wird die Datenbank normalisiert gespeichert, dann geht Leistung bei komplizierten Join–Anfragen verloren. Speichert der Benutzer die Daten aber denormalisiert, dann muß er mit extrem großen Datenbanken umgehen, die wiederum die Leistung mindern.

Relationale Datenbanksysteme haben keine OLAP–spezifischen Operation wie „Roll-Up“, „Drill-Down“ oder „Slicing and Dicing“ eingebaut. Es steht meist nur eine begrenzte Anzahl von vordefinierten Operationen zur Verfügung. Diese funktionale Einschränkung wird beispielsweise anhand der SQL–Operationen SUM und AVG ersichtlich. Es besteht nur die Möglichkeit diese Operationen auf mehrere Datensätze anzuwenden, aber nicht auf die Werte innerhalb eines Datensatzes. Des weiteren ist ein Benutzer, der eine analytische Anfrage entwerfen will, auf aufwendig entwickelte Anfrageoberflächen angewiesen. Für diese Anfrageoberflächen werden Spezialisten gebraucht, die von den Anwendern gewünschte Anfrage implementieren.

OLAP–Anfragen benötigen verdichtete Daten, die bei relationalen Datenbanksystemen nur in detaillierter Form vorliegen. Bei Anfragen stehen zwar die Operationen SUM und GROUP BY innerhalb einer SELECT–Anweisung zur Verfügung, aber diese ermöglichen keine Verfeinerung (Drill-Down) oder Verallgemeinerung (Roll-Up). Zudem gestalten sich solche Anfragen zeitintensiv und beanspruchen viel Ressourcen.

9.5 Implementierung eines Datenwürfels (data cube)

9.5.1 Gitterrahmenmodell

Das Gitterrahmenmodell wurde als Notation zur Darstellung der Abhängigkeiten der Anfragen untereinander entwickelt. Eine Anfrage kann man anhand ihrer Attribute, die in der GROUP-BY–Anweisung enthalten sind, spezifizieren. Hier wird die Notation festgelegt, daß jede Anfrage durch die Aufzählung ihrer Attribute in Klammern , wie zum Beispiel (ABCD), darstellbar ist.

A. Abhängigkeiten zwischen Anfragen: Man betrachte zwei Anfragen Q1 und Q2. Die Notation Q1 $\preceq$ Q2 bedeutet, daß Q1 beantwortet werden kann, indem die Ergebnisse von Q2 verwendet werden. Man sagt dann, daß Q1 abhängig von Q2 ist. Der Operator $\preceq$ definiert eine Halbordnung auf der Menge der Anfragen.

Um ein Gitter aufzubauen gilt:
1. $\preceq$ ist eine Halbordnung, und
2. es gibt ein maximales Element, eine Anfrage von der jede Anfrage abhängig ist.

B. Gitternotation: Man definiert ein Gitter durch eine Menge (von Anfragen) L und der Abhängigkeitsbeziehung $\preceq$ durch $< L, \preceq >$. Für die Elemente a und b eines Gitter $< L, \preceq >$, bedeutet a$\prec$b, daß a$\preceq$b und a$\neq$b.

Der Vorfahr, Nachfahr und der Nächste sind wie folgt definiert:

Vorfahr (ancestor)	anc(a)	$= \{b — a \preceq b\}$
Nachfahr (descendant)	desc(a)	$= \{b — b \preceq a\}$
Nächste (next)	next(a)	$= \{b — a \prec b$, es gibt kein c, $a \prec c$, $c \prec b\}$

Jedes Element des Gitters ist sein eigener Vorfahr und sein eigener Nachfahr. Der direkte Vorfahr von einem Element a gehört zu der Menge next(a).

C. Gitterdiagramm: Ein Gitter wird repräsentiert durch einen Graph (Abbildung 9.4) in welchem die Anfragen die Knoten sind und die Kanten zwischen a (unten) und b (oben) nur vorhanden sind, wenn b Element der Menge next(a) ist.

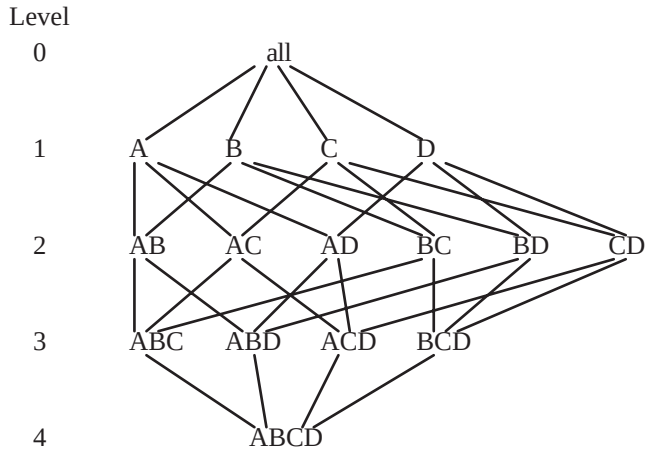


Abbildung 9.4: Beispiel für ein Gitter

D. Hierarchien: In der Realität bestehen die einzelnen Dimensionen aus mehr als einem Attribut. Ein einfaches Beispiel dafür ist die Unterteilung der Dimension Zeit in die Hierarchie: Tag, Monat, Jahr. Hierarchien sind sehr wichtig, da sie eine Basis für zwei typische OLAP–Operationen bilden:

- Drill-Down: Daten werden detaillierter betrachtet, z.B. werden die Verkäufe zuerst pro Jahr betrachtet und danach pro Monat.
- Roll-Up: Hier werden die Daten weniger detailliert betrachtet. Zuerst werden die Verkäufe pro Tag, später pro Monat betrachtet.

Durch das Vorhandensein von Hierarchien wird das Gittermodell komplizierter. Am Beispiel der Aufteilung der Dimension Zeit zeigt sich, das Hierarchien neue Abhängigkeiten in Anfragen einführen. Daten können bezogen auf das Jahr, den Monat oder den Tag berechnet und dargestellt werden.

Die Verwendung des Gitterrahmens hat mehrere Vorteile. Für die Implementierung definiert es eine eindeutige Struktur, an der abgelesen werden kann welche Anfrage materialisiert werden sollten. Benutzern erleichtert es die Erstellung der Anfragen. Durch die Darstellung als Graph wird die Festlegung der Reihenfolge der Materialisierung erleichtert.

9.6 Optimierungsmethoden

Die hier vorgestellten Optimierungsmethoden bilden die Grundlage für die später vorgestellten Implementierungsalgorithmen. Diese Optimierungsmethoden sind die Kriterien zur Bestimmung der Reihenfolge der Implementierung der einzelnen Anfragen.

1. Der kleinste Vorgänger (smallest-parent): Diese Optimierung versucht, aus der kleinsten zuvor berechneten Anfrage die geforderte Anfrage zu erzeugen, um nicht auf die Rohdaten zugreifen zu müssen. Im allgemeinen kann jede Anfrage aus anderen, bereits berechneten Anfragen erstellt werden. Als Beispiel zeigt Abbildung 9.4, daß Anfrage AB aus der Anfrage ABC berechnet werden kann.

2. Speicherergebnisse (Cache-results): Es wird versucht, durch das Halten der zuletzt berechneten Anfrage im Hauptspeicher so viele Anfragen wie möglich aus dieser zu berechnen, um die Zugriffe auf den Hintergrundspeicher so minimal wie möglich zu halten. Als Beispiel kann man an Abbildung 9.4 sehen, daß nach der Berechnung der Anfrage ABC versucht werden kann sofort die Anfrage AB zu berechnen, solange ABC noch im Hauptspeicher ist.

3. Amortisiertes Lesen (Amortize-scans): Um die Lesezugriffe auf den Hintergrundspeicher so minimal wie möglich zu halten, werden beim Berechnen der Anfrage ABCD alle abhängigen Anfragen (ABC, ABD, ACD, BCD) mit erzeugt.

4. Geteilte Sortierung (Share-sorts): Es wird versucht, bei bereits sortierten Anfragen deren Sortierung zu verwenden, um möglichst einfach alle identisch sortierten Anfragen zu erzeugen. Diese Optimierung wird bei der Implementierung durch die Sort-based Methode verwendet.

5. Geteilte Partitionierung (Share-partitions): Für Implementierungen durch die Hash-based Methode wird diese Optimierung verwendet, wenn die Hashtabelle zu groß für den Hauptspeicher ist. Die Daten werden partitioniert und die Aggregation wird für jede

Partition, die in den Hauptspeicher paßt, getrennt durchgeführt. Die Partitionierungskosten werden dadurch vermindert, daß die Partitionierung auf ein Attribut und auf alle Anfragen mit diesem Attribut angewandt werden kann.

Da bei OLAP–Datenbanken die zu aggregierenden Daten wesentlich größer sein können als der zur Verfügung stehende Hauptspeicher, wirken die oben aufgeführten Optimierungsmethoden möglicherweise einander entgegen. Um beispielsweise die Anfrage B zu berechnen, würde die erste Optimierungsmethode die Anfrage BC vor der Anfrage AB bevorzugen, wenn BC kleiner wäre, aber die zweite Optimierungsmethode würde die Anfrage AB bevorzugen, wenn diese noch im Hauptspeicher wäre und BC im Gegensatz dazu noch auf dem Hintergrungspeicher.

9.7 Implementierung basierend auf Sortierung

Als eine auf Sortierung basierende Methode zur Implementierung wird hier der PipeSort Algorithmus vorgestellt [39]. Dieser Algorithmus benutzt die Optimierung „share-sort“ (4), um alle Anfragen mit den gleichen Präfixen aus einer bereits erzeugten Anfrage zu berechnen. Dieses Vorgehen kann aber mit der Optimierung „smallest-parent“ (1) in Konflikt stehen, da der kleinste Vorgänger von AB, BDA sein könnte, versucht der Algorithmus PipeSort beide Methoden zu berücksichtigen. Um besonders gute Ergebnisse erzielen zu können, ist es wichtig, globale Planungen darüber vorzunehmen, in welcher Reihenfolge und welcher Sortierung die Anfragen erzeugt werden. Aus diesem Grund berücksichtigt PipeSort auch die Optimierungsmethoden „cache-results“ (2) und „amortize-scans“ (3).

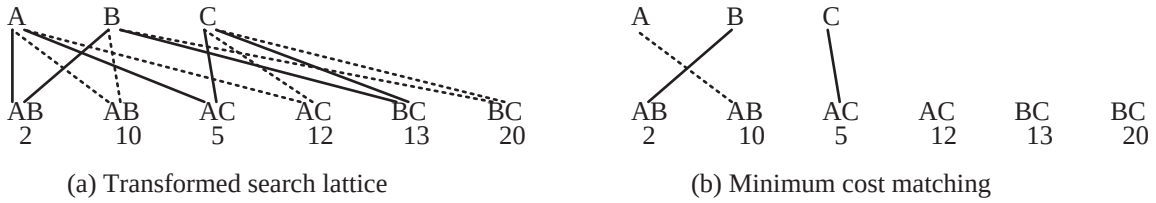
9.7.1 Der PipeSort-Algorithmus

Das Suchgitter: Das Suchgitter ist aufgebaut wie das bereits vorgestellte Gitterrahmenmodell. Zusätzlich werden zu jedem Knoten zwei Werte notiert. Der erste Wert gibt die Kosten A() für die Erzeugung der Anfrage j aus i an, unter der Bedingung, daß i bereits sortiert ist. Der zweite Wert gibt die Kosten S() für die Erzeugung der Anfrage j aus i an, unter der Bedingung, daß i noch sortiert werden muß. Das Ergebnis O ist ein Untergraph des Suchgitters, in dem jede Anfrage mit einem einzigen Vorfahr verbunden ist, aus dem er berechnet wird. Hat im Graph O j die selbe Sortierung wie i, dann sind die Kosten A() an dem Knoten j notiert, ansonsten die Kosten S(). Von einem Knoten i können mehrere Kanten S() wegführen. Das Ziel des Algorithmus ist es, einen Graphen O zu finden, der die kleinste Summe an Kosten hat (Abbildung 9.6).

Der Algorithmus: Der Algorithmus untersucht Ebene für Ebene, ausgehend von Ebene k = 0 bis Ebene k = N - 1, wobei N die Anzahl der Attribute ist. Für jede Ebene k findet der Algorithmus den besten Weg, diese Ebene aus der Ebene k + 1 zu berechnen.

Man verändert zuerst die Ebene k + 1 des originalen Suchgitters, indem man k zusätzliche Kopien jeder Anfrage in dieser Ebene macht. Somit hat jede Anfrage in der Ebene k + 1 genau k + 1 Knoten die der Anzahl der ausgehenden Kanten jeder Anfrage entsprechen.

Jede erzeugte Anfrage wird mit genau den selben Anfragen wie im Original verbunden. Die Kosten an dem original Knoten werden auf A() gesetzt, die der Kopien auf S(). Wir finden den minimal kostenden Graph durch den transformierten Graph (Abbildung 9.5).



(a) Transformed search lattice (b) Minimum cost matching

Abbildung 9.5: Erzeugung von Ebene 1 aus Ebene 2 Anfragen

Um das Vorgehen zu verdeutlichen wird in Abbildung 9.5 (a) gezeigt wie Ebene 1 Anfragen aus Ebene 2 Anfragen für ein drei Attribute Suchgitter erzeugt werden. Zunächst wird von jeder Anfrage eine Kopie gemacht. Durchgezogene Linien stellen die Kosten A() dar, während gestrichelte Linien die Kosten S() darstellen. Die Zahl unter den Anfragen ist die Summe der Kosten aller Kanten, die von der Anfrage wegführen. Im Graph für minimale Kosten (Abbildung 9.5 (b)) ist der Knoten A mit AB durch eine Kante S() verbunden und B durch eine Kante A(). Daher wird in Ebene 2 AB in der Attributreihenfolge BA erzeugt, so daß B ohne Sortierung und A durch erneute Sortierung erzeugt wird. Das gleiche gilt für C. Solange C mit AC durch eine Kante A() verbunden ist, wird AC in der Attributreihenfolge CA erzeugt. BC ist mit keinem Knoten verbunden und kann deshalb in beliebiger Sortierung erzeugt werden.

PipeSort:

(Eingabe: Suchgitter mit A() und S() als Kosten)
Für jede Ebene k = 0 bis N - 1 /* N ist die Anzahl der Attribute */
    /* erzeuge Ebene k durch Ebene k + 1 */
    Erstelle-Plan(k + 1 → k)
    Für jede Anfrage g in der Ebene k + 1
        Fixiere die Sortierung von g durch die Sortierung der Ebene k Anfrage
            die mit g durch eine Kante A() verbunden ist.

Erstelle-Plan (k + 1 → k)

Erzeuge k zusätzliche Kopien jeder Anfrage der Ebene k + 1
Verbinde jede Kopie einer Kante mit den gleichen Anfragen wie die original Anfrage
Notiere die Kosten A() zu der original Anfrage und die Kosten S() zu den Kopien.
Finde die minimale Zuordnung im transformierten Graph von Ebene k + 1 mit Ebene k

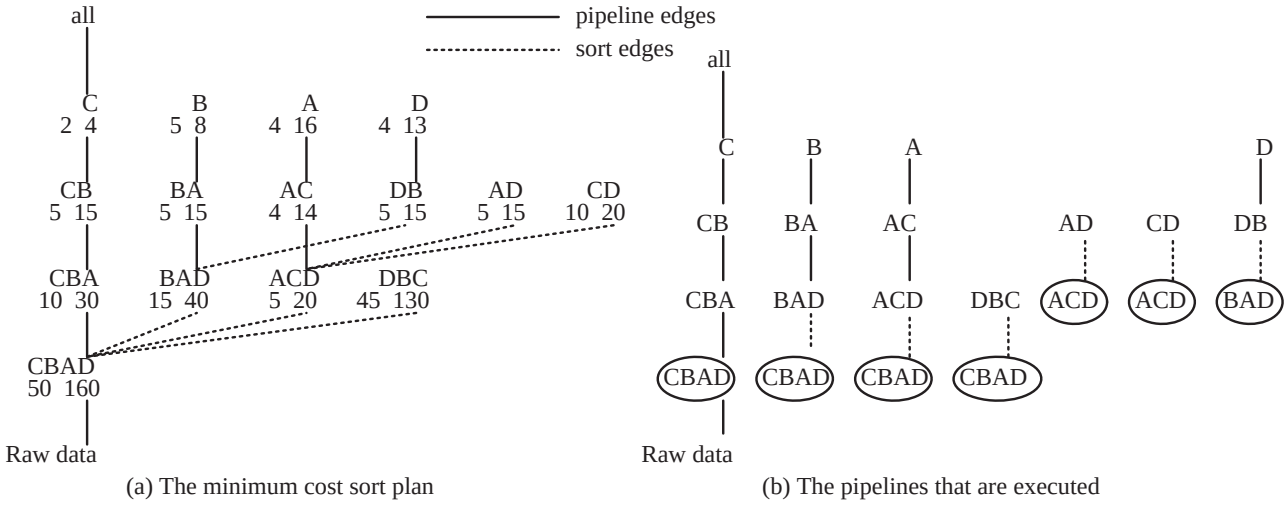


Abbildung 9.6: Anwendung des PipeSort Algorithmus auf das Suchgitter

Beispiel: In Abbildung 9.6 (a) wurde der PipeSort Algorithmus auf das Suchgitter von Abbildung 9.4 angewandt. In Abbildung 9.6 (b) ist die Ausführungsreihenfolge zur Erzeugung der Anfragen dargestellt. In Abbildung 9.6 (a) sind die Gesamtkosten optimal, obwohl die Gesamtzahl der Sortierungen nicht optimal ist.

Ergebnis: PipeSort erzeugt jede Anfrage aus einer Anfrage in der vorherigen Ebene. Obwohl es nicht bewiesen ist, daß das Vorgehen nach Ebenen optimal ist, konnte bisher noch keine Verbesserung durch das Zugreifen auf höhere Ebenen festgestellt werden. Durch Versuche konnte nachgewiesen werden, daß die Ergebnisse des PipeSort Algorithmus an die empirisch erwarteten Ergebnisse bei den meisten Datenbeständen heranreichen.

9.8 Implementierung basierend auf Hashverfahren

Eine weitere Methode einen Datenwürfel zu erzeugen ist die Verwendung von Hashverfahren [39]. Für diese Art der Implementierung werden die Optimierungsmethoden „cache-results“ (2) und „amortize-scans“ (3) verwendet. Zum Beispiel können die Anfragen AB und AC in einem Durchlauf aus ABC erzeugt werden, wenn die Hashtabellen für AB und AC in den Hauptspeicher passen. Nachdem AB erzeugt wurde könnte man A und B berechnen, während AB noch immer im Hauptspeicher ist, womit man die Zugriffe auf den Hintergrundspeicher für AB gespart hätte. Falls der Speicher nicht begrenzt wäre, könnte man alle Optimierungen wie folgt verwenden:

Berechne die größte Anfrage (Ebene N) aus den Rohdaten
Für k = N - 1 bis 0
    Für jede Ebene k + 1 Anfrage g
    Erzeuge in einem Durchlauf alle k Anfragen für welche g der kleinste Vorfahr ist
    Speichere g und gib den Speicher frei, den die Hashtabelle von g belegt hat

Da die zu aggregierenden Daten aber viel zu groß sind, um in den Hauptspeicher zu passen, wird bei der Erzeugung von Hashtabellen die Optimierungsmethode „share-partitions“ (5) verwendet. Hierbei werden die Anfragen nach Attributen partitioniert. Wenn beispielsweise die Partitionierung über das Attribut A erfolgt, können alle Anfragen, die A enthalten, unabhängig über jede Partition erzeugt werden. Die Ergebnisse über verschiedene Partitionen müssen dann wieder zusammengeführt werden. Der hier betrachtete Algorithmus PipeHash beinhaltet die Optimierungsmethoden „smallest-parent“ (1), „cache-results“ (2), „amortize-scans“ (3) und „share-partitions“ (5).

9.8.1 Der PipeHash-Algorithmus

Das Suchgitter: Die Grundlage für diesen Algorithmus bildet das gleiche Suchgitter wie auch schon beim PipeSort Algorithmus (Abbildung 9.4).

Der Algorithmus: Zuerst sucht der Algorithmus für jede Anfrage den Vorfahr mit der kleinsten geschätzten Gesamtgröße. Das Ergebnis dieses Schrittes ist ein minimal spannender Baum (MST). Abbildung 9.7 (a) zeigt den MST für ein Suchgitter mit vier Attributen. Unter jeder Anfrage steht deren Größe. Im nächsten Schritt gilt es die Anfragen zu bestimmen, die gemeinsam erzeugt werden können. Zudem müssen die Attribute gewählt werden, nach denen partitioniert wird. Als nächstes soll der MST in kleine Unterbäume aufgeteilt werden, die alle in einem Durchlauf der Anfrage an der Wurzel erzeugt werden können, damit die Lesezugriffe auf den Hintergrundspeicher minimal werden. Falls Daten mit Attributen partitioniert werden müssen, limitiert dies die Erzeugung des Unterbaums auf Anfragen, die diese Attribute enthalten. Wir wählen aus diesem Grund ein Attribut, das uns die Auswahl des größten Unterbaums ermöglicht.

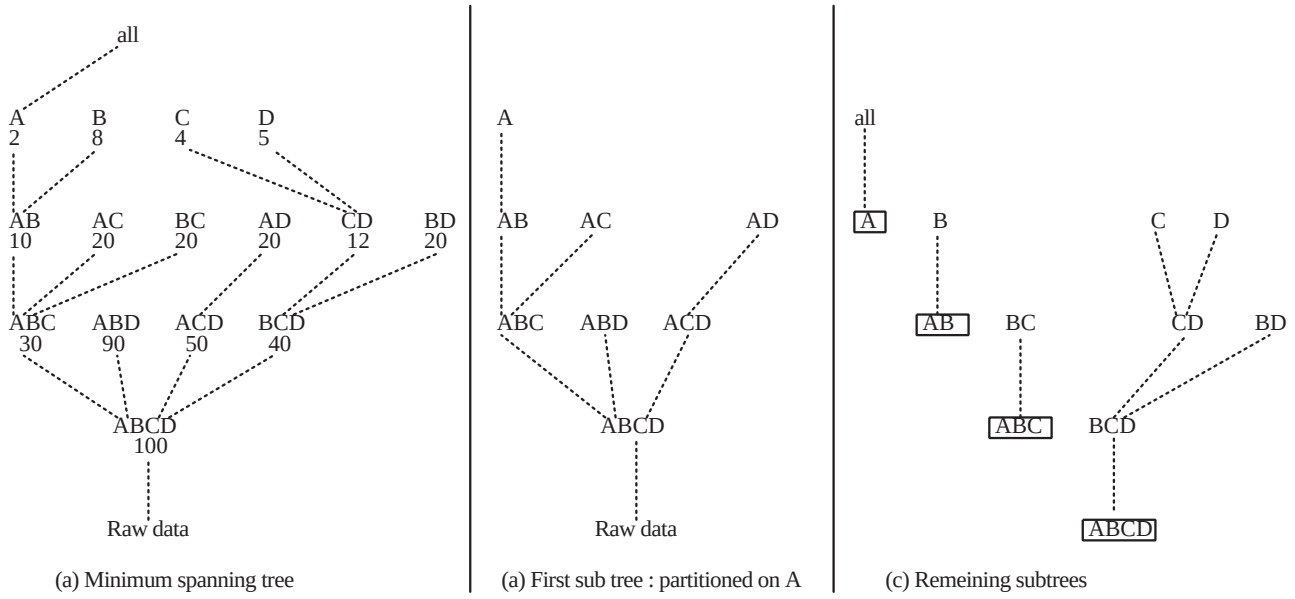


Abbildung 9.7: Anwendung des PipeHash Algorithmus auf das Suchgitter

PipeHash:

(Eingabe: Suchgitter mit geschätzter Größe jeder Anfrage)
Initialisiere Liste mit dem MST aus dem Suchgitter
Solange die Liste nicht leer ist
    Nehme einen Baum T aus der Liste
    T' = Wähle-Unterbaum von T der als nächstes ausgeführt wird
    Berechne-Unterbaum T'

Wähle-Unterbaum:

Wenn für T benötigter Speicher < verfügbarer Speicher,
dann gebe T zurück sonst, lasse S die Menge der Attribute von Wurzel(T) sein
    (Man wähle s (S um die Wurzel von T zu partitionieren.
        Für jede Wahl von s erhält man einen Unterbaum Ts von T
        mit der Wurzel von T und allen Anfragen die s enthalten)
    Sei Ps = maximale Anzahl von möglichen Partitionen von Wurzel(T),
        wenn nach s $\subset$ S partitioniert wird
        Man suche s $\subset$ S so, daß
            der von $\frac{Ts}{Ps}$ benötigte Speicher größer als verfügbarer Speicher,
            und Ts die größte Untermenge von S ist.
        Entferne Ts von T
        Dies hinterläßt T - Ts, was einen Wald mit mehreren kleinen Bäumen
        darstellt, der an die Liste angehängt wird
Gebe Ts zurück

Berechne-Unterbaum:

numParts = (von T' benötigter Speicher) * (fudgefaktor) / verfügbarer Speicher
Pertitioniere die Wurzel von T' in numParts
Für jede Partition von Wurzel(T')
    Für jeden Knoten in T' (mit Breitensuche)
        Erzeuge alle Kinder dieses Knoten in einem Durchlauf
        Falls der Knoten gecashed ist, speichere ihn und gebe den durch seine
        Hashtabelle belegten Speicher frei

Beispiel: Abbildung 9.7 illustriert den PipeHash Algorithmus für das vier Attribute Suchgittter von Abbildung 9.4. Abbildung 9.7 (a) zeigt den minimal aufspannenden Baum. Man nehme an, daß nicht genug Speicher vorhanden ist um den gesamten Baum auf einmal zu berechnen. Abbildung 9.7 (b) zeigt den ersten Unterbaum T-A der entsteht wenn A als Attribut zur Partitionierung gewählt wird. Nachdem T-A entfernt wurde bleiben noch

vier Unterbäume zurück (Abbildung 9.7 (c)). Keine dieser Anfrage enthält A. Um T-A zu berechnen, werden die Rohdaten zuerst nach A partitioniert. Für jede Partition erzeugt man zuerst ABCD; danach wird ABCD durchlaufen (solange es im Speicher ist) um ABC, ABD, ACD zu erzeugen; speichern ABCD und ABD; erzeugen AD von ACD, speichern ACD und AD; durchlaufen ABC um AB und AC zu erzeugen; speichern ABC und AC; durchlaufen AB um A zu erzeugen und speichern AB und A. Nachdem T-A erzeugt wurde bearbeiten wir alle restlichen Unterbäume in der Liste.

Ergebnis: PipeHash ist durch die Optimierung smallest-parent (1) vorbelastet. Für jede Anfrage bestimmen wir zuerst den kleinsten Vorfahr und wenden erst dann die anderen Optimierungsmethoden an. Zum Beispiel hätte man BC durch BCD (Abbildung 9.7 (c)) anstatt durch ABC erzeugen können. Dies hätte den extra Durchlauf durch ABC erspart. In der Praxis zeigte sich, daß die Optimierung nach Hintergrundspeicherzugriffen weniger wichtig als die Reduzierung der CPU-Belastung durch die Auswahl des kleinsten Vorfahr.

9.8.2 Vergleich zwischen PipeSort und PipeHash

Dataset	Number of grouping attributes	Number of tuples (in millions)	size (in MB)
Dataset-A	3	5.5	110
Dataset-B	4	7.5	121
Dataset-C	5	9	180
Dataset-D	5	3	121
Dataset-E	6	0.7	18

Abbildung 9.8: Verwendete Testdaten

Um die Effizienz der Algorithmen zu testen wurden verschiedene Datenbestände ausgesucht. Abbildung 9.8 zeigt die verschiedenen Datenbestände. Die Basis zur Evaluierung bildeten die beiden Algorithmen PipeHash, PipeSort. Zusätzlich wurden die beiden Verfahren NaiveHash und NaiveSort angewandt, die jede Anfrage einzeln entweder nach Sortierung oder Hashtabelle berechnen.

- In den verschiedenen Experimenten erwiesen sich die Algorithmen als zwei bis zu acht mal schneller als naive Methoden (naive Methoden = alle Anfragen werden einzeln ohne Optimierung nach Hash- bzw. Sort-Methode erzeugt).
- Die Leistung des PipeHash Algorithmus kam an die berechnete untere Grenze für Hash-Verfahren. Die maximale Leistungsdifferenz lag bei 8%.
- Die maximale Leistungsdifferenz für den PipeSort Algorithmus und der unteren Grenze für Sortierungsverfahren lag bei 22%.
- Für die meisten Datenbestände lag der PipeHash Algorithmus unter dem PipeSort Algorithmus.

9.9 Zusammenfassung

In diesem Text wurden zuerst die Unterschiede zwischen OLTP und OLAP diskutiert. Im weiteren Verlauf wurde darauf eingegangen, welche Methoden es gibt einen OLAP–Server effektiv zu implementieren. Hierbei wurde ein Gitternetznotation vorgestellt, die es erleichtert die Anforderungen an einen OLAP–Server zu veranschaulichen. Anhand dieses Gitternetzes kann man die Implementierung der Datenstruktur des OLAP–Servers strukturieren. Auch die beiden vorgestellten Algorithmen „PipeSort“ und „HashSort“ verwendeten zur Verdeutlichung des Ablaufes dieses Gitter. Als Grundlage für die beiden Algorithmen sind fünf Optimierungsmethoden vorgestellt worden. Beide Algorithmen haben versucht die Optimierungsmethoden so zu kombinieren, daß die Implementierung besonders effektiv ist. Letztendlich wurden zwei Verfahren vorgestellt, mit denen die effektive Implementierung der Datenstruktur eines OLAP–Servers ermöglicht wird.

Kapitel 10

Data-Mining

Matthias Barth

Kurzfassung *Data-Mining ist eine wichtige Anwendung für Unternehmen, die aus ihren gesammelten Verkaufsvorgängen und Datenbeständen Hinweise für Entscheidungen erhalten wollen. Es ist sehr wichtig, effiziente Wege zu finden, Data-Mining praktisch anzuwenden. Vor allem das Finden von Mustern in großen Datenbanken und das Aufstellen von Regeln, sogenannten association rules erfordert einige Vorüberlegungen, die hier vorgestellt werden. Zunächst werden allgemeine Rahmenbedingungen für das Generieren von Regeln festgelegt: Die Begriffe support und confidence einer Regel werden definiert. Anschließend wird auf die Algorithmen SetM und Discovery eingegangen, die association rules generieren. Neben deren Aufwandsanalyse soll anschliessend auch noch ein Ansatz mit einer gegebenen Taxonomie (is-a Hierarchie) auf den Verkaufsartikeln und seine Auswirkungen auf die Regelgenerierung betrachtet werden. Der zugehörige Algorithmus Basic wird schrittweise optimiert und verfeinert. Die so erhaltenen Algorithmen Cumulate und EstMerge laufen 2-5 Mal schneller als Basic.*

10.1 Einleitung

Das Ziel von Data-Mining, zu deutsch Datenbergbau ist das Finden von Wissen (*eng.* knowledge discovery) in großen Datenbanken. Ein wichtiges Einsatzgebiet des Data-Mining ist das Entdecken interessanter Zusammenhänge, sogenannter „Verbindungsregeln“ (*eng.* association rules) aus Sammlungen großer Datenmengen, worauf hier näher eingegangen werden soll. Data-Mining ist eine sehr wichtige Anwendung für Unternehmen, deren Wettbewerb immer mehr von der Entscheidungsfindung abhängig wird. Deswegen versuchen Firmen häufig, aus alten Geschäftsvorgängen und vergangenen Entscheidungen zu lernen, um anstehende Entscheidungen besser, d.h. sicherer treffen zu können. Um diesen Prozeß zu unterstützen, müssen natürlich erst einmal große Datenmengen gesammelt und gespeichert werden. Dies wird vom Data Warehouse bewerkstelligt, um die vielen gesammelten, verteilten operativen Daten eines Unternehmens in eine geeignete Form zu bringen und zur Verfügung zu stellen. Später werden diese Daten dann analysiert, d.h. dann findet das eigentliche Data-Mining statt. Data-Mining ist sozusagen auf das Data Warehouse „aufgesetzt“. Diese neue Form der Entscheidungsfindung ist sehr wichtig für

unterschiedliche Typen von Geschäftszweigen, z.B. erhalten Supermärkte Profile über ihre Kunden (Alter, Geschlecht etc.) und deren Kaufverhalten (Warenkorbanalyse).

10.2 Definitionen

Def.: Gegeben sei eine Menge I von Artikeln i (*eng.* items). Hat man zusätzlich eine Menge von Transaktionen D (*eng.* transactions) gegeben, in der jede Transaktion $T \in D$ aus einer Menge von Artikeln $i \in I$ besteht, so ist eine **„Verbindungsregel“** (*eng.* association rule) ein Ausdruck $X \Longrightarrow Y$, in dem X und Y Mengen von Artikeln (*eng.* itemsets) sind mit $X \cap Y = \{\}$.

Es kommt nun darauf an, in der Datenbank gewisse Muster (*eng.* patterns) zu finden. Kommen z.B. die items a, b, c oft zusammen in Transaktionen vor, so kommt das Muster $\{a, b, c\}$ oft vor. Das heißt, man könnte unter anderem auf die Regel $\{a, b\} \Longrightarrow \{c\}$ schließen (die Anordnung von a, b ist willkürlich). $\{a, b\}$ nennt man den Vorgänger, c nennt man Folgerung der Regel. Bevor man nun aber einfach beliebige Regeln aufstellt, sind gewisse Voraussetzungen bzw. Zwänge (*eng.* constraints) zu beachten. Intuitiv bedeutet dies, daß alle Transaktionen in der Datenbank, welche items aus X enthalten, mit einer gewissen Häufigkeit auch items aus Y enthalten müssen, um obige Regel aufzustellen.

Def.: Gegeben sei ein itemset X. Der **support** $s(X)$ von X ist definiert als die Anzahl der Transaktionen $T \in D$, welche alle items aus X enthalten. Der support einer Regel $X \Longrightarrow Y$ ist definiert als die Anzahl aller Transaktionen $T \in D$, welche alle items aus $X \cup Y$ enthalten.
Oft ist mit dem support eines itemsets auch die relative Häufigkeit in bezug auf alle Transaktionen gemeint.

Def.: Gegeben sei eine untere Schranke σ für den support. Ein itemset X wird **groß** (*eng.* large) bzw. **häufig** (*eng.* frequent) genannt, falls gilt $s(X) \geq \sigma$.
σ wird oft als min-support bezeichnet und kann vom Benutzer festgelegt werden.

Def.: Die **Konfidenz** c (*eng.* confidence) einer Regel $X \Longrightarrow Y$ ist durch

$$c = \frac{s(X, Y)}{s(X)} \tag{10.1}$$

festgelegt. [1] c ist die Wahrscheinlichkeit, daß eine Transaktion T mit items aus X auch items aus Y enthält.
Während confidence ein Maß für die „Stärke“ einer Regel darstellt, ist der support gleichbedeutend mit statistischer Signifikanz. Eine Motivation für die Einführung einer support-Schranke ist der Wunsch, interessante Regeln zu erhalten. Denn wenn der support nicht groß genug ist, bedeutet dies, daß die Regel nicht beachtenswert ist oder zumindest einen geringeren Stellenwert in der Analyse einnimmt.

Def.: Ein itemset mit k items heißt k-itemset.

[1] In der Literatur wird oft auch $c = \frac{s(Y)}{s(X,Y)}$ als confidence definiert.

10.3 Problemabgrenzung

Wie oben schon angedeutet, tritt bei einer gegebenen Menge T von Transaktionen das Problem auf, alle association rules zu Tage zu fördern, deren support und confidence grösser ist als die vom Benutzer festgelegte Schranke. Um nicht zu viele redundante Regeln zu finden, kann man vom Benutzer auch eine Grenze für das Interesse an einer Regel eingeben lassen.
Das Problem kann also folgendermaßen in 3 Teile zerlegt werden:

1. Finde alle itemsets, deren support größer als σ ist.

2. Verwende diese large itemsets, um association rules zu generieren. Die zugrundeliegende Idee ist, daß man z.B. bei large itemsets $\{a, b, d, e\}$ und $\{a, b\}$ die Regel $\{a, b\} \Longrightarrow \{d, e\}$ dadurch überprüft, indem man die confidence $c = \frac{s(\{a,b,d,e\})}{s(\{a,b\})}$ berechnet. Ist c größer als ein bestimmter Wert, dann hält die Regel der Prüfung stand und wird folglich ausgegeben.

3. Aus der Menge der Regeln werden alle Uninteressanten herausgefiltert.

Im Folgenden soll auf Algorithmen eingegangen werden, die den ersten Schritt realisieren. Hat man die large itemsets gegeben, so kann man mit den Algorithmen aus [44] die Regeln generieren.

10.4 Algorithmen

10.4.1 Basisalgorithmus

Die meisten Algorithmen zur Bestimmung der large itemsets arbeiten wie folgt:
In einem ersten Schritt werden die supports für einzelne items in der Datenbank abgezählt und alle large 1-itemsets gefunden. Dann werden iterativ für $k = 2, 3, 4, \ldots$ sogenannte Kandidaten (*eng.* candidate) k-itemsets aus den large ($k \Leftrightarrow 1$)-itemsets des vorherigen Durchlaufs gebildet. Der wesentliche Grundgedanke hierbei ist, daß alle ($k \Leftrightarrow 1$)-Teilmengen eines large k-itemsets ebenfalls large sind, umgekehrt sind also alle die k-itemsets Kandidaten itemsets, deren sämtliche ($k \Leftrightarrow 1$)-Teilmengen large sind.
Für diese Kandidaten werden anschließend bei einem Durchlauf durch die Datenbank die supports bestimmt. Diejenigen Kandidaten, welche als large erkannt werden, bilden die Grundlage für die Generierung der Kandidaten der Grösse $k + 1$ im nächsten Schritt.

Im folgenden wird nun auf den Algorithmus SetM eingegangen. Anschließend wird der Algorithmus Discovery betrachtet, der auch schon Grundzüge einer Hierarchie behandeln kann, bevor abschließend der Algorithmus Basic vorgestellt wird, welcher Hierarchien von items verwendet. Basic wird optimiert zu Cumulate und später zu EstMerge.

10.4.2 Algorithmus SetM

Dieser Algorithmus aus [41] ist mengenorientiert. Verwendet wird hier eine relationale Datenbank SALESDATA ($transid$, $item$). In ihr werden die Transaktionen als Tupel gespeichert. Ferner werden temporäre Relationen $R_k^{'}$ und R_k der Form ($transid$, $item_1, \ldots,$ $item_k$) benutzt. $R_k^{'}$ enthält alle k-itemsets, die eine large ($k \Leftrightarrow 1$)-Teilmenge enthalten. R_k enthält nur die large k-itemsets. Dabei werden die patterns der Einfachheit halber alphabetisch sortiert.
Der erste Schritt ist in SQL wie folgt implementiert:

INSERT INTO $R_k^{'}$
SELECT $p.transid$, $p.item_1, \ldots,$ $p.item_{k-1}$, $q.item$
FROM R_{k-1} p, SALESDATA q
WHERE $q.transid = p.transid$
AND $q.item > p.item_{k-1}$

Nun muß man die supports aller patterns aus $R_k^{'}$ berechnen und die large itemsets in einer weiteren Relation C_k abspeichern (dabei aber mehrfach autretende Mengen einfach aufführen):

INSERT INTO C_k
SELECT $p.item_1, \ldots,$ $p.item_k$, COUNT(*)
FROM $R_k^{'}$ p
GROUP BY $p.item_1, \ldots,$ $p.item_k$
HAVING COUNT (*) >= :minsupport

Bevor man nun die itemsets der Grösse $k+1$ generiert, muß man erst die Tupel aus $R_k^{'}$ auswählen, die einen support größer als die untere Schranke haben. Außerdem soll die resultierende Relation sortiert werden nach ($transid$, $item_1, \ldots,$ $item_k$), um im nächsten Iterationsschritt zum Aufstellen von $R_{k-1}^{'}$ wieder die Vorsortiertheit verwenden zu können:

INSERT INTO R_k
SELECT $p.transid$, $p.item_1, \ldots,$ $p.item_k$
FROM $R_k^{'}$ p, C_k q
WHERE $p.item_1 = q.item_1$ AND
$\vdots$
$p.item_k = q.item_k$
ORDER BY $p.transid$, $p.item_1, \ldots,$ $p.item_k$

Dieser Prozeß wird nun solange wiederholt bis $R_k = \{\}$. An einem Beispiel soll dieses Vorgehen deutlich werden. Die Beispieldatenbank besteht aus 10 Geschäftsvorgängen, die je 3 items enthalten (siehe Tabelle 10.1). Der minimum-support sei 30 %, d.h. 3 Transaktionen. Die confidence sei 70 %.

Der nächste Schritt wäre es nun, mithilfe der Relationen die Regeln zu generieren. Für jedes Muster der Länge k betrachtet man alle möglichen Kombinationen von ($k \Leftrightarrow 1$)

trid	item	item	item
10	A	B	C
20	A	B	D
30	A	B	C
40	B	C	D
50	A	C	G
60	A	D	G
70	A	E	H
80	D	E	F
90	D	E	F
99	D	E	F

trid	item
10	A
10	B
10	C
20	A
20	B
20	D
30	A
30	B
30	C
...	...

item	support
A	6
B	4
C	4
D	6
E	4
F	3

Tabelle 10.1: Transaktionen, zugehörige Relation und Relation C_1

trid	item_1	item_2
10	A	B
10	A	C
10	B	C
20	A	B
20	A	D
20	B	D
30	A	B
...	...	...

item_1	item_2	support
A	B	3
A	C	3
B	C	3
D	E	3
D	F	3
E	F	3

trid	item_1	item_2
10	A	B
10	A	C
10	B	C
20	A	B
30	A	B
30	A	C
30	B	C
...	...	...

Tabelle 10.2: Relationen R_2', C_2 und R_2

trid	item_1	item_2	item_3
10	A	B	C
30	A	B	C
20	A	B	D
40	B	C	D
80	D	E	F
90	D	E	F
99	D	E	F

item_1	item_2	item_3	supp.
D	E	F	3

trid	item_1	item_2	item_3
80	D	E	F
90	D	E	F
99	D	E	F

Tabelle 10.3: Relationen R_3', C_3 und R_3

Minimum support (%)	Ausführungszeit in Sekunden
0,1	6,90
0,5	5,30
1	4,64
2	4,22
5	3,97

Tabelle 10.4: Ausführungszeit von SetM bei variablem min-support

items auf der linken Seite einer Regel. Das verbleibende item steht auf der rechten Seite. Nun untersucht man für jede dieser Kombinationen aus linker und rechter Seite, ob die confidence die minimum-confidence erreicht (dazu benutzt man den support aus C_k und C_{k-1}). Ist dies der Fall, so wird die Regel ausgegeben. Im Beispiel von oben war der min-support 30% (3 Transaktionen) und die min-confidence 70%. Betrachtet werden soll z.B. das Muster $\{A, B\}$. Der support liegt bei 3, und $c = \frac{s(\{A,B\})}{s(\{A\})} = \frac{3}{6} = 50\% < 70\%$. Deswegen wird die Regel $A \Longrightarrow B$ nicht berücksichtigt. Jedoch ist $\frac{s(\{A,B\})}{s(\{B\})} = \frac{3}{4} = 75\% > 70\%$. Die Regel $B \Longrightarrow A$ wird also ausgegeben. Man sieht, daß die Reihenfolge der items in einer Regel von Bedeutung ist. In den nächsten Schritten werden nun iterativ alle Kombinationen von $k \Leftrightarrow 2, \ldots, 1$ item(s) auf der linken Seite und entsprechend $2, \ldots, k \Leftrightarrow 1$ items auf der rechten Seite getestet. Für eine detailliertere Betrachtung wird auf [44] verwiesen.

Aufwandsanalyse: Der Algorithmus SetM wurde nun mit Hilfe der Datenbank eines großen Einzelhandelsgeschäftes mit insgesamt 47.000 Verkaufsvorgängen getestet. Dazu gibt es einige interessante Ergebnisse.

1. Als erstes wurde untersucht wie die Größe der Relationen R_i sich mit jedem Durchlauf bei verschieden gewählten min-supports verändert. Das Ergebnis ist in Abb. 10.1 zu sehen. Die maximale Größe der erhaltenen Regeln ist 3, da in allen Fällen $|R_4| = 0$ ist (mit $|R_4|$ ist die Kardinalität von R_4 gemeint). Die Startrelation ist immer dieselbe mit $|R_1| \approx$ 116.000. Man sieht, daß sich für hinreichend kleinen min-support ($\leq 0,1\%$) die Größe der Relation R_i erst erhöht und danach verringert. Für alle anderen Werte verringert sich der Wert von R_i sofort. Für große Werte des min-supports ist diese Abnahme zuerst sehr hoch, danach fällt sie etwas zurück. Man sieht also, daß für kleinen min-support die Anzahl der erhaltenen Regeln zunimmt. Außerdem nimmt die Zahl der items auf der linken Seite zu.

2. Der nächste Untersuchungspunkt ist die Ausführungszeit in Abhängigkeit des min-supports. Das Ergebnis ist in Tabelle 10.4 sichtbar. Man sieht deutlich, daß der Algorithmus stabil in seinem Zeitaufwand ist.

3. Der letzte Punkt der Analyse bezieht sich auf das lineare Vervielfachen der Datenbank, d.h. man fügt die Datenbank n-mal hintereinander und untersucht nun den Zeitaufwand des Algorithmus. Man sieht in Abbildung 10.2, daß sich SetM linear verhält.

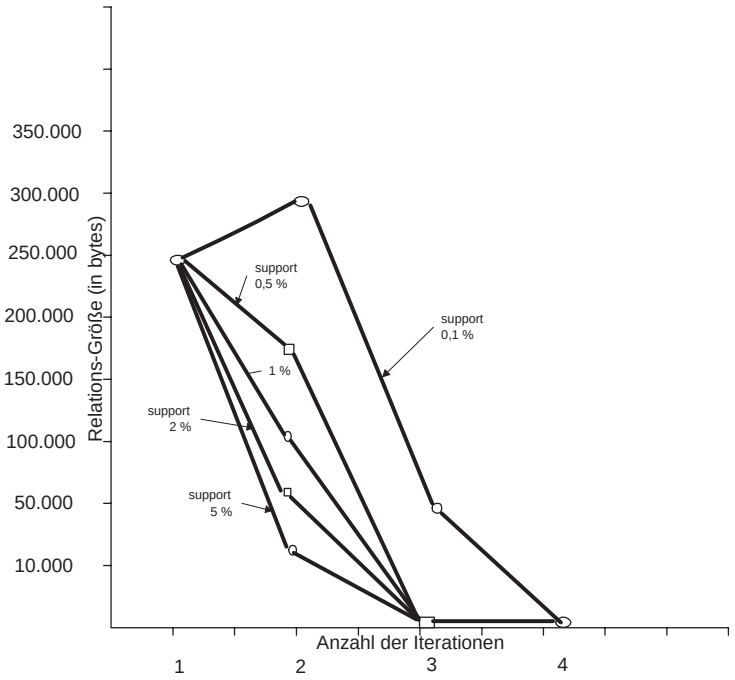


Abbildung 10.1: Größe der Relationen

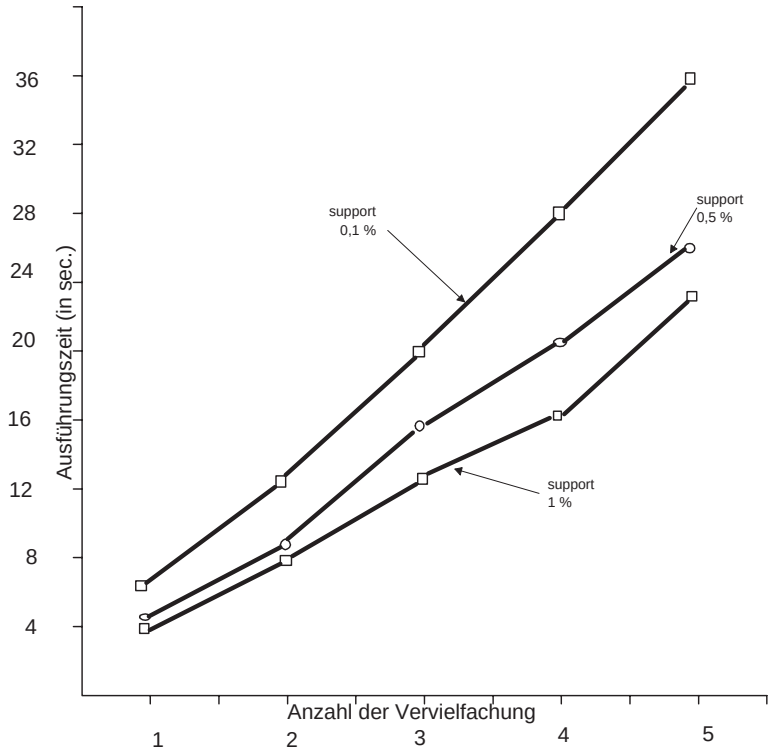


Abbildung 10.2: Lineares Vergrößern der Datenbank

10.4.3 Algorithmus Discovery

Dieser Algorithmus (siehe [42]) ist von der Vorgehensweise ähnlich zum Vorigen. Allerdings ist die Datenbank, auf die er zugreift, anders aufgebaut. Bei SetM werden (wie z.B. oben) die Daten in Reihen als eine Menge von Transaktionen gespeichert. In der nun folgenden Struktur, die man „zerlegte Speicherstruktur“ (*eng.* decomposed storage structure) nennt, hat jede Transaktion eine einheitliche Identifikationsnummer (TID). Die Datenbank ist als eine Menge von items in Spalten angelegt. Jede Spalte entspricht dabei einer eigenen Relation. Für jedes item sind dann die TIDs, die dieses item enthalten aufgezählt. Der Vorteil liegt darin, daß jeder Kandidat im Suchraum sein Gegenstück in der Datenbank hat. So ist der support für ein 1-itemset X einfach die Größe der Spalte X in der Datenbank. Man kann so mit wenigen Datenbankoperationen statt eines kompletten Durchlaufs durch die Datenbank die large itemsets finden. Die 1-itemsets sind einfach die Spalten, deren Größe die untere Schranke des supports erreicht.
Die 2-itemsets kann man bestimmen, indem man den Datenbankbefehl „Durchschnitt“ (*eng.* intersection) verwendet, z.B. $a \cap b = intersect(a, b)$. Das Ergebnis ist eine neue Spalte (und Relation) $\{a, b\}$, die die TIDs enthält, welche sowohl in a als auch in b vorkommen.
Um nicht zu viele dieser Datenbankoperationen durchführen zu müssen, kann man Informationen über Mengen von Kandidaten statt einzelner Kandidaten sammeln. Die Idee ist einfach : Seien A_1, A_2, A_3, A_4 large itemsets.
Statt den support für $\{A_1, A_2\}, \ldots, \{A_1, A_4\}, \{A_2, A_3\}, \ldots, \{A_3, A_4\}$ zu berechnen, bestimmt man zunächst $A_1 \cup A_2$ und $A_3 \cup A_4$ mit dem Vereinigungsbefehl (*eng.* union). $A_1 \cup A_2$ enthält alle TIDs, die entweder in A_1 oder in A_2 enthalten sind. Ist nun $(A_1 \cup A_2) \cap (A_3 \cup A_4)$ nicht large, so folgt daraus, daß $\{A_1, A_3\}, \{A_1, A_4\}, \{A_2, A_3\}, \{A_2, A_4\}$ nicht large sind. Ist der Durchschnitt aber large, so muß für jede der Mengen $\{A_1, A_2\}, \ldots, \{A_1, A_4\}, \{A_2, A_3\}, \ldots, \{A_3, A_4\}$ einzeln bestimmt werden, ob sie large ist oder nicht. Ist keine davon large, so war die Arbeit umsonst, was man auch als *false alarm* bezeichnet.
Durch die Berechnung von $A_1 \cup A_2$ kann man auch bestimmen, ob $\{A_1, A_2\}$ large ist: $s(\{A_1, A_2\}) = s(A_1) + s(A_2) \Leftrightarrow s(A_1 \cup A_2)$. Die Berechnung von $(A_1 \cup A_2) \cap (A_3 \cup A_4)$ kostet 3 DB-Operationen. Ist das Resultat nicht large, so muß keine weitere DB-Operation mehr durchgeführt werden. Ist das Resultat large, so muß man 4 weitere intersections durchführen. Man gewinnt also 3 oder verliert 1 Operation im Vergleich zur naiven Methode, bei der man alle 6 intersections durchführen muß.
Dieses Verfahren kann erweitert werden auf beliebig viele itemsets, die auf einmal berechnet werden. Man konstruiert im Prinzip einen Baum (siehe Abb. 10.3)
Discovery benutzt diese Konstruktion allerdings aus Effizienzgründen: Der Baum wird bis zu einer gewissen Höhe berechnet, ab der false alarms überhand nehmen, dann wird die Berechnung abgebrochen. Der support aller verbleibenden itemsets wird einzeln berechnet.

Konstruiert man bei diesem Algorithmus den Baum so, daß er eine Hierarchie darstellt, kann man einige der inneren Knoten auch mit Oberbegriffen bezeichnen, deren support wir dann ebenfalls kennen. Hierarchien sind allerdings nicht notwendigerweise Binärbäume, so daß man vielleicht Zwischenknoten einfügen muß (siehe K_1 in Abb. 10.4)

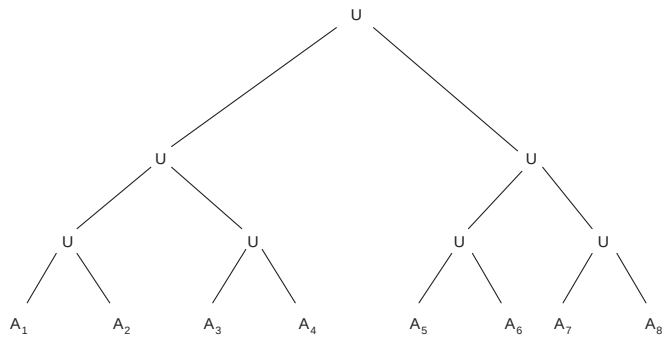


Abbildung 10.3: Schema des konstruierten Baumes

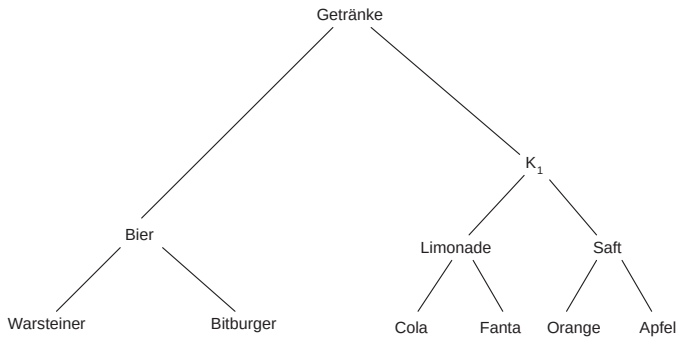


Abbildung 10.4: Hierarchie-Baum

10.4.4 Einschub: Taxonomien

Mit dem nächsten Algorithmus beginnt die Verwendung von Taxonomien (*eng.* taxonomies) über den items, denn diese kamen zwar im vorigen Algorithmus vor, standen jedoch nicht im Mittelpunkt. Diese Taxonomien sind sehr oft vorhanden. Ein Beispiel zeigt Abbildung 10.5. Die Taxonomie besagt, daß Schuhe Fußbekleidung sind, Pullover Kleidung usw. Frühere Ansätze konnten keine Taxonomien berücksichtigen und erlaubten nur items in der untersten Ebene. Doch Regeln zu finden, die verschiedene Ebenen des Baumes berühren sind von Vorteil, denn

- Es kann Regeln in unteren Ebenen geben, die keinen min-support haben, z.B. kann es wenige Leute geben, die Jacken zusammen mit Bergsteigerstiefeln kaufen, aber viele, die Jacken und Fußbekleidung kaufen. Eventuell können deswegen viele interessante Zusammenhänge nicht entdeckt werden.
- Taxonomien können dazu verwendet werden, uninteressante bzw. redundante Regeln zu streichen bzw. zu entfernen.

Aus diesem Grunde soll an dieser Stelle die Definition von association rules verfeinert werden.

Def.: Man nennt ein item $\hat{i}$ **Vorfahr** eines items i, falls $\hat{i}$ im Taxonomiebaum vor i steht. Man nennt ein itemset $\hat{X}$ einen Vorfahren eines itemsets X, falls man $\hat{X}$ aus X generieren

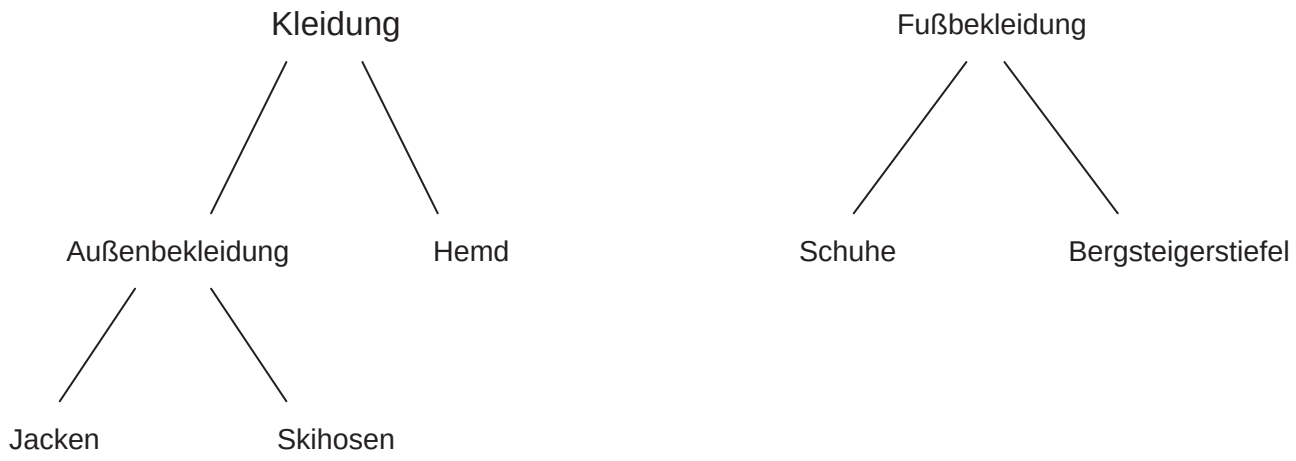


Abbildung 10.5: Taxonomie über Bekleidung

kann, indem man ein oder mehrere items aus X durch ihre Vorfahren ersetzt und $\hat{X}$ und X dieselbe Kardinalität besitzen. Analog nennt man ein item i **Nachfahr** eines items $\hat{i}$, falls i nach $\hat{i}$ im Taxonomiebaum steht.

Def.: Eine Transaktion T *unterstützt* (*eng.* to support) ein item i, falls $i \in T$ oder i ist ein Vorfahr (*eng.* ancestor) eines items in T. T unterstützt ein itemset X, wenn T jedes item $i \in X$ unterstützt.

Def.: Gegeben seien itemsets X und Y. Eine **„allgemeine Verbindungsregel“** (*eng.* general association rule) ist ein Ausdruck der Form $X \Longrightarrow Y$ mit $X \cap Y = \{\}$ **und** kein item aus Y ist Vorfahr eines items aus X. Dabei dürfen die items aus X und Y aus allen Ebenen der Taxonomie stammen.

Der Grund für den Ausschluß des Vorfahren liegt darin, daß eine Regel der Form $i \Longrightarrow ancestor(i)$ trivialerweise immer wahr (redundant) ist mit confidence c = 100 %.

10.4.5 Algorithmus Basic

Zunächst wird das Problem betrachtet, ob eine Transaktion T ein itemset X unterstützt. Verwendet man die einfache Transaktion, bedeutet das nach der obigen Definition, für jedes item i überprüfen zu müssen, ob es selbst oder einer seiner Nachfahren in der Transaktion enthalten ist. Diese Aufgabe wird deutlich einfacher, wenn man zuerst alle Vorfahren eines jeden items $i \in T$ zu T hinzufügt und eine erweiterte Transaktion T' erhält. Denn dann unterstützt T X genau dann, wenn T' eine Obermenge von X ist. Nachfolgend ist der Algorithmus Basic beschrieben, der nach dieser Methode arbeitet, die Notation ist in Tabelle 10.5 aufgeführt.

L_1 := {large 1-itemsets}
$k := 2$; // k gibt die Zahl der Schleifendurchläufe an
while ($L_{k-1} \neq \{\}$) **do**
begin
  C_k := Neue Kandidaten der Größe k, die aus L_{k-1} generiert werden.

L_k	Menge der large k-itemsets
C_k	Menge der k-itemsets, die Kandidaten sind (potentielle large k-itemsets)
H	Taxonomie

Tabelle 10.5: Notation zum Algorithmus Basic

forall transactions $T \in D$ **do**
begin
Füge alle Vorfahren jedes items aus T zu T dazu und
lösche alle doppelt vorkommenden items in T.
Erhöhe den support-Zähler aller Kandidaten in C_k,
die in T vorkommen.
end
L_k := alle Kandidaten in C_k mit minimum-support.
$k := k + 1$;
end
Ausgabe := $\bigcup_k L_k$;

Der erste Schritt des Algorithmus zählt lediglich die Häufigkeiten der 1-itemsets, um die large 1-itemsets zu finden. Ein späterer Durchlauf k besteht aus 2 Phasen. Zuerst werden nach obiger Idee die large $(k \Leftrightarrow 1)$-itemsets L_{k-1} verwendet, um die Kandidaten itemsets C_k zu generieren. Dazu dient eine spezielle Funktion. Die Idee ist folgende: Hat ein itemset X min-support, so auch alle $(k \Leftrightarrow 1)$-Teilmengen von X und umgekehrt ist jedes k-itemset large, dessen sämtliche $(k \Leftrightarrow 1)$-itemsets large sind (siehe SetM). Der Einfachheit halber nimmt man an, daß alle items jedes itemsets alphabetisch sortiert sind. Als erstes vereinigt man nun L_{k-1} mit sich selbst und erhält so neue large k-itemsets :

insert into C_k
select $p.item_1,\ p.item_2, \ldots,\ p.item_{k-1},\ q.item_{k-1}$
from $L_{k-1}\ p,\ L_{k-1}\ q$
where $p.item_1 = q.item_1, \ldots,\ p.item_{k-2} = q.item_{k-2},\ p.item_{k-1} < q.item_{k-1}$;

Als nächstes werden alle itemsets $X \in C_k$ gelöscht, die $(k \Leftrightarrow 1)$-Teilmengen enthalten, welche nicht Element von L_{k-1} sind. Das reduziert effektiv die Anzahl der weiter zu betrachtenden itemsets.

Beispiel: Gegeben sei folgende Datenbank D über der Taxonomie in Abb. 10.5.

Transaktion	Items
1	Hemd
2	Bergsteigerstiefel, Jacke
3	Bergsteigerstiefel, Skihose
4	Schuhe
5	Schuhe
6	Jacke

Im ersten Schritt werden alle large 1-itemsets bestimmt (hier sei der min-support 33% (d.h. 2 Transaktionen) und die min-confidence 60%). Dazu wird ausnahmsweise einmal der Taxonomie-Baum durchlaufen.

Itemset	Support
{Jacke}	2
{Außenbekleidung}	3
{Kleidung}	4
{Schuhe}	2
{Bergsteigerstiefel}	2
{Fußbekleidung}	4

Die Menge aller large 1-itemsets entspricht L_1. Im Vereinigungsschritt wird {Jacke} mit allen 5 restlichen itemsets vereinigt, dann {Außenbekleidung} mit den restlichen 4 usw. Es werden also $5 + 4 + 3 + 2 + 1 = 15$ 2-itemsets gebildet.
Im nächsten Schritt des Algorithmus wird jede Transaktion mit den Vorfahren der items aus der Taxonomie ergänzt und man erhält folgende Datenbank D':

Transaktion	Items
1	Hemd, Kleidung
2	Außenbekleidung, Bergsteigerstiefel, Fußbekleidung Jacke, Kleidung
3	Außenbekleidung, Bergsteigerstiefel, Fußbekleidung Kleidung, Skihosen
4	Fußbekleidung, Schuhe
5	Fußbekleidung, Schuhe
6	Außenbekleidung, Jacke, Kleidung

Anhand dieser Datenbank wird der support der Kandidaten itemsets bestimmt. Von obigen 15 Kandidaten 2-itemsets werden nun 6 wieder gestrichen, weil sie nicht min-support haben. Die large 2-itemsets sind dann:

Itemset	Support
{Außenbekleidung, Jacke}	2
{Jacke, Kleidung}	2
{Außenbekleidung, Kleidung}	3
{Außenbekleidung, Bergsteigerstiefel}	2
{Außenbekleidung, Fußbekleidung}	2
{Bergsteigerstiefel, Kleidung}	2
{Fußbekleidung, Kleidung}	2
{Fußbekleidung, Schuhe}	2
{Bergsteigerstiefel, Fußbekleidung}	2

Nimmt man nun noch die Regelgenerierung wie beim Algorithmus SetM beschrieben vor, so erhält man z.B. die Regel $Jacke \Longrightarrow Außenbekleidung$, die allerdings auf

der rechten Seite aus dem Vorfahren des items auf der linken Seite besteht. Also hat sie eine confidence von 100 % und ist redundant (s.o.). Andererseits hat die Regel *Außenbekleidung* $\Longrightarrow$ *Jacke* zwar support 2 und confidence 66,6 %, sie ist aber nicht besonders interessant, da man aus dem Verkauf von Außenbekleidung nur folgern kann, daß mit Verläßlichkeit von 66,6 % eine Jacke gekauft wurde. Dies sieht man jedoch schon am support von Jacke (2), Hemd (1) und Außenbekleidung (2+1 =3). Der nächste Algorithmus wird dies berücksichtigen. Läßt man alle itemsets weg, die aus einem item und dessen Nachfolger bestehen und generiert nur aus den restlichen itemsets die Regeln, so erhält man:

Regel	Support (%)	Conf.
Außenbekleidung $\Longrightarrow$ Bergsteigerstiefel	33 %	66,6 %
Außenbekleidung $\Longrightarrow$ Fußbekleidung	33 %	66,6 %
Bergsteigerstiefel $\Longrightarrow$ Außenbekleidung	33 %	100 %
Bergsteigerstiefel $\Longrightarrow$ Kleidung	33 %	100 %

Anm.: Man beachte, daß die Regeln Skihosen $\Longrightarrow$ Bergsteigerstiefel und Jacken $\Longrightarrow$ Bergsteigerstiefel nicht den minimum-support haben, aber die Regel Außenbekleidung $\Longrightarrow$ Bergsteigerstiefel diesen besitzt. Hier sieht man also, daß man mit Hilfe von Taxonomien interessante Regeln erhält, die ohne Oberbegriffe verloren gingen (wie schon bei Algorithmus Discovery angedeutet).

10.4.6 Algorithmus Cumulate

Cumulate (siehe [43]) entsteht aus aus dem Algorithmus Basic durch Hinzufügen einiger Optimierungen. Wie der Name schon suggeriert, werden alle itemsets einer bestimmten Größe in einem Durchlauf gezählt. Die Optimierungen sind im einzelnen:

1. Filtern der zu einer Transaktion hinzugefügten Vorfahren.
   Es ist nicht unbedingt erforderlich, so wie im Algorithmus Basic alle Vorfahren eines items in einer Transaktion T zu T hinzuzufügen. Stattdessen genügt es, wenn man diejenigen Vorfahren der items zu T hinzufügt, welche in einer oder mehreren der Kandidaten-itemsets vorkommen. Das heißt also, wenn sich das Original-item nicht in einer der itemsets befindet, kann es aus der Transaktion gelöscht werden.
   Beispiel hierzu: Gegeben sei die Taxonomie in Abb. 10.5. Sei außerdem {Kleidung, Schuhe} das einzige Kandidaten-itemset, für welches gerade der support bestimmt wird. Dann kann in jeder Transaktion, in der "Jacke" vorkommt, "Jacke" durch "Kleidung" ersetzt werden. Man braucht "Jacke" nicht in der Transaktion zu behalten, ebenso braucht man "Außenbekleidung" nicht zur Transaktion hinzuzufügen.

2. Vorberechnung der Vorfahren.
   Es ist effizienter, die Vorfahren eines jeden items einmal vorzuberechnen als jedesmal durch den Taxonomie-Baum zu laufen.

3. Herausfiltern der itemsets welche ein item und dessen Vorfahren enthalten.
   Um diese Optimierung zu begründen sind 2 Lemmas anzuführen (ihr Beweis ist in [45] angegeben):
   Lemma 1 *Der support für ein itemset X, daß sowohl ein item a als auch seinen Vorfahr $\hat{a}$ enthält, ist identisch zum support des itemsets $X \setminus \{\hat{a}\}$.*
   Lemma 2 *Wenn L_k, die Menge der large k-itemsets kein itemset beinhaltet, das sowohl ein item a als auch seinen Vorfahr $\hat{a}$ enthält, so beinhaltet auch C_{k+1} (die Menge der potentiellen large $(k+1)$-itemsets) kein solches itemset.*
   Lemma 1 zeigt, daß man kein itemset zu berücksichtigen braucht, daß sowohl ein item als auch seinen Vorfahr enthält. Diese Optimierung drückt sich im Algorithmus durch das Löschen der Kandidaten-itemsets der Größe 2 aus, welche die Voraussetzung erfüllen. Lemma 2 versichert, daß ein Löschen dieser Kandidaten 2-itemsets genügt, um sicherzustellen, daß in einem späteren Durchlauf nie Kandidaten itemsets generiert werden, die sowohl ein item a als auch dessen Vorfahr $\hat{a}$ enthalten.

Berechne H^*, die Menge aller Vorfahren jedes items aus H // Optimierung 2
L_1 := {large 1-itemsets}
$k := 2$; // k gibt die Zahl der Schleifendurchläufe an
while ($L_{k-1} \neq \{\}$) **do**
begin
  C_k := Neue Kandidaten der Größe k, die aus L_{k-1} generiert werden.
  if $(k = 2)$ **then**
    Lösche jeden Kandidaten aus C_2 der aus einem item und
      dessen Vorfahr besteht. // Optimierung 3
  Lösche jeden Vorfahr in H^* der nicht in einer der Kandidaten
    aus C_k vorkommt. // Optimierung 1
  forall transactions $T \in D$ **do**
  begin
    foreach item $i \in T$ **do**
      Füge alle Vorfahren von i in H^* zu T hinzu.
    Lösche alle mehrfach vorkommenden items in T.
    Erhöhe den Zähler aller Kandidaten in C_k, die in T vorkommen.
  end
  L_k := alle Kandidaten in C_k mit minimum-support.
  $k := k + 1$;
end
Ausgabe := $\bigcup_k L_k$;

10.4.7 Algorithmus EstMerge

Die Strategie dieses Algorithmus (siehe [43]) ist es, eine Stichprobe der Datenbank zu verwenden und damit den support der Kandidatenmengen in C_k abzuschätzen. In $C_k^{'}$ faßt man dann alle Kandidaten, von denen man erwartet, daß sie min-support haben, zusammen. Zu $C_k^{'}$ werden außerdem alle diejenigen Kandidaten hinzugefügt, von denen man zwar nicht min-support erwartet, deren sämtliche Vorfahren jedoch min-support haben.

Von den letztgenannten Kandidaten glaubt man nicht, daß sie min-support haben. Deswegen braucht man für keinen ihrer Nachfahren den support zu bestimmen (denn hat ein Vorfahr $\hat{X}$ einer Menge X nicht min-support, so hat auch X nicht min-support).
Haben jedoch einige dieser Kandidatenmengen wider Erwarten doch min-support, so bezeichnen wir die Menge aller Nachfahren dieser Kandidaten mit C_k''. Statt nun einen Extra-Durchlauf nur für C_k'' zu machen, kann man diese Kandidaten zusammen mit den Kandidaten in C_{k+1} abzählen (den support bestimmen). Dies geschieht, indem man alle Kandidaten mit min-support aus C_k' und alle Kandidaten aus C_k'' zu L_k zusammenfaßt. Obwohl man nicht weiß, ob die Kandidaten in C_k'' min-support haben oder nicht, nimmt man dies bei der Generierung von C_{k+1} einfach an. Die Korrektheit wird dadurch nicht beeinflußt. Nachfolgend steht der Algorithmus, wobei die Optimierungen, die in Cumulate vorgenommen wurden, immer noch gültig sind, aber wegen der Übersichtlichkeit nicht aufgeführt sind.

L_1 := {large 1-itemsets}
Generiere D_S, eine Stichprobe der Datenbank.
$k := 2$; // k gibt die Zahl der Schleifendurchläufe an
$C_1'' := \{\}$; // C_k'' beinhaltet Kandidaten der Größe k, deren support
// mit Kandidaten der Größe $k+1$ gezählt wird
while ($L_{k-1} \neq \{\}$ **or** $C_{k-1}'' \neq \{\}$) **do**
begin
  C_k := Neue Kandidaten der Größe k, die aus $L_{k-1} \cup C_{k-1}''$ generiert werden.
  Schätze den support der Kandidaten in C_k ab, indem die Stichprobe D_S
    einmal durchlaufen wird.
  C_k' := Kandidaten in C_k, von denen man erwartet, daß sie min-support
    haben und Kandidaten von deren sämtlichen Eltern man erwartet,
    daß sie min-support haben.
  Bestimme den support der Kandidaten in $C_k' \cup C_{k-1}''$, indem
    die Datenbank D einmal komplett durchlaufen wird.
  Lösche alle Kandidaten aus C_k, deren Vorfahren (in C_k') nicht min-support haben.
  C_k'' := Verbleibende Kandidaten in C_k, die sich nicht in C_k' befinden.
  L_k := Alle Kandidaten aus C_k' mit min-support.
  Füge alle Kandidaten in C_{k-1}'' mit min-support zu L_{k-1} dazu.
  $k := k+1$;
end
Ausgabe := $\bigcup_k L_k$;

Eine wichtige Frage, die sich für diesen Algorithmus noch ergibt, ist die zu verwendende Größe der Stichprobe. Tabelle 10.6 zeigt Wahrscheinlichkeiten, daß der support eines itemsets in der Stichprobe kleiner als $0,9s$ ist, wenn der richtige support (d.h. der support in der gesamten Datenbank) s ist. Hierbei wurde angenommen, daß die Anzahl der Transaktionen, die ein itemset X enthalten, eine binomialverteilte Zufallsvariable ist. Verwendet man z.B. eine Stichprobengröße von $n = 10.000$ Transaktionen, so ist die Wahrscheinlichkeit, daß die Schätzung des Kandidaten supports kleiner als 0,9 % ist bei richtigem support $s = 1$ % ca. 59 % (entspricht einem Fehler 1. Art). Die Größe der Datenbank an sich spielt dabei keine Rolle.

	$s = 5\ \%$	$s = 1\ \%$	$s = 0{,}5\ \%$	$s = 0{,}1\ \%$
n = 1.000	0,76	0,95	0,97	0,99
n = 10.000	0,07	0,59	0,77	0,95
n = 100.000	0,00	0,01	0,07	0,60
n= 1.000.000	0,00	0,00	0,00	0,01

Tabelle 10.6: $P(support\ eines\ itemsets\ X\ in\ Stichprobe < 0,9s)$

Die Tabelle zeigt, daß bei abnehmendem min-support die Stichprobengröße wachsen sollte.

Aufwandsbetrachtungen für die Algorithmen Basic, Cumulate, EstMerge
Es wurde eine Datenbank mit 1.000.000 Transaktionen betrachtet. Es wurden 250 Wurzeln und je 5 Nachfolger pro Knoten verwendet (siehe [43]). Hierbei wurden verschiedene Bereiche untersucht:

1. Minimum-support
   Der minimum-support wurde von 2 % bis auf 0,33 % variiert. Cumulate und EstMerge waren zwischen 3 und 4 Mal schneller als Basic. Je mehr der min-support abnahm, desto größer wurde der Unterschied. Bei hohem support wurde die Differenz zwischen EstMerge und Cumulate immer kleiner, da es nur wenige Regeln gab und so die meiste Zeit für das Durchschauen der Datenbank verwendet wurde (siehe Abbildung 10.6).

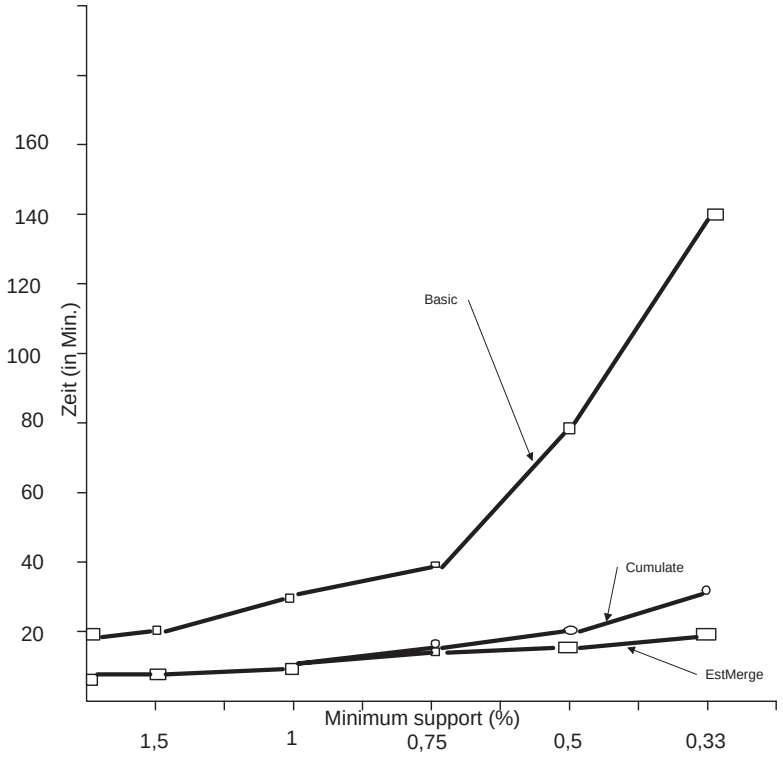


Abbildung 10.6: Minimum-support

2. Anzahl der Transaktionen
   Die Anzahl der Transaktionen variierte zwischen 100.000 und 10 Mio.. Dabei wurde

als Maßstab auf der y-Achse die verstrichene Zeit geteilt durch die Anzahl der Transaktionen verwendet. Eine Einheit wurde normalisiert als die Zeit, die Cumulate für 1 Mio. Transaktionen benötigt. Man sieht auch hier, daß Cumulate und EstMerge viel besser als Basic abschneiden (Abb. 10.7). Der größer werdende Unterschied zwischen EstMerge und Cumulate bei steigender Anzahl von Transaktionen kommt daher, daß bei konstanter Prozentzahl der Stichprobe zur Größe der Datenbank die Genauigkeit der Schätzungen des supports der Kandidaten steigt (siehe Tabelle 10.6.).

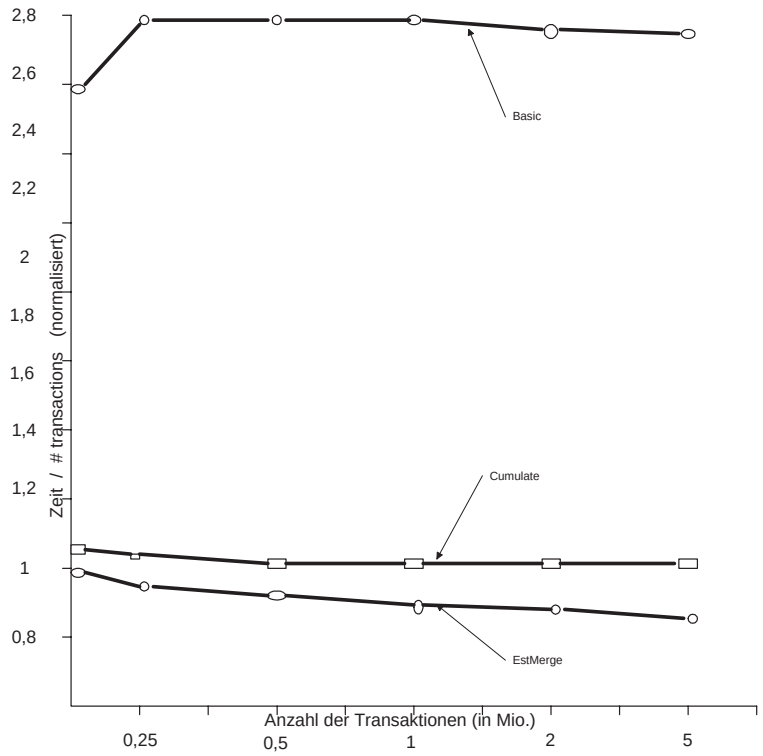


Abbildung 10.7: Anzahl der Transaktionen

3. Anzahl der items
   Bei variabler itemzahl von 10.000 - 100.000 und der hier verwendeten Speichertechnik erhöht sich im Taxonomiebaum, in welchem die items abgespeichert sind, die Anzahl der Ebenen. Die benötigte Zeit ändert sich bei Cumulate und EstMerge kaum, bei Basic hingegen erhöht sie sich jedoch stark (Abb. 10.8). Dies hat damit zu tun, daß Basic mehr Zeit damit verbringt, Kandidaten aus den Transaktionen zu finden. Die Größe der Transaktionen vergrößert sich nämlich durch das Hinzufügen der Vorfahren.

Resultat: Cumulate und EstMerge waren 2-5 mal schneller als Basic, wobei EstMerge zwischen 25-30 % schneller als Cumulate war.

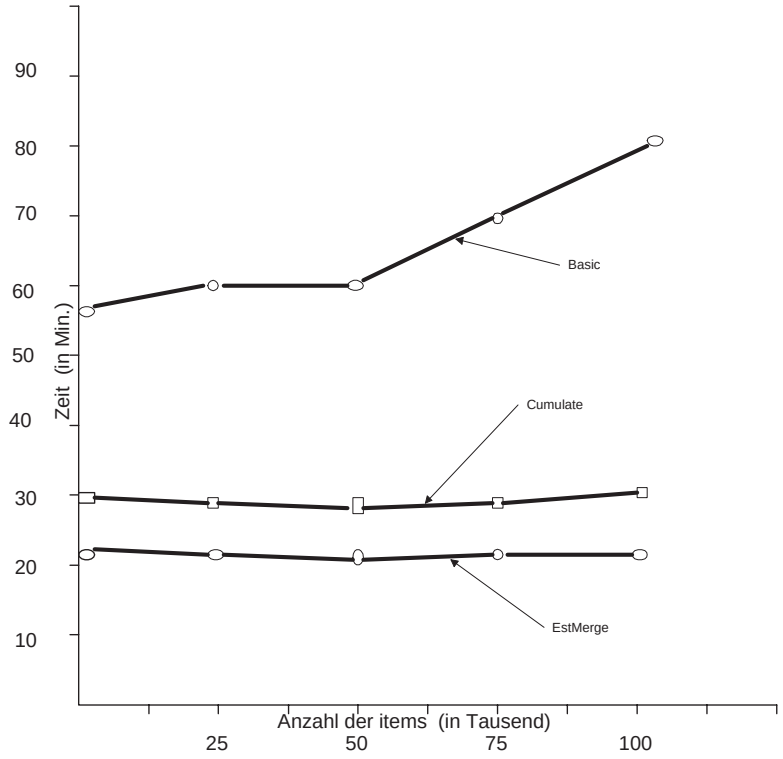


Abbildung 10.8: Anzahl der items

10.5 Schlußfolgerungen

Von den vorgestellten Algorithmen ist der Basic-Algorithmus der langsamste, da er alle supports durch komplette Datenbankdurchläufe bestimmt. Dagegen sind objektive Vergleiche der übrigen Algorithmen aufgrund der unterschiedlichen Rahmenbedingungen nicht durchführbar. Meiner Meinung nach wäre es möglich, daß SetM schneller ist als Discovery, Cumulate und EstMerge. SetM läßt nämlich die large itemsets innerhalb der Datenbank berechnen, während Cumulate bzw. EstMerge jede Transaktion einzeln als Anfrage an die Datenbank stellen und so den support bestimmen. Dies könnte sich evtl. in kleineren Netzwerken nachteilig auf den Zeitaufwand von Cumulate und EstMerge auswirken. Discovery stellt zwar kleinere Anfragen an das DB-System als SetM, diese jedoch häufiger. Auch hier ist möglicherweise SetM die schnellere Variante. Bedenklich erscheint mir jedoch, daß der Aufwand zum Sortieren der itemsets, der ja die untere Grenze $n * \log n$ hat, scheinbar nicht in die Aufwandsbetrachtung bei SetM eingeht (z.B. beim linearen Vervielfachen der Datenbank).

Kapitel 11

Data Warehousing in der Praxis

Reinhard Sablowski

Kurzfassung *In diesem Kapitel werden Beschreibungen von Data Warehouse-Projekten zusammengefaßt, die von Softwareunternehmen als Success Stories veröffentlicht wurden. Nach einem einheitlichen Fragenkatalog werden die Projekte strukturiert vorgestellt und deren wesentliche Probleme und Erfolgsfaktoren genannt. Als wesentliche Erfolgsfaktoren bei der Einführung des Data Warehousings sind aus der untersuchten Literatur die Unterstützung der oberen Führung des Unternehmens sowie die frühzeitige Zurverfügungstellung von Prototypen an die Endanwender erkennbar.*

11.1 Einleitung

In den letzten Jahren hat der Begriff des Data Warehousing derart an Bedeutung gewonnen, daß kein namhafter Hersteller von Datenbanksoftware mehr darauf verzichtet, ihn in seinen Prospekten zu erwähnen. In der Praxis ist der Erfolg der Einführung von Data Warehouses von vielen, häufig eher organisatorischen als DV-technischen Erfolgsfaktoren abhängig, so daß im Rahmen dieses Seminars der Ansatz, Erfahrungen bei der erfolgreichen Implementierungen von Data Warehouses in bestehenden Unternehmensstrukturen zu untersuchen, sinnvoll ist.

Zunächst sollen einige Beispiele von Data Warehouse-Implementationen, darunter ein ausführliches, aufgezählt und nach einem kurzen Fragenkatalog ausgewertet werden. Anschließend wird der Versuch unternommen, Gemeinsamkeiten der Ansätze zu ermitteln.

Wegen der hohen Aktualität der Aufgabenstellung waren in der Literatur noch wenige Quellen auffindbar. Daher wurden Informationen durch Suche im Internet beschafft. Diverse Suchmaschinen wie YAHOO und Internet Search wurden verwendet, um Adressen relevanter Seiten aufzufinden. Die Newsgruppe comp.databases.olap wurde regelmäßig verfolgt; Hinweisen auf Internet- Adressen oder weiteren Quellen wurde nachgegangen, auch persönliche Nachfragen and Autoren wurden gestellt.

11.1.1 Disclaimer

Durch die nur eingeschränkt verfügbare Primär- und fast überhaupt nicht vorhandene Sekundärliteratur kann den Aussagen dieser Arbeit keine wissenschaftlich abgesicherte Bedeutung beigemessen werden. Im wesentlichen wurden Beschreibungen ausgewertet, die von Software- und Beratungsfirmen über erfolgreich abgeschlossene Projekte veröffentlicht wurden, sog. *Success Stories*.

Wichtig zu verdeutlichen ist die hieraus entstehende zwangsläufige Selektion, den *Failure Stories* sind verständlicherweise nicht publiziert.

11.1.2 Der Fragenkatalog

Um eine halbwegs vergleichbare Darstellung der verschiedenen, zum großen Teil von Beratungs- und Softwarefirmen verbreiteten Quellen herzustellen, wurde den einzelnen Berichten ein Schema zugrundegelegt.

- Das Unternehmen
- Gründe zur Implementation des Data Warehouse
- Art und Umfang der verarbeiteten Daten
- Eingesetzte Technik
- Hauptschwierigkeit beim Erreichen des Zieles
- Schlüsselfaktoren für den Erfolg
- Weiteres, geplantes Vorgehen

11.2 Kurzportraits von Data Warehouse-Projekten

11.2.1 Entergy Services

Das Unternehmen

Entergy Services [46] ist der fünftgrößte amerikanische Stromversorger, mit 2,5 Millionen Kunden und einem Jahresgewinn von etwa 10 Mrd DM. Seit Gründung war Entergy durch ein Monopol geschützt; die Preise für Entergys Leistungen wurden durch öffentliche Kommissionen festgelegt. Angesichts einer kommenden Deregulation des amerikanischen Energiemarktes erwartet Entergy nun erstmals Wettbewerb um seine Kunden.

Gründe zur Implementation des Data Warehouse

Zur Verbesserung der Wettbewerbsfähigkeit benötigt Entergy eine wesentlich bessere und detailliertere Übersicht über die Kosten der Energieerzeugung und bessere Daten über ihr Profil an derzeitigen und potentiellen Kunden. Es werden sowohl mehr als auch detailliertere Informationen als bisher benötigt. Dabei sind bereits große Datenmengen in verschiedenen relationalen und nicht-relationalen Großrechnersystemen gespeichert, dort aber nicht effektiv für die Firmenanalysten zugreif- oder verwendbar. Berichte sind ausschließlich als Standardreports, die von firmeneigenen Programmierern erzeugt werden, verfügbar. Kundenbezogene Daten werden in einer hierarchischen IMS-Datenbank, weitere kaufmännische Daten in relationalen DB2-Tabellen verwaltet.

Die Standardreports waren in der Regel gut geeignet, mit der Suche nach Informationen zu beginnen, erlaubten den Analysten aber nicht, sich in die Details hineinzuarbeiten, um beispielsweise eine größere Kostenänderung nachzuvollziehen.

Art und Umfang der verarbeiteten Daten

Das größte der sechs eingesetzten Data Warehouses ist die Finanzbuchführung mit derzeit 24 GB. Das am stärksten wachsende Warehouse betrifft die Umsätze und nimmt mit einer Größe von 22 GB doppelt so stark wie die Finanzbuchführung zu. Insgesamt werden 61 GB an Daten verwaltet, für Anfang 1996 wurden 100 GB erwartet. Während die täglichen Transaktionen weiterhin auf den fünf Großrechnern der Firma abgewickelt werden, finden regelmäßige Datenübertragungen in die Warehouses statt. Tägliche, wöchentliche und monatsweise Batch- Übertragungen werden durchgeführt.

Eingesetzte Technik

Entergy verwendet zur Speicherung der Daten vier SyBase SQL-Server auf einer SunSPARC 2000. Zum Datenzugriff werden Microsoft Access und selbstgeschriebene Powerbuilder-Applikationen genutzt.

Hauptschwierigkeit beim Erreichen des Zieles

Eine über die Jahre gewachsene Struktur von Novell-Netzen mit insgesamt tausenden von PC-Computern war zusammenzuführen. Dabei hatte jede Abteilung das ehemals einzeln entstandene Netzsegment unterschiedlich ausgestaltet. So gut wie alle marktverfügbaren Kommunikationsprotokollen wurden auch vorgefunden.

Schlüsselfaktoren für den Erfolg

Entergy betrachtet die sorgfältige Ausdefinition des Datenmodelles als erfolgskritisch. Des weiteren stand die oberste Unternehmensleitung hinter dem Projekt. Das Data Warehouse wurde von Beginn an als das zukünftige Hauptwerkzeug zur Lieferung von Managemententscheidungsdaten gesehen.

Weiteres, geplantes Vorgehen

Wegen des rapiden Wachstums der Warehouses wird nicht ausgeschlossen, daß zukünftig ein weiterer Server zur Archivierung der Historie oder ein Übergang zu einer verdichteten Speicherung erforderlich wird.

11.2.2 Woolworth, Großbritannien

Das Unternehmen

Woolworth [47] ist eine große Handelskette, die in Großbritannien 779 Warenhäuser unterhält. Im Jahre 1994 wurden 330 Millionen Kundentransaktionen durchgeführt, davon 50 Millionen allein in der Vorweihnachtszeit. Im Jahr vor der Untersuchung hat Woolworth zum ersten mal seit sieben Jahren das Ziel nicht erreicht, seine Umsätze signifikant zu steigern. Als Folge davon wurden Marktanteile verloren, obwohl der Markt leicht gewachsen ist.

Gründe zur Implementation des Data Warehouse

Woolworth hat drei Jahre lang ein E-POS (Electronic Point of Sales)-System eingeführt. Ein erstes Entscheidungsunterstützungssystem wurde vor zwei Jahren auf einem AS/400-Rechner erstellt. Für das Folgesystem wurde schnell klar, daß zur Auswertung der E-POS-Daten auf Hochgeschwindigkeitstechnologie umgestiegen werden mußte. In der Weihnachtssaison des folgenden Jahres, 1995, wurde ein Test gefahren, der ergab, daß der Einsatz der Daten zur automatisierten, zentralen Lagernachfüllung allein Millionen englische Pfund durch geringere Lagerhaltung und erhöhte Umschlagsmengen erwirtschaften konnte.

Art und Umfang der verarbeiteten Daten

Im Warehouse sind Umsätze und Lagerbewegungen aus dem E-POS-System enthalten, die mit Vertriebs-, Lager-, Planungs- und Vorhersagedaten kombiniert werden. Derzeit sind 250 GB Daten enthalten. Daten werden über mehr als 20 Schnittstellen aus operationalen Systemen gewonnen.

Eingesetzte Technik

Wegen der überaus großen Datenmenge wurde von Anfang an, als die AS/400-Lösung sich als zu leistungsschwach erwies, auf massiv parallele Hardware gesetzt. Zum Einsatz kommt ein Tandem K20000-System, auf dem die relationale Datenbank NonStop SQL/MP läuft.

Hauptschwierigkeit beim Erreichen des Zieles

Woolworth hat keine wesentlichen Schwierigkeiten erlebt. Der IT-Services- Controller merkt jedoch an, daß er lieber ohne den immensen Zeitdruck gearbeitet hätte, der durch den dringenden Bedarf an der Data Warehouse- Lösung erzeugt wurde. Dafür hätte er gern mehr Zeit darauf verwendet, Anwender zu schulen und Prozesse gründlich durchzudesignen.

Schlüsselfaktoren für den Erfolg

Wesentlicher Schlüsselfaktor war die Erkenntnis, daß allein die automatisierte Lagerergänzung bereits die Investition in das Data Warehouse bezahlt macht, und zusätzliche Nutzen an vielen Stellen leicht erkennbar waren. Des weiteren bemerkt Woolworth, daß die eingesetzte, massiv parallele Infrastruktur in Verbindung mit der Beratung durch das Herstellerunternehmen wesentlicher Erfolgsfaktor war.

Weiteres, geplantes Vorgehen

Bei der Einrichtung des bestehenden Systems wurde großer Wert auf eine schnelle Entwicklung gelegt. Zur Zeit wird beabsichtigt, High-Performance Merchandising und später Warenkorbanalysen und detaillierte Kundendaten zur Unterstützung von Mikromarketing-Strategien zu verwenden.

11.2.3 Ingram Book Company

Das Unternehmen

Ingram [48] ist ein amerikanischer Buchgroßhändler, der mehr als 48000 Verkaufsstellen und Büchereien versorgt. Jedes Jahr werden über 147 Millionen Bücher, Tonkasssetten und Multimedia-CD-ROMS ausgeliefert. Dabei werden 90% der USA-weiten Bestellungen am folgenden Tag ausgeliefert.

Gründe zur Implementation des Data Warehouse

Die interne EDV-Abteilung hatte festgestellt, daß die Benutzer ihrer EDV zunehmend Ad-Hoc-Reports und Download-Anforderungen an das System stellten. Der vorliegende Bericht stellt den Ausbau eines bereits bestehenden Data Warehouses dar. Ingram hat große Anstrengungen unternommen, gute Software zum Thema Knowledge Discovery und Hardware für sein Data Warehouse zu finden. Dabei wurden die Autoren fündig - der Bericht geht stark auf das IBM-Produkt Visual Warehouse ein.

Art und Umfang der verarbeiteten Daten

Bei Ingram wird ein unternehmensweites Data Warehouse in Verbindung mit mehreren Bereichs-Warehouses eingesetzt. Aussagen über die Datenmengen werden nicht explizit gemacht, die im Unternehmensblickpunkt angegebenen Daten lassen aber erhebliche Datenmengen vermuten. Bei 147 Mio. Medien pro Jahr und nur 40 Byte je Datensatz wächst das Warehouse, Stammdaten nicht eingerechnet, um jährlich 5,5 GB.

Eingesetzte Technik

Ingram setzt als Software IBM s Visual Warehouse ein. Hardwareseitig wird ein IBM 320 Pentium (95 MHz, 64MB Hauptspeicher, 6,5 GB Plattenplatz) eingesetzt, der über ein Novell-Netzwerk verfügbar ist. Ein Token-Ring-Netz verbindet den Server mit den IBM MVS/ESA-Großrechnern. Deren Datenbanksystem ist DB2. Der Server wird unter OS/2 betrieben, das installierte Datenbanksystem ist DB2 für OS/2.

Hauptschwierigkeit beim Erreichen des Zieles

Bei Ingram sind die ersten Schritte zum Data Warehouse von der internen EDV-Abteilung ausgegangen. Obwohl vom großen Interesse seitens der Anwender berichtet wird, wird berichtet, daß zunächst kaum Gelder für das Projekt verfügbar waren und erst mit dem zunehmenden Interesse erster Anwender Firmengelder in verwendbarem Maßstab verfügbar wurden. Auch zum Zeitpunkt der Erstellung des Berichts ist die EDV-Abteilung noch stark von sich aus daran interessiert, ihre Data Warehousing-Lösungen intern im Hause zu demonstrieren. Anscheinend ist zu jener Zeit die allgemeine, interne Nachfrage der Bereichs-EDV-Abteilungen noch nicht stark ausgeprägt.

Schlüsselfaktoren für den Erfolg

Ingram unterstreicht stark, daß die enge Partnerschaft mit IBM ein Schlüssel zum Erfolg bei der Einführung des Data Warehousing war.

Weiteres, geplantes Vorgehen

Aufgrund des erwarteten Anstiegs der Systemlast wird der Anschaffung eines RS/6000-Rechners entgegengesehen, dessen Datenbanksystem DB2/6000 sein soll.

11.2.4 SNCF Die französische Eisenbahn

Das Unternehmen

Die SNCF[1] [49] setzt täglich 400 TGV s[2] und 1200 klassische Züge ein. TGV s sind Hochgeschwindigkeitszüge, die im Durchschnitt mit etwa 300 km/h fahren und bisher

[1] Societe Nationale des Chemins de Fer Francais
[2] Train a Grande Vitesse

eine Spitzengeschwindigkeit von 515,3 km/h erreicht haben. Die SNCF ist die einzige bedeutende Eisenbahngesellschaft Frankreichs.

Gründe zur Implementation des Data Warehouse

Um den Wünschen der Fahrgäste zu entsprechen und dauerhaft die Mittel zur Finanzierung des TGV-Ausbaues sicherzustellen, muß SNCF in der Lage sein, die Nutzung eingesetzter Ressourcen zu optimieren und dafür Verkehrsaufkommen, Preise und weitere Faktoren vorherzusagen. Das bislang eingesetzte Logistik- und Reservierungssystem war, obwohl im Einsatz, kaum imstande, 1600 tägliche Züge effizient zu versorgen. Darüber hinaus existierten starre Preisstrukturen; ohne eine flexible Fahrpreisgestaltung wurde häufig am Bedarf vorbeigewirtschaftet. SNCF war dringend darauf angewiesen, die maximale Zahl an Sitzplätzen zu hohen Preisen zu verkaufen, wo und wann immer der Bedarf groß war, während Sonderangebote die weniger populären Strecken fördern sollten. Das Management forderte ein ausgefeiltes Passagiermanagementsystem, das, ähnlich dem bei Luftverkehrsgesellschaften eingesetzten, zusätzlich leistungsfähige Werkzeuge zur Unterstützung von Management-Entscheidungen anbietet.

Art und Umfang der verarbeiteten Daten

Die SNCF verarbeitet ihre Daten in einem zweigeteilten System. Das *Thales* genannte Reservierungssystem steht neben der als *Aristoteles* bezeichneten Back-Office-Lösung. Zusammen bilden Sie das Gesamtsystem *Sokrates*. Über den Umfang der vorliegenden Daten wird nur vage der Terabyte[3]-Bereich genannt. Mit dem neu aufgebauten Data Warehouse ist das Management beispielsweise in der Lage zu erkennen, wo die Reisenden der Route Paris-Lyon ihre Tickets gekauft haben, ob sie Mahlzeiten oder Mietwagen angefordert hatten und wie viel sie für die Fahrkarte bezahlt haben. Die Daten kommen jede Nacht aus dem Sokrates-System. Das Reservierungssystem kann aufgrund der im Data Warehouse vorhandenen Daten jetzt vorhersagen, wieviele Leute voraussichtlich einen Zug reservieren werden, schätzen, wie viele davon nicht erscheinen und die Plätze entsprechend überbelegen. Anschließend werden die verfügbaren Plätze in Preiskategorien eingeteilt, um die Passagiererträge weitmöglichst zu steigern.

Eingesetzte Technik

Bei der SNCF kommt die AT&T Enterprise Information Factory (EIF) zum Einsatz. EIF-Systeme unterscheiden sich von Data Warehouses dadurch, daß neben dem reinen Informationszweck an das Management auch automatisiert Aktionen aufgrund der enthaltenen Daten ausgelöst werden. Die Daten werden auf einem massiv parallelen System verarbeitet.

[3] $1TB = 1024GB = 1.099.511.627.776 Byte$

Hauptschwierigkeit beim Erreichen des Zieles

Wesentliche Schwierigkeit bei der Einführung der EIF-Lösung war die teilweise Ablösung alter Systeme, ein Ansatz der über das reine Data Warehousing hinausgeht. Die eigentlichen Warehousing-spezifischen Probleme sind daneben nicht spürbar hervorgetreten.

Schlüsselfaktoren für den Erfolg

Die bisher eingesetzten Mittel zur Einführung der EIF/Data Warehousing- Lösung bei der SNCF belaufen sich auf ca. DM 60 Mio. Wesentlicher Erfolgsfaktor war, daß dem Management die Erkenntnis vermittelt werden konnte, allein durch die verbesserte Kapazitätsauslastung und den erhöhten, kundenverfügbaren Service, diese Kosten wieder rechtfertigen zu können. Aus technischer Sicht werden die technische Beratung und die professionellen Dienste der ebenfalls für Hard- und Softwarelieferung verantwortlichen Firma AT&T genannt.

Weiteres, geplantes Vorgehen

Durch die erstmals zentral gut strukturiert vorliegenden Informationen sind bei den EDV-Anwendern der SNCF neue Bedürfnisse entstanden. Während bisher nach einzelnen Routen ausgewertet wurde, stehen jetzt beliebige Daten des gesamten Schienennetzes gleichzeitig zur Verfügung. Hieraus werden deutliche Steigerungen der Systemlast mit den entsprechend erforderlichen Erweiterungen der eingesetzten Infrastruktur erwartet. Des weiteren hat die SNCF das Thales-System in der Zwischenzeit an Eurostar lizensiert. Eurostar wird den TGV durch den Großbritannien mit Frankreich verbindenden Chunnel (Channel Tunnel) betreiben.

11.2.5 3M Minnesota Mining and Manufacturing

Das Unternehmen

3M ist die Kurzform für Minnesota Mining and Manufacturing, ein Unternehmen, das seit Jahrzehnten zu den großen, weltweit agierenden Mischkonzernen gehört. 3M setzt im Jahr etwa 15 Mrd. Dollar um. Bekannte Produkte sind zum Beispiel die weltweit sehr erfolgreichen PostIt- Haftnotizzettel.

Gründe zur Implementation des Data Warehouse

Bei 3M ist man der Ansicht, daß Kundenzufriedenheit zu den höchsten Zielen des Unternehmens gehört. Durch die stark in Sparten geteilte Struktur des Konzerns war es häufig schwierig, Kunden gegenüber wie eine einzige, zusammengehörige Firma aufzutreten. Zentral gepflegte, unternehmensweit zugängliche Daten zu haben wurde als wesentliches Schlüsselkriterium zur Verbesserung der Situation erkannt. Ein 3M-Mitarbeiter

führt aus, daß mehrfache Datenquellen mit inkonsistenten Datendefinitionen vorhanden waren. Die IT-Abteilung strebte angestrengt an, eine Datenquelle zu erstellen, die dann die offizielle Version darstellen sollte.

Art und Umfang der verarbeiteten Daten

Der umfassende Ansatz 3M s zur allgemeinen Unternehmensdatenmigration in das Data Warehouse bedingt verschiedenste Arten von Daten. Bei Einführung des Systems waren 70GB Plattenplatz vorgesehen. Die nächste Erweiterung war bereits geplant.

Eingesetzte Technik

Softwareseitig wird das Produkt HP Open Warehouse von Hewlett-Packard eingesetzt. Zudem wird HP Intelligent Warehouse verwendet. In der Warehouse-Konfiguration von 3M ist ein HP 9000 Modell der T500 Klasse mit symmetrischem 3-Wege-Multiprocessing integriert. Er ist gleichzeitig Warehouse-Hub und Datenserver. Die Grundkonfiguration enthält 70GB Plattenplatz und einen Hauptspeicher von 1024 MB. Da Open Warehouse verschiedene Datenbankensysteme unterstützt, war 3M in der Lage, sich eine Plattform auszuwählen. Zum Einsatz kommt das unter UNIX laufende Red Brick-System zur Datenhaltung sowie Microsoft Access als Front End.

Hauptschwierigkeit beim Erreichen des Zieles

Der umfassende Ansatz der Unternehmensweiten Datenmodellneubildung stellt die größten, nicht Data Warehouse-spezifischen Anforderungen an das IT- Team. Data Warehouse-seitig ist vor allem das dezentrale Modell mit spartenweit teilweise unabhängig pflegbaren Teil-Warehouses eine aufwendige Lösung.

Schlüsselfaktoren für den Erfolg

Die unmittelbare Unterstützung der obersten Führungsebene und die grundsätzlich erkannten Probleme bei der Gewinnung verläßlicher Daten aus den vorher vorhandenen Systemen war wesentlicher Erfolgsfaktor. Zudem wird die von HP bezogene Software als gelungene Entscheidung gelobt. Besonders beeindruckend sei die Möglichkeit, verteilte Datenquellen so zu sehen, als seien sie an einem Ort vereint.

Weiteres, geplantes Vorgehen

3M wird sein Data Warehousing-Projekt fortsetzen.

11.3 Warehousing aus der Sicht der oberen Führung - Allied Signal

Allied Signal ist ein in Sparten geteiltes, weltweit tätiges Technologie- Unternehmen mit 12 Mrd. Dollar Jahresumsatz. Die Sparten sind Luft- und Raumfahrt, Automobilzulieferungen und Spezialmaterialien wie Fasern, Chemikalien, Kunststoffe und Schaltkreislaminaten. Der Leiter der Sparte Luft- und Raumfahrt, Dan Burnham, hat ein Data Warehouse-Programm selbst maßgeblich initiiert und vorangetrieben. Er wollte im Januar 1993 erreichen, daß Geschäftsberichte schneller verfügbar und besser organisiert waren. Burnham verlangte beispielsweise Daten über die zehn besten Kunden je Geschäftsfeld mit Umsatzzahlen und Gewinn je Kunde oder Produktlinie, oder Liefermengen nach Inland/Ausland oder zivilen/militärischen Kunden. Solche Informationen waren in den einzelnen Geschäftseinheiten verfügbar, wurden jedoch bis dato nicht auf Spartenebene zusammengefaßt. Manche Daten waren aber schlicht nicht maschinell verfügbar. Um zum Beispiel die Qualitätskosten eines Artikels zu ermitteln, mußte jede Geschäftseinheit separat angerufen und nach den Nacharbeitskosten gefragt werden. Diese Daten wurden anschließend mit weiteren Daten verknüpft und ausgewertet. Die größte Hürde auf dem Weg zu brauchbaren Daten für Herrn Burnham war die Verteiltheit der Daten in den einzelnen Geschäftseinheiten. Dutzende von Geschäftsfeldern mußten Daten aufliefern, die zügig in ein einziges System geladen werden mußten. Ein eingeschaltetes Beratungsunternehmen führte das System Comshare Commander vor und stieß nach Vorführung verschiedener Drill-Down-Möglichkeiten auf breite Zustimmung.

Die grundsätzliche Erstellung des Systems dauerte drei Monate. Nach einem weiteren Monat mit Schulungen und durch Benutzerkommentare ausgelösten Feintunings wurde das System 150 Sachbearbeitern an 15 verschiedenen Orten verfügbar gemacht. Viele der anfänglichen Benutzer des Systems waren Manager, die dafür verantwortlich waren, daß die Daten rechtzeitig im Warehouse des Präsidenten verfügbar waren. Zusätzlich wurde ein Mitarbeiter jedes Standortes zum Support-Spezialisten ausgebildet. Dabei war der Ansatz, daß dieser Mitarbeiter im weiteren Einsatz höchstens ein Prozent seiner Zeit tatsächlich mit Support verbringen sollte.

Bis dorthin war das Warehouse für die Manager der Geschäftseinheiten schlicht ein praktisches System, ihre Daten der Sparte anzuliefern. Schnell wurde jedoch erkannt, daß, wenn das System langfristig erfolgreich sein sollte, nicht nur dem Präsidenten ein Nutzen entstehen sollte.

Hierzu wurde ein Rapid-Prototyping-Ansatz gewählt. Statt eines ausführlichen Lasten- und Pflichtenheft-Ansatzes wurden schnell nach intuitiver Nützlichkeit erstellte Module entwickelt und den Endanwendern verfügbar gemacht, deren rücklaufende Kommentare zur schnellen Verbesserung genutzt und so zügig für viele Leute ein Anreiz geschaffen, das System zu verwenden. Durch Rapid Prototyping ist die Nutzung des EIS (Executive Information System) stark gewachsen. Aus den 150 Arbeitsplätzen der Sparte Luft- und Raumfahrt ist ein konzernweit eingesetztes 500-Platz-System geworden.

Zu Beginn des Warehouse-Projektes wurden 29 Kennwerte der Unternehmensperformance von Allied Signal ermittelt. Eine Anwendung zur Ermittlung der Kundenzufriedenheit auf Basis von Retourzahlen und Lieferpünktlichkeit wurde entwickelt; diese Kenndaten

konnten nun erstmalig im grafischen Verlauf über die Zeit und im Vergleich verschiedener Geschäftseinheiten betrachtet werden.

Zudem wurde ein System zur Vorhersage von Flugzeugauslieferungen erstellt. Allied Signal speichert Daten über jedes Flugzeug, das innerhalb der nächsten zehn Jahre weltweit produziert werden wird, die Hersteller, die Zulieferer und die Triebwerksbauer. Wesentlicher Erfolgsfaktor war der Support durch das oberste Management, der eine Initialzündung und stetig weiterer Antrieb für das gesamte Projekt war.

11.4 Ausführliches Beispiel

Als ausführliches Beispiel soll die Einführung eines Data Warehouses bei Health Canada, Laboratory Centre for Desease Control (CDC) vorgestellt. Es basiert auf Oracle Version 7.1.4 und hält derzeit etwa 18 GB Daten bei einer jährlichen Wachstumsrate von 2 GB.

11.4.1 Zielgruppe und Leistungsanforderung

Am CDC wurde ein Data Warehouse implementiert, daß die Krankheitsdaten der Bürger des Staates Kanada aufbereitet. Hierbei wird im wesentlichen verdichtet auf geographische Lage, Alter des Erkrankten, Geschlecht des Erkrankten sowie die Art der Erkrankung.

Benutzer des Warehouses sind beispielsweise

- Parlamentsabgeordnete, die Informationen über die akute Ausbreitung einer Seuche benötigen
- Das Presseamt des Gesundheitsministeriums zur Erstellung von Auswertungen zum allgemeinen Gesundheitszustand der Bevölkerung
- Forscher, die Zeitreihen über die Ausbreitung spezieller Erkrankungen untersuchen
- Epidemiologen, die allgemeine Grundlagenforschung anstellen
- Krankenversicherungen, die wirtschaftliche Auswirkungen von Krankheiten vorhersagen möchten

Während der Konzeption des Warehouses waren Mitglieder dieser Gruppen mehrfach an offenen Diskussionen beteiligt. Risiken und Möglichkeiten der Verfügbarkeit der Daten wurden diskutiert.

Wichtig für die einfache Nutzbarkeit der Daten war, daß die historische Uneinheitlichkeit der Schlüsselbegriffe und Bezeichnungen aufgelöst wurde. Zum Beispiel konnte im Jahr 1986 der Schlüssel 35 für Ontario stehen, im Jahr 1989 der Schlüssel 935. Wichtig war, zu erreichen, daß der Benutzer einheitlich *Ontario* abfragen kann.

Weitere Anforderungen waren

- Überblicksfunktionen über die Seuchen im ganzen Staat
- Drill-down-Möglichkeiten entlang der Achsen Krankheit, Staat und Alter
- Inkrementelles Erzeugen von Abfrageergebnissen und die Möglichkeit, zu bestehenden Ergebnissen weitere hinzuzufügen
- Schätzung von Kosten und benötigten Ressourcen
- Zweisprachigkeit (Englisch und Französisch)

11.4.2 Datenhaltung und -strukturen

Aufgrund der höchst unterschiedlichen Anforderungen der einzelnen Nutzergruppen wurde entschieden, die Daten auf der niedrigstmöglichen Detailebene vorzuhalten. Hierdurch hat der Anwender die Möglichkeit, sich über Drill-down-Techniken aus Übersichtsberichten bis hinunter zum einzelnen Erkrankungs- oder Verletzungsfall vorzuarbeiten. Dieser Teil des Datenbankenwurfes enthielt keine besonderen Komplexitäten. Auf Design-Ebene wurden alle Tabellen normalisiert angelegt. Darüberhinaus wurden verschiedene Verdichtungsstufen angelegt, zum Beispiel für den Ort von Location[4] über County[5], Province, Region und Country[6]. Als besondere Schwierigkeit wurde erkannt, daß die von den einzelnen Gesundheitsämtern angelieferten Daten wechselnden Verschlüsselungen unterlagen. Die Ortskennungen im Jahre 1986 unterschieden sich signifikant von denen des Jahres 1989. Unter anderem aus diesem Grund wurde eine benutzerdefinierbare Gruppierung zugelassen, die solche Schlüsselinkonsistenzen zu neuen, sinnvoll verwendbaren Gruppen zusammenfassen konnten.

Ähnlich wurde für das Alter und die Krankheitsursache verfahren, beim Alter beispielsweise verdichtet nach Wochen, Monaten und Jahren. Die Zeit zum Upload der Daten von fünf Jahren, ca. 2 GB, betrug vier Stunden.

11.4.3 Benutzeroberfläche und Antwortzeiten

Da das Datenmodell darauf ausgelegt war, so gut wie alle Abfragedetaillierungslevel der Kombinationen Krankheitsgrund, Ort und Alter zu unterstützen, verursachte der große Umfang der Daten sehr lange Antwortzeiten. Deshalb wurden Überlegungen angestellt, ob andere Ansätze erfolgversprechender sein konnten.

Die an das System gestellten Anforderungen konnten in zwei prinzipielle Kategorien eingeteilt werden. Zunächst die Abfragen, die auf Detaillevel von Wissenschaftlern für nicht vorab, das heißt, zur Ladezeit spezifizierbare Forschungen gestellt wurden - hier war lediglich über die besonders ansprechende und intuitive Benutzeroberfläche eine Steigerung der Benutzerfreundlichkeit machbar.

[4] Ort
[5] Landkreis
[6] Land

Die andere Abfragekategorie verwendet normalerweise vorabdefinierbare Kombinationen von Krankheitsgrund, Ort und Alter. Zum Beispiel wurde häufig die Frage nach der Häufigkeit von Krebs nach Provinzen und in Altersstufen von fünf Jahren gestellt. Solche Sichten, die für häufig gestellte Abfragen benötigt wurden, wurden als Tabellen gespeichert (materialisiert). Diese Tabellen müssen bei jedem neuen Laden von Daten wieder erzeugt werden, was in der Regel jährlich, für manche Datenquellen aber auch monatlich geschieht. Die Systementwickler erwarten allerdings von neuen, Data Warehouse-Techniken unterstützenden Eigenschaften der Oracle-Versionen 7.2 und 7.3 weitere verbesserungen der Unterstützung von Materialisierungen.

11.4.4 Selbst Entwickeln oder Software einkaufen?

Bei der Auswahl der Werkzeuge, die dem Endanwender für den Datenzugriff zur Verfügung gestellt werden sollten, wurden nicht nur die Anforderungen an die Rückgewinnung teilweise verdichteter Daten aus dem Warehouse sondern insbesondere auch die günstige Möglichkeit der Weiterverarbeitung in Tabellenkalkulationen, Präsentationen oder für Publikationen in Betracht gezogen. Die Hauptanforderung war, daß Selektionen über die Achsen Krankheitsgrund, Ort und Alter über ein Formular möglich war und kein Endanwender sich mit der Pflege von Dateien der ihn interessierenden Kombinationen von Schlüsselnummern und Codes befassen mußte. Zudem sollten die Kennwerte einmal gestellter Abfragen für zukünftige Abfragen aufbewahrt werden. Das Werkzeug für die Achsenselektion wurde mit Hilfe von Oracle CASE 5.0 mit SQL*Forms 3.0 entwickelt. Ein Upgrade auf Oracle*Forms war geplant. Zur weiteren Verarbeitung wurde davon abgesehen, selbst GUI-Module zu definieren. Vielmehr wurde über eine ASCII-Schnittstelle auf die vielfältigen Möglichkeiten externer, auch PC-gestützter Systeme wie Lotus, Harvard Graphics oder MapInfo zurückgegriffen.

11.4.5 Schlußfolgerung

Die Einführung des Data Warehouses am Health Canada, Laboratory Centre for Disease Control war aus folgenden Gründen erfolgreich:

- Es wurden die Anforderungen aller in Frage kommender Benutzer in Betracht gezogen und nicht nur die einiger Intensiv-Nutzer
- Eingefahrene, inzwischen De-Facto-Standardverfahren wurden daraufhin untersucht, ob sie tatsächlich den Geschäftsprozeß abbilden oder lediglich eingefahrene, den Einsatz sonst verfügbarer Technologie behindernde Gewohnheiten waren
- Das Warehouse baut auf einem soliden Datenmodell auf, das durch mehrere Stufen des Prototyping verifiziert wurde. Obwohl Denormalisierungen manchmal verlockend schienen, wurde der Flexibilität von Benutzerabfragen Vorrang gegeben
- Hardware wurde als notwendiges, der Aufgabe anzupassendes Werkzeug und nicht als gegebene Welt begriffen.

11.5 Zusammenfassung

Insgesamt fallen nachstehende Punkte auf.

11.5.1 Auffälligkeiten

Support durch die oberste Führung

Der Support durch die oberste Führung eines Unternehmens wird von mehreren Berichten hervorgehoben. Vier von fünf hier betrachteten Wirtschaftsunternehmen geben dies explizit an. Bei Entergy und Allied Signal ist der Support die wesentlich bestimmende Erfolgsgröße, bei Woolworth und der SNCF besteht ein mittelbarer Erfolgseinfluß. Lediglich bei Ingram spielt der Support der obersten Führung keine wesentliche Rolle

Massiv parallele Hardware

Woolworth und die SNCF betonen stark, massiv parallele Hardware als erfolgskritischen Faktor einzusetzen. Bei 3M wird symmetrisches Multiprocessing verwendet, die eher kleine Lösung von Ingram kommt mit einem IBM-PC aus. Entergy verwendet zur Speicherung der Daten vier SyBase SQL-Server auf einer SunSPARC 2000.

Front Ends

Sowohl Entergy als auch 3M setzen als Front-End Microsoft Access ein. Bei Entergy wird zudem Powerbuilder verwendet. Die weiteren Unternehmen setzen die zum Warehouse-System gehörigen Front-Ends ein.

11.5.2 Bewertung

Der Erfolg von Data-Warehouse-Projekten hängt, soweit aus der Literatur erkennbar, in starkem Maße von der Unterstützung der oberen Führung des Unternehmens ab. Ebenfalls erfolgskritisch ist, daß die Mitarbeiter und Anwender des projektierten Warehouses bereits in frühen Phasen Ergebnisse der durchgeführten Arbeiten verfügbar haben (Prototyping). Häufig bringen diese Prototypen den Anwendern erst zu Bewußtsein, wie umfangreich und wertvoll die bisher ungenutzt im Unternehmen vorhandenen Daten sind. Die eingesetzte Hardware muß selbstverständlich Antwortzeiten zulassen, die den Anwender nicht verärgern, spielt darüber hinaus allerdings keine wesentliche Rolle. Wichtiger für den Erfolg scheint eine Warehouse-Software zu sein, die die Administration des Warehouses mit geringem Aufwand möglich macht. Die Führung von Meta-Daten wird in diesem Zusammenhang in mehreren Projekten als besonders wichtig eingeschätzt, da ohne sie der Administrationsaufwand des Warehouses sehr groß wird.

Literaturverzeichnis

[1] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. *Proceedings of 100th Anniversary Meeting of the Information Processing Society of Japan,* pages 7-18, Tokyo, Japan, October 1994.

[2] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. Technical report, Dept. of Computer Science, Stanford University, 1995.

[3] J. Chomicki. History-less checking of dymamic integrity constraints. In *Proceedings of the Eighth International Conference on Data Engineering,*pages 557-564, Phoenix, Arizona, February 1992.

[4] IEEE Computer. *Special Issue on Heterogeneous Distributed Database Systems,* 24(12), December 1991.

[5] A. Gupta, H.V. Jagadish, and I.S. Mumik. Data integration using self-maintainble views. Technical memorandum, AT&T Bell Laboratories, November 1994.

[6] Jennifer Widom.*Research Problems in Data Warehousing.* Proc. of 4th Int'l Conference on Information and Knowlege Management (CIKM), Nov. 1995

[7] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The stanford Data Warehousing Project. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing* 18(2):41-48, June 1995.

[8] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse.* QED Publishing Group, Boston, Massachussetts,1993.

[9] W. Labio and H. Garcia-Molina. Efficient snapshot defferential algorithms in data warehousing. Technical report, Dept. of Computer Science, Stanford University,1995

[10] A. Levy and Y. Sagiv. Queries independent of updates in *Proceedings of the Ninetenth international Conference on Very Large Data Bases,* pages 171–181, Dublin, Ireland, August 1993

[11] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing,* IEEE Data Engineering Bulletin 18(2), June 1995

[12] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases,* Singapore, December 1995

[13] R. Ramakrishnan, K.A. Ross, D. Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. in *Proceedings of the International Logic Programming Symposium,* pages 204-218, 1994

[14] J. Widom and S.Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, San Francisco, California, 1995

[15] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer,* 25(3):38-49, March 1992.

[16] G. Zhou, R. Hull, R. King, and J.-C. Franchitti. Data intergration and warehousing using H20. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing,* 18(2):29-40, June 1995.

[17] Y. Zhuge, H. Garcia-Molina, J.Hammer, and J.Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data,* pages 316-327, San Jose, California, May 1995.

[18] V. Poe. *Building a Data Warehouse for Decision Support.* Prentice Hall, 1996. Chapter 7: Designing the Database for a Data Warehouse.

[19] Stephen Peterson. *Stars: A Pattern Language for Query Optimized Schema.* Sequent Computer Systems, Inc., 1994

[20] Neil Raden. *Technology Tutorial: Modeling A DataWarehouse.* CMP Publications Inc., 1996

[21] *Star Schemas and STARjoin Technology.* Red Brick Systems

[22] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, J. Widom. *The TSIMMIS Approach to Mediation: Data Models and Languages (Extended Abstract)*, Dept. of CS, Stanford

[23] Y. Papakonstantinou, H. Garcia-Molina, J. Ullman. *MedMaker: A Mediation System Based on Declarative Specifications.* Dept. of CS, Stanford

[24] A. Rajaraman, Y. Sagiv, J. Ullman [1994], *Answering queries using templates with binding patterns.* PODS, Mai, 1995

[25] Chandra, A.K. and P.M. Merlin [1977]. *Optimal implementation of conjunctive queries in relational databases.* Proc. Ninth Annual ACM Symposium on the Theory of Computing, pp. 77-90

[26] Mauricio A. Hernandez, Salvatore J. Stolfo *The Merge/Purge Problem for Large Databases.* Departement of Computer Sience, Columbia University, New York, NY 10027

[27] Amit P. Sheth, Sunit K. Gala, Shamkant B. Navathe *On Automatic Reasoning for Schema Integration.* Bell Communication Research

[28] Y. Zhuge, H. Garcia-Molina, and J.L. Wiener. The strobe algorithms for multi-source warehouse consistency. Technical report, Stanford University, October 1995. Verfügbar über anonymous ftp von db.stanford.edu als pub/zhuge/1994/consistency-full.ps.

[29] Y. Zhuge, H. Garcia-Molina, J. Hammer, und J. Widom. View maintenance in a warehousing environment. Technical report, Stanford University, Oktober 1994. Verfügbar über anonymous ftp von db.stanford.edu als pub/zhuge/1994/anomaly-full.ps.

[30] H. Gupta, V. Harinarayan, A. Rajaraman und J. D. Ullman. Index Selection for OLAP. Technical Report 1135, Stanford University, Computer Science Department, 1996.

[31] A. Gupta, I. S. Mumick und K. A. Ross. Adapting Materialized Views after Redefinitions. In M. Carey und D. Schneider (Hrsg.), *Proceedings of the 1995 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, Seiten 211–222, San Jose, California, Mai 1995. ACM, ACM Press.

[32] Christoph Breitner und Uwe Herzog. Data–Warehouse als Schlüssel zur Bewältigung der Informationsflut. *Computerwoche (Sonderheft EXTRA)*, Seiten 16–19, Februar 1996.

[33] Richard Finkelstein. Understanding the Need for On–Line Analytical Servers. White–Paper, 1995.

[34] Kenan Technologies. An Introduction to Multidimensional Database Technology. White–Paper, 1995.

[35] Pilot Software. An Introduction to OLAP. White–Paper, 1995.

[36] R. Agrawal, A. Gupta und S. Sarawagi. Modeling Multidimensional Databases. Technischer Bericht, IBM Almaden Research Center, San Jose, CA, 1995.

[37] J. Gray, A. Bosworth, A. Layman und H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group–By, Cross–Tab, and Sub–Totals. In S.Y.W. Su (Hrsg.), *International Conference on Data Engineering (ICDE)*, Seiten 152–159, New Orleans, LA, Februar 1996.

[38] V. Harinarayan, A. Rajaraman und J.D. Ullmann. Implementing Data Cubes Efficiently. Technischer Bericht, Stanford University, 1995.

[39] S. Sarawagi, R. Agrawal und A. Gupta. On Computing the Data Cube. Technischer Bericht, IBM Almaden Research Center, San Jose, CA, 1995.

[40] S. Sarawagi und M. Stonebraker. Efficient Organization of Large Multidimensional Arrays.

[41] Maurice Houtsma, Arun Swami. *Set-oriented data mining in relational databases.* Data & Knowledge Engineering 17, S.245-262, 1995

[42] M. Holsheimer, M. Kersten, H. Mannila, H. Toivonen. *A perspective on databases and data mining*, CWI , Report CS R9531 1995, 1995

[43] Ramakrishnan Srikant, Rakesh Agrawal. *Mining Generalized Association Rules.* Proceedings of the 21st VLDB Conference Zurich, Switzerland, 1995

[44] Rakesh Agrawal, Tomasz Imielinski, Arun Swami. *Mining Association Rules between Sets of Items in Large Databases.* Proceedings of the ACM SIGMOD Conference on Management of Data, S.207-216, Washington, D.C., 1993

[45] Maurice Houtsma, Arun Swami. *Set-oriented mining of association rules.* International Conference on Data Engineering, Taipei, Taiwan, 1995

[46] Deregulation Powers Entergy's Award-Winning Data Warehouse. Commmentary by Boris Bosch, Entergy's Data Warehouse Manager. Entnommen aus dem World Wide Web unter http:www.tekptnr.com/tpi/tdwi/cases/entergy.htm

[47] Boosting Retail Sales Through Data Warehousing. Commentary by Jonathan Eales, IT Services Controller, and Dan Bernard, Systems Director, Woolworths. Entnommen aus dem World Wide Web unter http://www.tekptnr.com/tpi/tdwi/cases/woolwrth.htm

[48] Ingram Book Company Combines Departmental and Corporate Warehouses. Commentary by Doug Cothern, Ingram's Vice President of Applications. Entnommen aus dem World Wide Web unter http://www.tekptnr.com/tpi/tdwi/cases/ingram.htm

[49] Beyond The Data Warehouse: Reinventing the French National Railroad. Commentary by Cecile Queille, SNCF Director of Operations Research. Entnommen aus dem WWW unter http://www.tekptnr.com/tpi/tdwi/cases/sncf_rr.htm

[50] 3M Boosts Customer Satisfaction by Making Its Decentralized Enterprise Look Like One Entity to Its Analysts and Customers. Commentary by Dale Fayerweather, Lead Analyst, 3M Corporate Information Technology. Entnommen aus dem World Wide Web unter http://www.tekptnr.com/tpi/tdwi/cases/3m.htm

[51] How one small group with Comshare software and a can-do approach to development sparked an EIS movement at Allied Signal. ComShare Case Study. Entnommen aus dem World Wide Web unter http://www.comshare.com/customers/allied.htm

[52] Design and Implementation of a Data Warehouse, Dr. Amin Adatia, Vortrag gehalten am CASE Day at International Oracle Users Week, Philadelphia, 17.09.1995, angefordert beim Autor über EMail 70142,2503@compuserve.com