Indexpartitionierung in Oracle8i
Analyse und Bewertung unter Leistungsgesichtspunkten

— Studienarbeit —

Christian Kauhaus
ckauhaus@informatik.uni-jena.de

Betreuer

Jan Nowitzky
Prof. Dr. Klaus Küspert

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik
Institut für Informatik
Lehrstuhl für Datenbanken und Informationssysteme
Ernst-Abbe-Platz 1–4
07743 Jena

23. Januar 2001

Kurzfassung

Große Tabellen verlangen einen angepaßten physischen Datenbankentwurf. Neben dem Mittel der Tabellenpartitionierung, die sich bereits bewährt hat, ist die Indexpartitionierung dazugekommen. In dieser Arbeit soll untersucht werden, ob die Partitionierung von Indexen nennenswerte Leistungssteigerungen bei der Anfragebearbeitung ergibt. Dazu wird eine Messung auf einem Oracle8i DBMS, Version 8.1.5, durchgeführt. Die Meßdaten geben Hinweise auf Leistungssteigerungen in einigen Fällen. Es treten jedoch starke Leistungseinbußen in vielen anderen Fällen auf. Der Optimierer des verwendeten DBMS scheint darüberhinaus grundsätzliche Probleme im Umgang mit Indexpartitionierung zu haben.

Inhaltsverzeichnis

1 Einführung **5**
1.1 Problemstellung 5
1.2 Ziele der Arbeit 5

2 Grundlagen **7**
2.1 Indexarten 7
2.2 Partitionierung 9
2.3 Anfrageausführung 11
2.4 Möglichkeiten der Leistungssteigerung 13

3 Vorstellung eines Partitionierungskonzepts **15**
3.1 Indexart 15
3.2 Attributzahl 15
3.3 Partitionierungsart 15
3.4 Attributbeziehung 16

4 Implementierung in Oracle8i **18**
4.1 Umsetzung des Partionierungskonzepts 18
4.2 Mögliche Partitionierungsschemata 22

5 Ablauf der Messung **25**
5.1 Testdaten 25
5.2 Anfragen 28
5.3 Einfluß des Optimierers 30
5.4 Meßmethode 33

6 Meßergebnisse **36**
6.1 Tabelle: *Hash*, Index: *Hash* (HH) 37
6.2 Tabelle: *Hash*, Index: *Range* (HR) 44
6.3 Tabelle: *Range*, Index: *Range* (RR) 48

7 Fazit **58**
7.1 Zusammenfassung der Ergebnisse 58
7.2 Ausblick 60

A Tabellen **62**

Literatur **81**

Abbildungsverzeichnis

2.1 Knotenformate im B*-Baum 8
2.2 Beispiel eines Bitmap Indexes 9
3.1 Attributbeziehung als Matrix 17
4.1 Lokale Partitionierung 19
4.2 Globale Partitionierung 19
4.3 *Prefixed* Partitionierung 20
4.4 *Non-prefixed* Partitionierung 20
4.5 Mögliche Attributbeziehungen pro Partitionierungsart 23
5.1 Das Schema des TPC-H Benchmarks 25
5.2 Attribute der Tabelle `lineitem` 26
5.3 Punktanfrage 29
5.4 Bereichsanfrage 29
5.5 Verbundanfrage 29
5.6 Bereichsanfrage ohne *Hint* 31
5.7 Bereichsanfrage mit *Hint* 31
6.1 B*-Baum Index, 2 Attribute, HH Partitionierung, Punktanfrage . . . 37
6.2 B*-Baum Index, 2 Attribute, HH Partitionierung, Bereichsanfrage . . 38
6.3 B*-Baum Index, 2 Attribute, HH Partitionierung, Verbundanfrage . . 39
6.4 B*-Baum Index, 1 Attribut, HH Partitionierung, Punktanfrage 40
6.5 B*-Baum Index, 1 Attribut, HH Partitionierung, Bereichsanfrage . . 41
6.6 B*-Baum Index, 1 Attribut, HH Partitionierung, Verbundanfrage . . 42
6.7 Bitmap Index, 2 Attribute, HH Partitionierung, Punktanfrage 43
6.8 Bitmap Index, 2 Attribute, HH Partitionierung, Bereichsanfrage . . . 43
6.9 Bitmap Index, 2 Attribute, HH Partitionierung, Verbundanfrage . . . 44
6.10 B*-Baum Index, 2 Attribute, HR Partitionierung, Punktanfrage . . . 45
6.11 B*-Baum Index, 2 Attribute, HR Partitionierung, Bereichsanfrage . . 46
6.12 B*-Baum Index, 2 Attribute, HR Partitionierung, Verbundanfrage . . 47
6.13 B*-Baum Index, 1 Attribut, HR Partitionierung, Punktanfrage 48
6.14 B*-Baum Index, 1 Attribut, HR Partitionierung, Bereichsanfrage . . 49
6.15 B*-Baum Index, 1 Attribut, HR Partitionierung, Verbundanfrage . . 49
6.16 B*-Baum Index, 2 Attribute, RR Partitionierung, Punktanfrage . . . 50
6.17 B*-Baum Index, 2 Attribute, RR Partitionierung, Bereichsanfrage . . 51
6.18 B*-Baum Index, 2 Attribute, RR Partitionierung, Verbundanfrage . . 52
6.19 B*-Baum Index, 1 Attribut, RR Partitionierung, Punktanfrage 53
6.20 B*-Baum Index, 1 Attribut, RR Partitionierung, Bereichsanfrage . . 53
6.21 B*-Baum Index, 1 Attribut, RR Partitionierung, Verbundanfrage . . 54
6.22 Bitmap Index, 2 Attribute, RR Partitionierung, Punktanfrage 55
6.23 Bitmap Index, 2 Attribute, RR Partitionierung, Bereichsanfrage . . . 55
6.24 Bitmap Index, 2 Attribute, RR Partitionierung, Verbundanfrage . . . 56

Tabellenverzeichnis

4.1 Beispiel für lokale/globale und *prefixed*/*non-prefixed* Partitionierung 21
5.1 Partitionierungsattribute 27
A.1 B*-Baum Index, 2 Attribute, Partitionierungsart HH, Punktanfrage . 63
A.2 B*-Baum Index, 2 Attribute, Partitionierungsart HH, Bereichsanfrage 64
A.3 B*-Baum Index, 2 Attribute, Partitionierungsart HH, Verbundanfrage 65
A.4 B*-Baum Index, 1 Attribut, Partitionierungsart HH, Punktanfrage . . 66
A.5 B*-Baum Index, 1 Attribut, Partitionierungsart HH, Bereichsanfrage 66
A.6 B*-Baum Index, 1 Attribut, Partitionierungsart HH, Verbundanfrage 66
A.7 Bitmap Index, 2 Attribute, Partitionierungsart HH, Punktanfrage . . 67
A.8 Bitmap Index, 2 Attribute, Partitionierungsart HH, Bereichsanfrage . 68
A.9 Bitmap Index, 2 Attribute, Partitionierungsart HH, Verbundanfrage . 69
A.10 B*-Baum Index, 2 Attribute, Partitionierungsart HR, Punktanfrage . 70
A.11 B*-Baum Index, 2 Attribute, Partitionierungsart HR, Bereichsanfrage 71
A.12 B*-Baum Index, 2 Attribute, Partitionierungsart HR, Verbundanfrage 72
A.13 B*-Baum Index, 1 Attribut, Partitionierungsart HR, Punktanfrage . . 73
A.14 B*-Baum Index, 1 Attribut, Partitionierungsart HR, Bereichsanfrage 73
A.15 B*-Baum Index, 1 Attribut, Partitionierungsart HR, Verbundanfrage 73
A.16 B*-Baum Index, 2 Attribute, Partitionierungsart RR, Punktanfrage . 74
A.17 B*-Baum Index, 2 Attribute, Partitionierungsart RR, Bereichsanfrage 75
A.18 B*-Baum Index, 2 Attribute, Partitionierungsart RR, Verbundanfrage 76
A.19 B*-Baum Index, 1 Attribut, Partitionierungsart RR, Punktanfrage . . 77
A.20 B*-Baum Index, 1 Attribut, Partitionierungsart RR, Bereichsanfrage 77
A.21 B*-Baum Index, 1 Attribut, Partitionierungsart RR, Verbundanfrage 77
A.22 Bitmap Index, 2 Attribute, Partitionierungsart RR, Punktanfrage . . 78
A.23 Bitmap Index, 2 Attribute, Partitionierungsart RR, Bereichsanfrage . 79
A.24 Bitmap Index, 2 Attribute, Partitionierungsart RR, Verbundanfrage . 80

1 Einführung

1.1 Problemstellung

Die Datenmengen, die in relationalen Datenbankmanagementsystemen (DBMS) gehalten werden, sind in den letzten Jahren stetig gewachsen. Während DBMS mit einem Wachstum der Datenbestände über eine Vergrößerung der Tabellenzahl verhältnismäßig gut umgehen konnten, erwies sich der Umgang mit extrem großen Tabellen als neue Herausforderung. Viele Aspekte der Implementierung, z. B. Verteilung auf physische Speichermedien, Anfrageausführung oder Archivierung, konnten mit großen Tabellen nicht gut umgehen. Im Hinblick darauf, aber auch in Reaktion auf die allgemein gestiegenen Anforderungen an die Verfügbarkeit und Leistungsfähigkeit von DBMS, insbesondere im *Data Warehouse* Umfeld, sind die Datenbankhersteller gezwungen worden, einige Konzepte in ihre Produkte einfließen zu lassen.

Von den verfügbaren Techniken zum Umgang mit großen Tabellen kommt vor allem einem angepaßten *physischen Entwurf* [NM00] eine große Bedeutung zu. Um mit sehr großen Tabellen umgehen zu können, bietet sich die *Tabellenpartitionierung*, also die Zerlegung in Teiltabellen, an. Dieses Verfahren ist schon seit einiger Zeit im praktischen Einsatz. Aber eine Zerlegung der Tabellen ist nur ein erster Schritt. Mit der Tabellenpartitionierung wurde in der Regel auch die Partitionierung von Zugriffspfaden (Indexpartitionierung) möglich, die nun zunehmend ins Blickfeld der Anwender gerät.

Indexpartitionierung ist die logische Fortführung des Tabellenpartitionierungsgedankens. Als Vorteile für Indexpartitionierung werden Verbesserungen bei der Administration, der Wartbarkeit und der Leistung genannt. Allerdings stellt sich die Frage, ob diese genannten Vorteile in der Praxis wirklich auftreten und einer kritischen Untersuchung standhalten. Im Rahmen dieser Arbeit soll der Frage der Leistungsverbesserung nachgegangen werden. Dabei baut diese Arbeit sehr stark auf [Bau00] auf, worin theoretische Potentiale der Leistungsverbesserung bei Indexpartitionierung gefunden worden sind. Diese theoretischen Möglichkeiten sollen anhand einer praktischen Messung in einem DBMS bestätigt werden. Die anderen beiden Vorteile, Administration und Wartbarkeit, sollen anderen Untersuchungen vorbehalten bleiben.

1.2 Ziele der Arbeit

Diese Arbeit hat den Zweck, die Leistungsverbesserung durch den Einsatz von Indexpartitionierung zu überprüfen und zu quantifizieren. Dazu werden Messungen auf einem Oracle8i Datenbanksystem Version 8.1.5 durchgeführt. Es soll explizit um das praktische Verhalten dieses DBMS gehen, wobei klar ist, daß die Ergebnisse damit stark von der verwendeten Soft- und Hardware abhängig sind. Weiterhin ist klar, daß aufgrund der Tatsache der Messung eine große Anzahl weiterer Unsicherheitsfaktoren dazukommt. Diese Problematik ist dieselbe bei allen *Benchmarks* und läßt sich kaum umgehen. Dennoch sollen mit dieser Arbeit die vorliegenden theoretischen Erkenntnisse um praktische Meßwerte, die exemplarisch an einem System aufgenommen worden

sind, ergänzt werden. Natürlich ist die Übertragbarkeit der hier gewonnenen Werte auch nur auf ein Oracle8i-System Version 8.1.6 in Frage zu stellen. Dennoch soll diese Arbeit dazu dienen, Schwachstellen und interessante Punkte der Indexpartitionierung unter Leistungsgesichtspunkten herauszufinden, die dann mit anderen Systemen und Konfigurationen verglichen werden können.

Um die zu untersuchende Stoffmenge etwas einzugrenzen, wurde nur die Leistungsverbesserung für Anfragen betrachtet. Partitionierung und speziell Indexpartitionierung wird auch eine Auswirkung auf das Verhalten des DBMS beim Einfügen, Löschen oder Ändern haben. Weiterhin liegen die Leistungsengpässe bei vielen Anwendungen mit großen Tabellen, insbesondere bei *Data Warehouses*, vor allem bei Anfragen, da diese großen Datenbestände schnell auf bestimmte Zusammenhänge hin untersucht werden sollen. Änderungen sind seltener und weniger kritisch.

Trotz dieser Einschränkung auf reine Anfragen bleiben beim Einsatz von Tabellen- und Indexpartitionierung noch viele mögliche Kombinationen. Oracle8i unterstützt sowohl B*-Baum als auch Bitmap-Indexe, die beide jeweils ein- oder mehrdimensional definiert werden können. Die Partitionierung kann nach verschiedenen Partitionierungsfunktionen vorgenommen werden und die Kriterien, nach denen partitioniert wird, können unterschiedlich sein. Weiterhin kann die Partitionierung des Indexes in verschiedenen Verhältnissen zur Partitionierung der Tabelle stehen. Ziel dieser Arbeit ist es, in einem breiten Meßfeld möglichst viele Kombinationen abzudecken. Es soll nicht aufgrund von theoretischen Annahmen ein Großteil der Möglichkeiten außen vor gelassen und nur eine kleine Reihe sehr spezieller Anfragen und Partitionierungen angesehen werden. Stattdessen sollen Zusammenhänge zwischen den Kombinationen aufgezeigt und nach Möglichkeit Einzeleffekte bestimmter Parameter isoliert werden. In der Analyse der Einzeleffekte soll sich zeigen, an welchen Stellen die „Knackpunkte“ liegen, deren vergleichende Untersuchung auf anderen Systemen lohnt.

Diese Arbeit ist wie folgt gegliedert. Nach einer kurzen Einführung in Indexe und deren Partitionierung in Abschnitt 2 sollen die verschiedenen Kombinationen in einem Partitionierungskonzept gegliedert und systematisiert werden (Abschnitt 3). Anschließend soll in Abschnitt 4 die konkrete Umsetzung dieses Konzepts in Oracle8i untersucht werden. Nach der Vorstellung der Meßmethoden in Abschnitt 5 werden dann in Abschnitt 6 die Meßergebnisse präsentiert. Abschnitt 7 schließt die Arbeit mit einem Fazit ab.

2 Grundlagen

In diesem Abschnitt sollen die zum Verständnis der Indexpartitionierung notwendigen Grundlagen kurz dargestellt werden. Neben einer Einführung in die Themen Indexarten und Partitionierung soll außerdem das grundlegende Modell der Ausführung einer `select`-Anfrage im Rahmen partitionierter Tabellen und Indexe beleuchtet werden.

2.1 Indexarten

Indexe, oder Zugriffspfade, dienen dem schnellen Auffinden von Tupeln einer Relation nach inhaltlichen Kriterien. Als Hilfsstrukturen dienen sie damit nicht dem direkten Speichern von Daten und sollten somit ein ausgewogenes Verhältnis zwischen Leistungsgewinn beim Auffinden von Tupeln und Mehraufwand beim Einfügen, Ändern und Löschen bieten. Daraus folgt, daß die Organisationsform des Indexes der jeweils zugrundeliegenden Relation mit ihren spezifischen Attributeigenschaften angepaßt sein sollte.

Aus der Vielzahl möglicher Unterscheidungsmerkmale [HS97] sollen einige kurz vorgestellt werden, die bei der Einordnung der folgenden Indexarten hilfreich sind:

Primär-/Sekundärschlüssel Der Begriff Schlüssel soll nicht im engen Sinne des relationalen Modells gebraucht werden, sondern für jede Attributkombination stehen, die die zu selektierenden Tupel beschreibt. Ein Primärschlüssel ist jede minimale Attributkombination, die ein Tupel eindeutig identifiziert. Alle anderen Attributkombinationen sollen im folgenden Sekundärschlüssel genannt werden, wenn sie zur Selektion von Tupeln dienen.

Primär-/Sekundärindex Ein Primärindex nutzt die Dateiorganisation der Tabelle bzw. ist direkt mit der Tabelle in eine Datenstruktur integriert. Obwohl mehrere Primärschlüssel pro Tabelle denkbar sind, kann ein Primärindex nur einen davon unterstützen, nämlich den, der die phyische Sortierreihenfolge der Tabelle bestimmt. Alle anderen Indexarten, die die physische Organisation der Tabelle nicht ausnutzen, sondern nur über TIDs (*tupel identifier*) auf sie zugreifen, sollen Sekundärindexe genannt werden. Im Rahmen dieser Arbeit werden nur Sekundärindexe untersucht.

Ein-Attribut-/Mehr-Attribut-Index Ein Ein-Attribut-Index stellt einen Zugriffspfad über ein einziges Attribut bereit. Dagegen bietet der Mehr-Attribut-Index Unterstützung für den Zugriff über mehrere Attribute. Er kann dennoch als eindimensionaler Index implementiert sein, wenn die mehreren Attribute vor dem Indexzugriff zu einem Schlüsselwert konkateniert werden.

In der Literatur sind eine Fülle von verschiedenen Indexarten vorgeschlagen worden, von denen wenige in der Praxis in größerem Umfang eingesetzt werden. In dieser Arbeit sollen zwei davon, der B*-Baum Index und der Bitmap Index, untersucht werden.

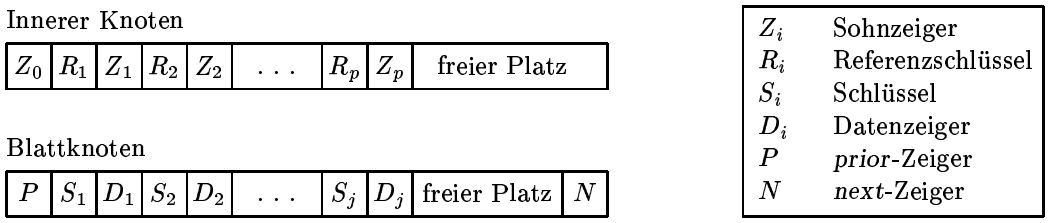

Abbildung 2.1: Knotenformate im B*-Baum mit $p \in [k+1 \ldots 2k+1]$ und $j \in [k^* \ldots 2k^*]$ (aus [HR99])

B*-Baum Index Ein B*-Baum [Wed74] ist ein mehrstufiger, hohler Suchbaum, der durch die Parameter (k, k^*, h) charakterisiert wird. Jeder Weg von der Wurzel zum Blatt hat die Länge h, d. h. der Baum ist vollständig balanciert. Die inneren Knoten, die nicht Wurzel oder Blatt sind, haben mindestens $k+1$ und höchstens $2k+1$ Söhne. Die inneren Knoten, die in der Regel die Größe einer Datenbankseite haben, bestehen aus einer abwechselnden Folge von Referenzschlüsseln und Verweisen auf Söhne, deren Schlüssel durch die Referenzschlüssel eingegrenzt werden. Die Blätter dagegen bestehen aus k^* bis $2k^*$ Paaren von Schlüsselwert und Verweis auf das zugehörige Tupel. Außerdem hat jedes Blatt am Anfang und Ende jeweils einen Zeiger auf das entsprechende Nachbarblatt, so daß sequentielles Durchlaufen der Blätter einfach möglich ist. Die Knotenformate sind in Abbildung 2.1 grafisch dargestellt.

Aufgrund des hohen Verzweigungsgrades pro Knoten (in der Praxis sind Werte von 500 oder mehr üblich) bleibt die Baumhöhe auch bei sehr vielen zu indizierenden Tupeln niedrig. Da in der Regel eine Seite pro Knoten benutzt wird, bleibt somit auch die Anzahl der Seitenzugriffe, die zum Auffinden eines Schlüssels benötigt werden, gering.

B*-Bäume sind eine weit verbreitete und in vielen Fällen effiziente Indexart. Bei Mehr-Attribut-Indexen erweisen sie sich jedoch als ungünstig, da in diesem Fall das am weitesten links stehende Attribut die Position im Baum dominiert. Der Index bietet dann für Anfragen, deren Prädikate nicht mindestens einen linken Präfix der Attribute mit Werten belegen, keine Unterstützung. Für diesen Fall sind zahlreiche Varianten des B-Baums (KDB-Baum [Rob81], hB-Baum [LS90], UB-Baum [Bay96] u. a.), vorgeschlagen und teilweise schon implementiert worden, auf die hier aber nicht näher eingegangen werden soll. Ein damit verwandtes Problem besteht darin, daß die in einer Anfrage enthaltenen *UND*- oder *ODER*-Verknüpfungen nicht auf Indexoperationen abgebildet werden können. Stattdessen müssen die Ergebnismengen der einzelnen Anfrageprädikate über die B*-Bäume ermittelt und anschließend im Hauptspeicher über Mengenoperationen verknüpft werden. Einige der B-Baum-Varianten decken einfachere Verknüpfungen ab, für den allgemeinen Fall ist jedoch keine Lösung bekannt.

Bitmap Index Diese Indexart, die die Tupel über Bitlisten verwaltet, ist im Umfeld von *Data Warehouse* Anwendungen verbreitet [HR99, SH99]. Für jeden Wert aus dem Wertebereich des zu indizierenden Attributs wird eine Liste angelegt, auf der

TID	Kaufteil	Fertigung	Normteil	Baugruppe	Endprod.
r_1	0	0	0	0	1
r_2	1	0	0	0	0
r_3	0	1	0	0	0
r_4	0	1	0	0	0
r_5	0	0	0	0	1
r_6	1	0	0	0	0
r_7	0	0	1	0	0
⋮	⋮	⋮	⋮	⋮	⋮

Abbildung 2.2: Beispiel eines Bitmap Indexes für das Attribut `typ` (aus [SH99])

für jedes Tupel vermerkt ist, ob das Attribut diesen Wert annimmt oder nicht (siehe Abbildung 2.2).

Der Bitmap Index ermöglicht es, Attribute mit schlechter Selektivität platzsparend zu indizieren, da er für jedes Tupel eines Attributs A nur $|\mathrm{dom}(A)|$ Bits benötigt. In [HR99] wird der Einsatz von Bitmap Indexen ab 10 % Selektivität pro Attributwert empfohlen. Je besser die Selektivität ist, um so mehr Platz verbraucht der Bitmap Index, weshalb er für Primärschlüssel schlecht geeignet ist. Allerdings kann der Platzbedarf reduziert werden, wenn statt der Attributwerte eine Kodierung mit „künstlich“ schlechterer Selektivität benutzt wird, z. B. ein Symbol für einen ganzen Wertebereich [WB98]. Diese Vorgehensweise ist natürlich für Primärschlüssel ebenso ungeeignet, kann aber die effizienten Verknüpfungsmöglichkeiten von Bitmap Indexen für Attribute mit guter Selektivität ausnutzen.

Logische Verknüpfungen lassen sich effizient realisieren, indem die entsprechenden Mengenoperationen direkt auf Indexebene mit den Bitlisten durchgeführt werden. Damit lassen sich auch Mehr-Attribut-Indexe aufbauen, indem pro Attribut ein Bitmap Index angelegt wird. Belegt das Selektionsprädikat mehrere Attribute mit Werten, können die entsprechenden Bitlisten verknüpft werden. Dabei spielt die Reihenfolge der Attribute im Gegensatz zu B*-Bäumen keine Rolle mehr.

2.2 Partitionierung

Bei der Partitionierung von Datenbankobjekten (Tabellen oder Indexen) handelt es sich um eine vollständige und disjunkte Zerlegung in Fragmente [NM00]. Dabei bedeutet vollständig, daß jedes Tupel einer Partition zugeordnet werden kann und disjunkt, daß sich die Partitionen nicht überlappen. Bei der Partitionierung unterscheidet man die Grundvarianten *horizontal* und *vertikal*. Horizontale Partitionierung bedeutet, daß die Tupel selbst nicht fragmentiert werden und die Partitionen aus Gruppen von Tupeln bestehen, die nach bestimmten Kriterien zusammengestellt sind. Bei der vertikalen Partitionierung werden hingegen die Tupel zerlegt und Gruppen von Spalten in den Partitionen gespeichert.

Die Zuordnung von Tupeln zu Partitionen geschieht mit Hilfe einer *Partitionierungsfunktion*. Diese Funktion bildet jedes Tupel auf eine bestimmte Partition ab.

Dabei sind grundsätzlich zwei Arten von Partitionierungsfunktionen möglich: *wertebasierte* und *nicht wertebasierte* Partitionierungsfunktionen. Bei nicht wertebasierten Funktionen wird die Partition eines Tupels mit Hilfe eines Fremdkriteriums festgelegt, z. B. einer Zufallszahl oder der Reihenfolge des Auftretens eines Tupels. Nicht wertebasierte Funktionen haben den Nachteil, daß die Zuordnung eines Tupels zu einer Partition in der Regel weder wiederholbar noch im Nachhinein nachvollziehbar ist [Bau00]. Daher werden in der Praxis größtenteils wertebasierte Partitionierungsfunktionen verwandt. Diese berechnen die Partition aus einem oder mehreren Attributwerten des Tupels. Das hat den Vorteil, daß die Zuordnung auch im Nachhinein nachvollziehbar bleibt. Oft ist es günstig, die Partitionierungsfunktion von Attributen abhängig zu machen, die auch in den Anfrageprädikaten verwendet werden. Damit kann das DBMS durch Anwendung der Partitionierungsfunktion sofort die Partition bestimmen, in der das gesuchte Tupel liegen muß.

Übliche wertebasierte Partitionierungsfunktionen sind (siehe auch 4.1.5):

Bereich Der Wertebereich eines Attributs wird in zusammenhängende Unterbereiche eingeteilt, jeder Unterbereich wird einer Partition zugeordnet.

Hash Horizontale Partitionierung kann als Spezialfall gängiger *Hashing*-Verfahren angesehen werden. Fast jede in der Literatur vorgestellte *Hash*-Funktion kann auch als Partitionierungsfunktion verwendet werden.[1]

Abgeleitet Als Ausgangswert wird der Wert eines Attributs einer anderen Tabelle benutzt, deren Tupel mit den Tupeln der zu partitionierenden Tabelle irgendwie in Zusammenhang gebracht werden können (z. B. Fremdschlüsselbeziehung). Dieser Ausgangswert dient als Eingabe einer weiteren Partitionierungsfunktion. Damit ist es möglich, mehrere Tabellen abhängig von den Werten einer Tabelle zu partitionieren [MN00].

Realisiert wird die horizontale Partitionierung von Tabellen einfach dadurch, daß eine gegebene Tabelle anhand der Partitionierungsfunktion in kleinere Einheiten aufgeteilt wird. Besondere Vorkehrungen zum späteren Auffinden der Tupel sind bei wertebasierten Partitionierungsfunktionen nicht nötig, da die Partition, in der sich ein Tupel befindet, bei bekanntem Wert der Partitionierungsattribute durch Anwenden der Partitionierungsfunktion herausgefunden werden kann. Die Partitionierung von Indexen weicht jedoch davon ab, da ein Index eine komplexe Struktur ist und nicht ohne weiteres in kleinere Einheiten zerlegt werden kann. Man baut stattdessen pro Partition einen separaten kleinen Teilzugriffspfad auf, der alle Tupel, die diese Partition umfaßt, referenziert. Dabei ist zu beachten, daß die Indexpartitionen durchaus unterschiedlich von den Tabellenpartitionen sein können. Ob und wie sich die Teilindexe dabei zu einem Gesamtindex zusammensetzen lassen, ist erst einmal unerheblich und bleibt Gegenstand einer gesonderten Betrachtung (siehe 4.1.3).

1. Eine große Anzahl von *Hash*-Algorithmen sind in Algorithmen-Lehrbüchern wie [CLR92, Sed92] zu finden.

2.3 Anfrageausführung

Anfragen gegen partitionierte Tabellen können auf verschiedene Weisen ausgeführt werden. Eine `select`-Anfrage kann neben den drei bekannten Ausführungsstrategien für nichtpartitionierte Tabellen und Indexe (Indexzugriff, *Index Scan*, *Table Scan* [Mit95, SH99]) noch über zwei weitere Weisen im partitionierten Fall ausgeführt werden. Da im Rahmen dieser Arbeit nur `select`-Anfragen benutzt werden, sollen Ausführungsstrategien für weitere DML-Operationen hier nicht betrachtet werden.

Die Ausführungsstrategien für `select`-Anweisungen im partitionierten Fall sind, geordnet nach Kosten:

2.3.1 Direkter Indexzugriff

Falls das Selektionsprädikat ein „=“ enthält oder eine Bereichsselektion mit wenigen diskreten Werten ausgeführt wird, ist es am günstigsten, die benötigten TIDs direkt per Indexzugriff zu ermitteln, zu sortieren und die Tupel einzulesen. In der Regel lagert das DBMS dabei die gelesenen Indexseiten in den Puffer ein, so daß die betroffenen Indexseiten nur einmal gelesen werden müssen und danach im Hauptspeicher schnell verfügbar sind. Pro selektiertes Tupel ist in der Regel eine weitere Leseoperation notwendig.

Dieses Verfahren spielt seine Stärken bei Anfragen mit guter Selektivität, insbesondere Punktanfragen, aus. Dabei ist es fast unerheblich, ob die Tabelle oder der Index partitioniert ist, da bei wertebasierter Partitionierung die betroffene Partition sofort ermittelt werden kann, sofern die dazu notwendigen Attribute im Selektionsprädikat vorkommen. Unter Umständen allerdings kann eine Tabellenpartitionierung ausgenutzt werden, wenn mehrere Werte angefragt sind, deren Tupel sich in verschiedenen Partitionen befinden. In diesem Fall können die Tupel parallel geholt werden, sofern die Hardware das erlaubt.

Falls die Selektivität der Anfrage jedoch schlechter ist, sind beim direkten Indexzugriff zu viele Kopfbewegungen des Plattenlaufwerks nötig. Diese kosten mehr Zeit, als gebraucht würde, den Index oder die Tabelle sequentiell einzulesen (s. u.). Dabei muß beachtet werden, daß eine Partitionierung der Tabelle mit steigender Partitionszahl die Selektivität pro Partition verschlechtert. So gesehen wäre es vorteilhaft, wenn das DBMS die Ausführungsstrategie pro Partition individuell entscheidet und so den Indexzugriff nur auf Partitionen anwendet, in denen die partitionsbezogene Selektivität gut ist. Allerdings würde dabei auch der Optimierungsaufwand proportional zur Anzahl der vorhandenen Partitionen steigen, was unter Umständen zu viel Zeit in Anspruch nimmt.

2.3.2 *Index Partition Scan*

Insbesondere für Bereichsanfragen ist es sinnvoll, den Index in Sortierreihenfolge zu durchlaufen. Durch Partitionierung kann es weiterhin die Möglichkeit geben, diesen *index scan* auf nur eine oder wenige Partitionen zu beschränken.

Ist das Attribut oder die Attributkombination, nach der der Index sortiert ist, gleichzeitig Partitionierungsattribut, kann kaum eine Verbesserung erzielt werden, da aufgrund der Sortierreihenfolge eines Indexes das erste und letzte Tupel eines Bereichs auch ohne Partitionsausschluß einfach bestimmt werden kann. Eine Verbesserung ist jedoch möglich, wenn Indexschlüssel (nach dem der Index sortiert ist) und Partitionierungsattribut unterschiedlich sind, jedoch beide durch Anfrageprädikate eingeschränkt werden. In diesem Fall kann eine Leistungsverbesserung dadurch erzielt werden, daß zum einen ein Partitionsausschluß stattfinden kann und zum anderen die Sortierreihenfolge des Indexes ausgenutzt wird.

2.3.3 *Index Full Scan*

Dieser entspricht dem klassischen *index scan* im nicht partitionierten Fall. Hierbei werden natürlich alle Indexpartitionen sequentiell gelesen und die TIDs in Sortierreihenfolge verarbeitet. Neben dem Vorteil, daß die TIDs bereits sortiert vorliegen, gibt es den Nachteil, daß bei nicht-geclusterten Indexen diese Sortierreihenfolge nicht der physischen Reihenfolge der Tupel auf der Platte entspricht und es durch die daraus resultierende Kopfbewegung unter Umständen länger dauern kann, die Tupel sortiert von der Platte zu holen, anstatt die Tabelle direkt sequentiell einzulesen.

Falls die Tupel aber komplett im Puffer liegen, d. h. die Anfrage aus dem Hauptspeicher beantwortet werden kann, ist dieses Verfahren besonders schnell. Durch die Ausgabe der Tupel in Sortierreihenfolge kann unter Umständen auf eine nachfolgende teure Sortieroperation verzichtet werden.

2.3.4 *Table Partition Scan*

Kann aufgrund des Selektionsprädikates ein Partitionsausschluß durchgeführt werden und es ist kein passender Index da oder die Selektivität der Anfrage bezüglich der betrachteten Partition ist schlecht, bietet es sich an, eine oder mehrere Partitionen sequentiell zu lesen und die erhaltenen Tupel anhand des Selektionsprädikates zu filtern. Diese Methode gehört zu den Hauptverbesserungen durch Tabellenpartitionierung, da die Indexnutzung aus genannten Gründen nicht immer sinnvoll ist und auf diese Weise dennoch die zu betrachtende Datenmenge signifikant verkleinert wird, insbesondere bei vielen Partitionen. Die Leistung kann weiterhin verbessert werden, wenn zusätzlich Techniken wie *Multi-Block I/O* oder *Read Ahead* angewandt werden.

Eine Partitionierung, insbesondere eine mit vielen Partitionen, kann daher als eine Art Indexersatz über den Partitionierungsattributen angesehen werden. Natürlich kann dieser Indexersatz keine einzelnen TIDs benennen, man kann ihn stattdessen als eine Art Bereichsindex auffassen, der zu jeder Wertebelegung der Partitionierungsattribute im Selektionsprädikat einen Bereich (=Partition) benennen kann, in dem die gewünschten Tupel zu finden sind. Wenn die Partitionsgröße hinreichend klein und der *table partition scan* hinreichend schnell ist, kann das eine interessante Alternative zu herkömmlichen Indexstrukturen sein.

2.3.5 *Table Full Scan*

Als letzte und teuerste Möglichkeit bietet sich das Äquivalent zum klassischen *table scan* an, bei dem alle Partitionen einer Tabelle sequentiell nach den gewünschten Tupeln durchsucht werden. Diese Ausführungsstrategie ist immer möglich und mit Abstand am teuersten.

2.4 Möglichkeiten der Leistungssteigerung

Im vorausgegangen Abschnitt wurde bereits deutlich, daß die potentiellen Leistungssteigerungen bei der Tabellenpartitionierung größer sind als bei einer Indexpartitionierung, da die Indexe oftmals im Puffer gehalten werden und von Tabellenpartitionierung ungleich größere Datenmengen profitieren. Daher liegen die Hauptgründe für den Einsatz von Indexpartitionierung im administrativen Bereich. So ist es z. B. möglich, den Tabellen- sowie zugehörigen Indexpartitionen gezielt getrennte physische Speicherbereiche zuzuordnen. Dennoch bietet die Indexpartitionierung prinzipiell drei Möglichkeiten der Leistungssteigerung, die im Einzelnen vorgestellt werden sollen.

2.4.1 Partitionsausschluß

Der in Abschnitt 2.3.2 erwähnte Partitionsausschluß führt zwar theoretisch zu Leistungsverbesserungen, bewirkt in der Praxis aber nur bei sehr großen Tabellen mit sehr großen Indexen etwas. Sind die Indexe klein genug, daß sie im Puffer gehalten werden können, sind die Kosten für den Indexzugriff in jedem Fall klein gegenüber den Kosten für den Tabellenzugriff. Erst bei Indexen, die nicht mehr komplett in den Puffer passen, ermöglicht der Partitionsausschluß, nur die relevanten Indexseiten im Puffer zu halten und damit die Indexzugriffskosten gering zu halten.

2.4.2 Verkleinerung der Indexstrukturen

Aufgrund der verringerten zu indizierenden Datenmenge kann die Datenstruktur in den einzelnen Indexpartitionen verkleinert werden. Bei B*-Baum Indexen hat das aufgrund des in der Regel hohen Grades des Baums kaum Auswirkungen, denn es werden nur wenige Ebenen eingespart, wenn überhaupt. Bei Bitmap Indexen kann die Bitliste entsprechend der Datenreduzierung verkleinert werden, was abhängig vom eingesetzten Kompressionsverfahren (wenn der Bitmap Index überhaupt komprimiert wird) eine signifikante Beschleunigung der Verarbeitung einer Indexpartition ergeben kann.

In beiden Fällen ist die Leistungssteigerung jedoch gering und kommt nur bei großen Datenmengen, vielen Partitionen und in Verbindung mit Partitionsausschluß zum Tragen.

2.4.3 *Intra-Query*-Parallelität

Bei partitionierten Tabellen und Indexen ist es möglich, daß mehrere Threads oder Tasks des DBMS Teile der Anfrage parallel bearbeiten. Falls dabei parallel Indexseiten gelesen werden müssen, erzeugt das unter Umständen ein nicht optimales Zugriffsmuster für das Plattenlaufwerk, auf dem der Index gespeichert ist, da der Kopf zwischen verschiedenen Positionen hin- und herspringen muß, was sehr zeitaufwendig ist. Ist dagegen der Index partitioniert und die Partitionen auf verschiedene physische Plattenlaufwerke verteilt, kann die Anfrage ohne Behinderungen parallel bearbeitet werden.

Diese Überlegungen gelten natürlich nur, wenn nicht die relevanten Indexseiten sowieso im Puffer liegen (2.4.1) und zur Anfragebearbeitung nur Tabellenseiten von der Platte gelesen werden müssen.

3 Vorstellung eines Partitionierungskonzepts

Um eine Beurteilung der Auswirkung von Einzeleffekten bei der Indexpartitionierung machen zu können, müssen die zahlreichen konzeptionellen und implementierungsbedingten Optionen, von denen ein Teil im vorangegangenen Abschnitt vorgestellt wurde, sinnvoll schematisiert und klassifiziert werden. Diese Klassifikation, die im folgenden eingeführt werden soll, definiert eine Menge von Partitionierungsschemata, die, sofern möglich und sinnvoll (siehe 4.2), im Datenbanksystem erzeugt werden. Die Antwortzeitmessung der Anfragen über alle diese Kombinationen ermöglicht es, Einzeleffekte herauszusuchen und zu analysieren.

Die in dieser Arbeit verwendete Klassifikation besteht aus vier Elementen: Indexart, Attributzahl, Partitionierungsart, Attributbeziehung. Diese sollen in den folgenden Abschnitten einzeln vorgestellt werden.

3.1 Indexart

Oracle8i unterstützt die beiden grundlegenden Indexarten B*-Baum und Bitmap Index [LO99]. Da beide partitioniert werden können, Bitmap Indexe allerdings nur eingeschränkt, gibt es an dieser Stelle zwei zu betrachtende Möglichkeiten.

3.2 Attributzahl

Indexschlüssel können entweder aus einem oder mehreren Attributen bestehen. Bei der Implementierung von B*-Baum Indexen werden mehrere Schlüsselattribute zu einem Schlüssel konkateniert. Das bedingt, daß mehrdimensionale Punktanfragen nur dann sinnvoll mittels eines B*-Baum Indexes ausgewertet werden können, wenn alle Attribute von links durchgehend mit Werten belegt sind, d. h. wenn ein linker Präfix des Schlüssels belegt ist [SH99, Bay96]. Mehrdimensionale Bereichsanfragen werden von B*-Baum Indexen gar nicht unterstützt. Aus diesem Grund sollen für B*-Baum Indexe zwei Varianten unterschieden werden: Indexe mit eindimensionalen Schlüsseln und Indexe mit mehrdimensionalen Schlüsseln. Der zweite Fall kann dahingehend vereinfacht werden, daß Indexe mit Schlüsseln aus zwei Attributen benutzt werden. Indexschlüssel mit mehr als zwei Attributen bringen qualitativ nichts Neues.

Diese Unterscheidung ist für Bitmap Indexe nicht notwendig, da mehrdimensionale Schlüssel gut unterstützt werden, indem man für jedes Schlüsselattribut einen eigenen Bitmap Index anlegt. Das DBMS kann dann bei Anfragen die Informationen der einzelnen Indexe über Boolesche Operationen effizient verknüpfen.

3.3 Partitionierungsart

Wie in Abschnitt 4.1.5 auf Seite 21 noch genauer erläutert wird, sollen von den in Oracle8i möglichen Partitionierungsarten die *Range*- und *Hash*-Partitionierung näher untersucht werden. Da sowohl Tabelle als auch Index partitioniert sein können, gibt es demnach vier Kombinationen. Die Partitionierungsarten sollen abgekürzt als

$\{TI : T, I \in \{R, H\}\}$ bezeichnet werden, wobei T für die Partitionierung der Tabelle und I für die Partitionierung des Indexes steht. Demnach heißt z. B. die Kombination HR, daß die Tabelle *hash*- und der zugehörige Index *range*-partitioniert werden soll.

3.4 Attributbeziehung

Drei Attributkombinationen spielen bei der Tabellen- und Indexpartitionierung eine Rolle: Indexschlüssel IS, Index-Partitionierungsattribute P_I und Tabellen-Partitionierungsattribute P_T, wovon IS und P_I sowie P_I und P_T jeweils in einer bestimmten Beziehung zueinander stehen.

Der Indexschlüssel IS sollte in allen Schemata möglichst gleich bleiben, damit in jedem Fall dieselben Anfragen gestellt werden können und die Ergebnisse vergleichbar bleiben. Die Partitionierungsattribute des Indexes P_I und der Tabelle P_T beschreiben, welche Attribute als Eingabe für die jeweilige Partitionierungsfunktion dienen sollen.

Im allgemeinen Fall, d. h. wenn IS, P_I und P_T geordnete Mengen mehrerer Attribute darstellen, können die Attributmengen A und B in folgenden Beziehungen zueinander stehen [Bau00]:

1. $A \cap B = \emptyset$. Die Attributmengen weisen keine Gemeinsamkeiten auf.
2. $A = B$. Die Attributmengen sind gleich.
3. $A \subset B$ (Präfix). Die Attributmenge A ist echte Teilmenge von B, wobei A ein linker Präfix von B ist.
4. $A \subset B$ ($\neg$Präfix). Die Attributmenge A ist echte Teilmenge von B, wobei A kein linker Präfix von B ist.
5. $A \supset B$ (Präfix). Die Attributmenge A ist echte Obermenge von B, wobei A ein linker Präfix von B ist.
6. $A \supset B$ ($\neg$Präfix). Die Attributmenge A ist echte Obermenge von B, wobei A kein linker Präfix von B ist.
7. $A \cap B \neq \emptyset$ (Präfix). Die Attributmengen überlappen einander, ohne von einem der vorangehenden Fälle erfaßt worden zu sein, wobei A ein linker Präfix von B ist.
8. $A \cap B \neq \emptyset$ ($\neg$Präfix). Die Attributmengen überlappen einander, ohne von einem der vorangehenden Fälle erfaßt worden zu sein, wobei A kein linker Präfix von B ist.

Im speziellen Fall eindimensionaler Indexschlüssel und Partitionierungsattribute kommen natürlich nur die Fälle 1 und 2 in Betracht.

Diese acht Beziehungen können sowohl zwischen Indexschlüssel und Index-Partitionierungsattributen als auch zwischen Index- und Tabellen-Partitionierungsattributen auftreten. Analog zu den Partitionierungsarten in Abschnitt 3.3 werden die

	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (P)	$P_I \subset IS$ ($\neg$P)	$P_I \supset IS$ (P)	$P_I \supset IS$ ($\neg$P)	$P_I \cap IS \neq \emptyset$ (P)	$P_I \cap IS \neq \emptyset$ ($\neg$P)
$P_T \cap P_I = \emptyset$	11	12	13	14	15	16	17	18
$P_T = P_I$	21	22	23	24	25	26	27	28
$P_T \subset P_I$ (P)	31	32	33	34	35	36	37	38
$P_T \subset P_I$ ($\neg$P)	41	42	43	44	45	46	47	48
$P_T \supset P_I$ (P)	51	52	53	54	55	56	57	58
$P_T \supset P_I$ ($\neg$P)	61	62	63	64	65	66	67	68
$P_T \cap P_I \neq \emptyset$ (P)	71	72	73	74	75	76	77	78
$P_T \cap P_I \neq \emptyset$ ($\neg$P)	81	82	83	84	85	86	87	88

Abbildung 3.1: Attributbeziehung als Matrix

Attributbeziehungen mit $\{ti : t, i \in \{1, \ldots, 8\}\}$ bezeichnet, wobei t für die Beziehung zwischen Tabellen- und Index-Partitionierungsattribut und i für die Beziehung zwischen Index-Partitionierungsattributen und Indexschlüssel steht. Somit lassen sich die Attributbeziehungen als Matrix darstellen, deren Zeilen und Spalten von 1 bis 8 numeriert sind. Die Zeile wird durch die erste Ziffer t bestimmt und die Spalte durch die zweite Ziffer i (Abbildung 3.1). Beispielsweise würde bei einer Tabelle, die die Attribute (A, B, C, D) umfaßt, und einem Index, der die Schlüsselattribute (A, B) hat, die Attributbeziehung 35

$$\begin{aligned} P_T &= (B, C) \\ P_I &= (A, B, C) \end{aligned}$$

bedeuten.

Weiterhin wurde der Spezialfall, daß die Tabelle gar nicht partitioniert ist, als Index-Attributbeziehung $i = 0$ zusätzlich vorgesehen. Die erste Ziffer t bezeichnet in diesem Fall die Beziehung zwischen Tabellen-Partitionierungsattributen und Indexschlüssel.

4 Implementierung in Oracle8i

Nachdem im letzten Abschnitt ein Konzept vorgestellt wurde, das die verschiedenen Varianten der Tabellen- und Indexpartitionierung klassifiziert, soll nun auf die konkrete Implementierung der Tabellen- und Indexpartitionierung in Oracle8i eingegangen werden. Die folgenden Betrachtungen und Messungen wurden mit Oracle Version 8.1.5 erstellt und spiegeln den Stand der Implementierung von Tabellen- und Indexpartitionierung in dieser Version wieder. Dennoch sollten die Betrachtungen in diesem Abschnitt auch auf andere Versionen von Oracle8i übertragbar sein und den Vergleich mit anderen DBMS erleichtern.

Die Implementierung von Tabellen- und Indexpartitionierung in Oracle8i, die im folgenden vorgestellt wird, ist in [Heß97] beschrieben und wird im Rahmen von [Bau00] ausführlich bewertet.

4.1 Umsetzung des Partionierungskonzepts

4.1.1 Equi-Partitionierung

Mehrere Objekte werden genau dann als equi-partitioniert bezeichnet, wenn die Partitionierung nach denselben Attributen auf dieselbe Art und Weise erfolgt. Objekte in diesem Sinne können sowohl Tabellen als auch Indexe sein. Equi-Partitionierung ist selbstverständlich nur dann möglich, wenn die zu betrachtenden Objekte gleiche Attribute enthalten. Daher kommt diese Möglichkeit vor allem für Tabellen und ihre Indexe in Betracht.

4.1.2 Lokale/Globale Partitionierung

Falls Tabellen und ihre Indexe equi-partitioniert sind, gehört zu jeder Indexpartition genau eine Tabellenpartition. In diesem Fall kann man Indexe in Oracle8i als lokal partitioniert kennzeichnen, indem beim Anlegen das Schlüsselwort `local` angegeben wird. Die lokale Partitionierung kann dann vom Optimierer vorteilhaft ausgenutzt werden.[2] Lokal partitionierte Indexe haben weiterhin den Vorteil, daß Oracle8i die Indexpartitionen automatisch verwaltet. Wird eine Tabellenpartition angelegt, verändert oder gelöscht, passiert dasselbe automatisch mit der zugehörigen Indexpartition. Diese Eigenschaft ist ein wesentlicher administrativer Vorteil und kann es unter Umständen wert sein, leichte Leistungseinbußen dafür in Kauf zu nehmen.

Liegt keine Equi-Partitionierung vor, d. h. nicht alle Schlüssel einer Indexpartition verweisen auf Tupel in derselben Tabellenpartition, können Indexe nur global partitioniert angelegt werden. Die Vorteile lokaler Partitionierung können in diesem Fall nicht ausgenutzt werden. Die Unterschiede zwischen lokaler und globaler Partitionierung sind in den Abbildungen 4.1 und 4.2 schematisch illustriert. In beiden

2. Zumindest behauptet das [LR99] – wie Abschnitt 6 zeigt, scheinen in der Praxis die Probleme des Optimierers mit partitionierten Indexen zu überwiegen.

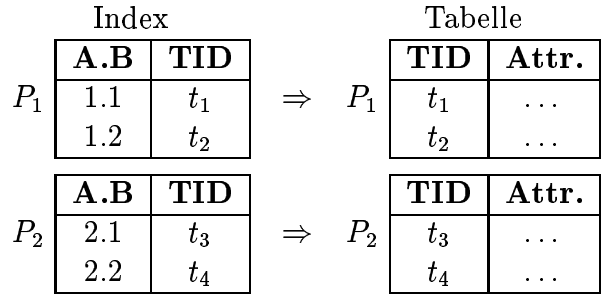

Abbildung 4.1: Lokale Partitionierung

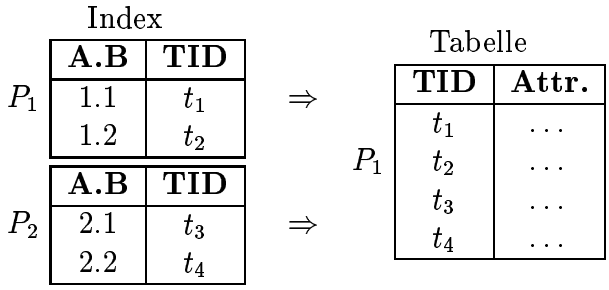

Abbildung 4.2: Globale Partitionierung

Fällen besitzt eine Tabelle einen Index, der als Indexschlüssel die konkatenierten Attribute A und B besitzt. Im ersten Fall in Abbildung 4.1 sind Tabelle und Index equi-partitioniert und der Index kann als lokal deklariert werden. Im zweiten Fall in Abbildung 4.2 sind Tabelle und Index nicht equi-partitioniert, so daß der Index als global partitioniert angelegt werden muß. Leider ist Oracle8i nicht in der Lage, die Equi-Partitionierung von Index und Tabelle automatisch zu erkennen, so daß undeklarierte, aber mit der zugehörigen Tabelle equi-partitionierte Indexe standardmäßig als global partitionierte Indexe angelegt werden.

4.1.3 *Prefixed*/*non-prefixed* Partitionierung

In Oracle8i wird neben lokal/global noch eine Unterscheidung hinsichtlich der Reihenfolge der Partitionierungsattribute gemacht [Fee99, Heß97]. Eine Indexpartitionierung ist *prefixed*, wenn die Partitionierungsattribute ein linker Präfix des Indexschlüssels sind. Beispiele für eine *prefixed* und *non-prefixed* Partitionierung eines Indexes, der die konkatenierten Attribute A und B als Schlüssel hat, sind in den Abbildungen 4.3 und 4.4 gegeben.

Die Ursache für diese Unterscheidung liegt in der Implementierung von Mehr-Attribut-Indexen als eindimensionale Indexstrukturen. Existiert ein Indexschlüssel aus mehreren Attributen, so wird deren Konkatenation im Index benutzt. Somit hängt die Sortierreihenfolge des Indexes hauptsächlich von dem Attribut ab, das am weitesten links steht, weiterhin vom zweiten Attribut von links, usw. Wenn jetzt eine Partitionierung des Indexes vorgenommen wird, die die Partitionen nach einem linken Präfix des Indexschlüssels aufteilt, so bleibt innerhalb der einzelnen Indexpartitionen

Index

	A.B	TID
P_1	1.1	t_1
	1.2	t_2
P_2	2.1	t_3
	2.2	t_4

Abbildung 4.3: *Prefixed* Partitionierung

Index

	A.B	TID
P_1	1.1	t_1
	2.1	t_3
P_2	1.2	t_2
	2.2	t_4

Abbildung 4.4: *Non-prefixed* Partitionierung

die Sortierreihenfolge erhalten und die Partitionen lassen sich wieder so zusammenfügen, daß der nicht partitionierte Index in Sortierreihenfolge zu erkennen ist. Sind dagegen die Partitionierungsattribute kein linker Präfix des Indexschlüssels, so bleibt die Sortierreihenfolge innerhalb der einzelnen Indexpartitionen nicht erhalten und die Partitionen lassen sich nicht mehr zum ursprünglichen Index zusammenfügen. Falls Bitmap Indexe benutzt werden, lassen sich Mehr-Attribut-Indexe jedoch auch effizient durch eine entsprechende Anzahl von Ein-Attribut-Indexen erzeugen, da bei Bitmap Indexen die Anfrageergebnisse mehrerer Indexe effizient auf Bitmap-Ebene verknüpft werden können. Die Partitionierung der in diesem Fall verwendeten Ein-Attribut-Indexe ist natürlich immer *prefixed*.

Grundsätzlich ist zu beachten, daß sich die Eigenschaft *prefixed/non-prefixed* auf den Zusammenhang zwischen Indexschlüssel und Index-Partitionierungsattributen bezieht und lokal/global auf den Zusammenhang zwischen Tabellen- und Index-Partitionierungsattributen. Das heißt insbesondere, daß die beiden Eigenschaften unabhängig voneinander sind. Das kleine Beispiel in Tabelle 4.1 verdeutlicht diesen Zusammenhang. Es zeigt die vier Partitionierungsmöglichkeiten anhand einer Tabelle, die mindestens die Attribute A und B in dieser Reihenfolge enthält, und ihres Indexes, der diese beiden Attribute als Indexschlüssel hat. Die Abkürzung *IS* bedeutet Indexschlüssel, P_T und P_I stehen für Partitionierungsattribute von Tabelle bzw. Index. Von diesen vier Möglichkeiten ist eine, nämlich *non-prefixed*/global, in Oracle8i nicht erlaubt [Heß97].

$IS = (A, B)$		
	lokal	**global**
prefixed	$P_T = (A)$, $P_I = (A)$	$P_T = (B)$, $P_I = (A)$
non-prefixed	$P_T = (B)$, $P_I = (B)$	$P_T = (A)$, $P_I = (B)$

Tabelle 4.1: Beispiel für lokale/globale und *prefixed*/*non-prefixed* Partitionierung

4.1.4 B*-Baum/Bitmap Indexe

Als Indexarten werden von Oracle8i standardmäßig die Varianten B*-Baum und Bitmap unterstützt. Beide Indexarten lassen sich partitionieren, allerdings muß die Indexart in allen Indexpartitionen gleich sein. Bitmap Indexe lassen sich nur lokal partitionieren, weil der verwendete Algorithmus zur Konvertierung von Indexeinträgen zu *Row IDs* keine Information über *Tablespaces* besitzt – von einem Bitmap Index referenzierte Tupel müssen immer demselben *Tablespace* und damit derselben Partition angehören wie der Index bzw. die Indexpartition selbst.

Sowohl B*-Baum als auch Bitmap Indexe sind in Oracle8i immer als Sekundärindexe implementiert. Primärindexe gibt es in Oracle8i nur bei indexorganisierten Tabellen, die im Rahmen dieser Arbeit aber nicht untersucht werden.

4.1.5 *Range-*/*Hash-*/*Composite*-Partitionierung

Indexe und Tabellen können in Oracle8i auf drei Arten partitioniert werden: *Range, Hash* oder *Composite*. Die *Range*-Partitionierung (Bereichspartitionierung) legt die Partitionen durch Aufteilung des Wertebereichs der Partitionierungsattribute in zusammenhängende Bereiche fest. Die Bereichsgrenzen müssen bei der Partitionsdefinition angegeben werden. Falls mehrere Partitionierungsattribute vorhanden sind, wird zur Bereichsbestimmung der SQL-Vektorvergleich [ISO92] angewandt. Das heißt, daß die Attribute von links nach rechts verglichen werden. Das erste Attribut, daß größer oder kleiner als die zugehörige Bereichsgrenze ist, legt bereits die Partition fest. Nur bei Gleichheit wird das weiter rechts stehende Attribut verglichen. Damit wird die Partitionierung fast vollständig durch das am weitesten links stehende Attribut festgelegt. Der SQL-Vektorvergleich kommt der Implementation der Indexe durch B*-Bäume sehr entgegen (vgl. [SH99]), ist allerdings bei Bitmap Indexen nicht unbedingt sinnvoll und demnach als Einschränkung zu betrachten.

Bei der *Hash*-Partitionierung werden die Tupel entsprechend einer Hash-Funktion über die Partitionen verteilt. Als Partitionierungsfunktion kommt eine systemgenerierte Funktion zum Einsatz, über deren Verteilungseigenschaften keine Aussagen getroffen werden können. Nach experimentellen Beobachtungen ist die Verteilung erst ab einer größeren Anzahl von verschiedenen Attributwerten gleichmäßig, denn bei einer *Hash*-Partitionierung nach einem Attribut mit nur sieben verschiedenen Werten auf sieben Partitionen ergab sich eine so ungleichmäßige Verteilung der Tupel, daß einige Partitionen ganz leer blieben. Daher eignen sich nur Attribute mit größerem Wertebereich als Partitionierungsattribute für *Hash*-Partitionierung. Weiterhin ist die

Hash-Partitionierung von Indexen nur als lokale Partitionierung möglich [LO99], d. h. in diesem Fall muß die zugehörige Tabelle ebenso *hash*-partitioniert sein. Allerdings ist es möglich, zu einer *hash*-partitionierten Tabelle einen global *range*-partitionierten Index anzulegen.

Die *Composite*-Partitionierung ist eine zweistufige Hintereinanderschaltung von *Range*- und *Hash*-Partitionierung. Dabei wird das Objekt zuerst per *Range*-Partitionierung in Partitionen zerlegt, welche wiederum per *Hash*-Partitionierung in Subpartitionen aufgeteilt werden. Die Effekte bei dieser Partitionierungsart setzen sich aus den Einzeleffekten von *Range*- und *Hash*-Partitionierung zusammen [Bau00], mit einer interessanten Ausnahme: mit einer *Composite*-Partitionierung, die aus einer einzigen Partition in der ersten Stufe und mehreren Partitionen in der zweiten Stufe besteht, ließe sich ein global *hash*-partitionierter Index erzeugen. Da diese Möglichkeit sehr speziell ist und ansonsten hier wenig neu dazukommt, wird die *Composite*-Partitionierung im Rahmen dieser Arbeit nicht weiter untersucht.

In der Literatur wird weiterhin die abgeleitete Partitionierung erwähnt [Rah94]. Bei abgeleiteter Partitionierung bestimmt ein Attribut einer anderen Tabelle die Partition, was insbesondere für Fremdschlüsselbeziehungen und Stern-Schemata interessant ist. Diese Partitionierungsart wird von Oracle8i nicht unterstützt, genausowenig wie die Partitionierung nach selbstdefinierten Funktionen. Lediglich die Partitionierung von Indexen, deren Schlüssel selbstdefinierte Funktionen sind, ist möglich.[3] Allerdings unterstützt diese Art von Index auch nur Anfragen an genau diese Funktion, so daß der weitaus größte Teil der Flexibilität von Partitionierung nach selbstdefinierten Funktionen verloren ist. Auch das soll hier nicht weiter untersucht werden.

4.2 Mögliche Partitionierungsschemata

Die Partitionierungsmöglichkeiten in Oracle8i sind nicht in der Lage, das in Abschnitt 3 vorgestellte Partitionierungskonzept vollständig umzusetzen. Daher soll untersucht werden, welche Teile des Partitionierungskonzepts in Oracle8i realisiert werden können.

Von den vier Partitionierungsarten *HH HR RH RR* (Tabellenpartitionierung/ Indexpartitionierung, siehe 3.3) ist eine, *RH*, nicht möglich. Die Partitionierungsart *RH* würde bedeuten, daß die Tabelle *range*- und der Index *hash*-partitioniert ist. Da aber nur lokal *hash*-partitionierte Indexe erlaubt sind und bei *range*-partitionierter Tabelle eine Equi-Partitionierung prinzipiell nicht möglich ist, fällt diese Partitionierungsart aus.

Ebenso sind einige der Attributbeziehungen von vornherein ausgeschlossen. Um diesen Zusammenhang zu verdeutlichen, ist die Matrix der Attributbeziehungen (Abbildung 3.1 auf Seite 17) in Abbildung 4.5 wiederholt, wobei alle möglichen Beziehungen durch ein Kästchen gekennzeichnet sind. Oracle8i unterstützt lokal parti-

3. Das heißt, daß man in Oracle8i einen Index nicht nur über ein Attribut A einer Relation erstellen kann, sondern auch über eine selbstdefinierte Funktion $f(A)$. Wenn man diesen Index anschließend partitioniert, fließt natürlich die Funktion $f(A)$ auch in die Partitionierungsfunktion mit ein.

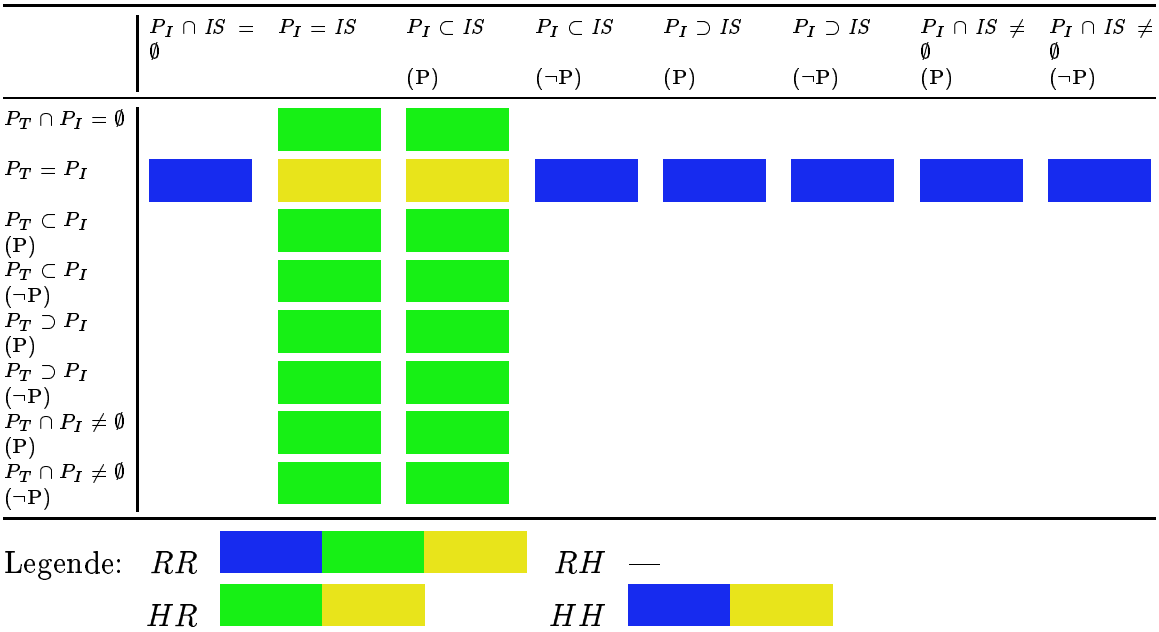

Abbildung 4.5: Mögliche Attributbeziehungen pro Partitionierungsart

tionierte *prefixed*- und *non-prefixed*-Indexe, aber nur global partitionierte *prefixed*-Indexe (4.1.3). Daher sind in in Abbildung 4.5 alle Spalten außer 2 und 3 generell weiß gelassen. Die Attributbeziehungen in den Spalten 2 und 3[4] sind *prefixed* Varianten, wobei der Fall 2 natürlich keinen echten Präfix darstellt. Die Fälle in den Spalten 1 und 4 bis 8 sind keine *prefixed* Partitionierungen im Oracle-Sinne, da dort P_I mehr Elemente umfaßt als IS. Daher sind die Attributbeziehungen in den Spalten 1 und 4 bis 8 ausgeschlossen. Eine Ausnahme bildet die zweite Zeile, denn die Beziehungen dort sind die lokalen Partitionierungen. Da Oracle8i lokale Partitionierungen unabhängig von *prefixed* oder *non-prefixed* unterstützt, sind in dieser Zeile alle Kombinationen möglich.

Nicht alle dieser (bereits eingeschränkten) Attributbeziehungen sind jedoch in jeder Partitionierungsart möglich: in der Partitionierungsart HR fallen die lokalen Partitionierungen aus, da aufgrund der *Hash*-Partitionierung der Tabelle und der *Range*-Partitionierung des Indexes Equi-Partitionierung unmöglich ist. Weiterhin fallen alle Kombinationen *außer* den lokalen Partitionierungen in der zweiten Zeile bei der Partitionierungsart HH aus, da *hash*-partitionierte Indexe in Oracle8i nur lokal partitionierbar sind [LO99]. Diese Einschränkung wird in Abbildung 4.5 durch die verschiedene Schattierung der Felder deutlich gemacht: für jede Partitionierungart kommen nur die in der Legende aufgeführten Felder in Betracht. Das heißt, daß für die Partitionierungart RR alle gekennzeichneten Felder tatsächlich möglich sind, während für die anderen beiden Partitionierungarten nur eine Teilmenge daraus erlaubt ist.

4. $P_I = IS$ und $P_I \subset IS$ (Präfix)

Eine weitere Einschränkung besteht bei der Partitionierung von Bitmap Indexen. Dort ist generell nur lokale Partitionierung möglich. Das heißt, daß alle Attributbeziehungen außer die in der 2. Zeile nicht erlaubt sind. Auch die Partitionierungsart HR ist in diesem Fall nicht mehr erlaubt, da es sich dabei um keine lokale Partitionierung handelt. Insgesamt bleiben für die Partitionierung von Bitmap Indexen nur die Partitionierungsarten RR und HH mit den lokalen Partitionierungen (2. Zeile) übrig.

5 Ablauf der Messung

5.1 Testdaten

Um Leistungsgewinne durch Indexpartitionierung messen zu können, ist eine geeignete und hinreichend große Testdatenbank nötig, gegen die bei verschiedenen Partitionierungsschemata Anfragen gestellt werden können. Allerdings genügt eine große Datenbank allein nicht, denn für die Effektivität von Index- und Tabellenpartitionierung ist allein die Größe einzelner Tabellen entscheidend [Mül00]. Daher bot sich das Schema des TPC-H Benchmarks [TPC99], der aus dem Data Warehouse-Umfeld stammt, an. Dieses Schema (Abbildung 5.1) enthält eine sehr große Faktentabelle `lineitem`, die sich gut zur Partitionierung eignet. Das Meßprinzip besteht darin, daß immer dieselben Anfragen gegen diese Tabelle, die jedesmal unterschiedlich partitioniert wird, gestellt werden.

Der TPC-H Standard definiert eine Größe *SF (scale factor)*, mit der die meisten Tabellen der Datenbank linear skaliert werden können. Bei $SF = 1$ ergibt sich laut [TPC99] eine Datenbankgesamtgröße von ca. 1 GB ohne Zugriffspfade, wobei sich in der Praxis abhängig von Partitionierung und interner Fragmentierung leicht höhere Werte herausstellten. Davon entfallen ca. 640 MB auf die Tabelle `lineitem`, die in diesem Fall aus ca. 6 000 000 Tupeln besteht. Die Tabelle `lineitem` bietet also günstige Voraussetzungen, partitionierungsbedingte Leistungssteigerungen zu messen.

Die Tabelle `lineitem` hat 16 Attribute, die in Abbildung 5.2 dargestellt sind. Die Attribute `orderkey` und `linenumber` bilden den Primärschlüssel und sind aufgrund ihrer hohen Selektivität für die Anfragen gut geeignet. Diese Attribute bilden bei B*-Baum Indexen den Indexschlüssel, und die Attribute `partkey` und `suppkey` werden als zusätzliche Partitionierungsattribute benutzt. Bei der Messung mit Bitmap-Indexen sind diese Attribute (mit Ausnahme von `linenumber`) weniger gut geeignet, da ihr Wertebereich relativ groß ist. Stattdessen werden die Attribute `quantity`, `discount` und `tax` benutzt, deren Wertebereich ausreichend klein ist. Eine Sonderstellung nimmt das Attribut `linenumber` ein, das bei beiden Indexarten benutzt wird. Es ist zwar mit einem Wertebereich von 7 Werten klar dem Bitmap Index zuzuordnen,

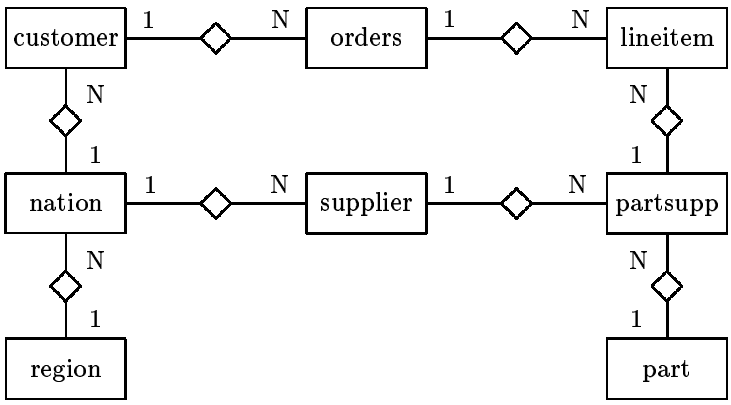

Abbildung 5.1: Das Schema des TPC-H Benchmarks

lineitem	
orderkey	*B*-Baum*
partkey	*B*-Baum*
suppkey	*B*-Baum*
linenumber	*B*-Baum, Bitmap*
quantity	*Bitmap*
extendedprice	
discount	*Bitmap*
tax	*Bitmap*
returnflag	
linestatus	
shipdate	
commitdate	
receiptdate	
shipinstruct	
shipmode	
comment	

Abbildung 5.2: Attribute der Tabelle `lineitem`

aber da es Teil des Primärindexes ist, kommt auch die Messung mit B*-Baum Indexen nicht ohne es aus. Als Alternative kämen nur Sekundärschlüssel mit mindestens drei Attributen in Betracht, die aufgrund der daraus folgenden deutlich höheren Komplexität der Partitionierungsattribute die Klarheit der Ergebnisse trüben würden.[5] Wenn `linenumber` aber in den Anfragen vorkommen soll, muß es auch Bestandteil des Indexschlüssels sein, denn ohne Indexnutzung wäre die Indexpartitionierung sinnlos!

Nach Auswahl der Indexschlüssel- und Partitionierungsattribute können nun die Partitionierungsattribute entsprechend der Attributbeziehungen konkretisiert werden. Wenn man den Indexschlüssel als konstant voraussetzt, ergeben sich durch Anwendung der in Abschnitt 3.4 vorgestellten Beziehungen die konkreten Partitionierungsattribute. Das ist beispielsweise für den Fall „B*-Baum Index und mehrere Attribute“ in Tabelle 5.1 nachvollzogen. Dort wurden die Attribute `orderkey` und `linenumber` als Indexschlüssel fest gewählt und alle anderen Partitionierungsattribute davon abgeleitet. Beim Ableiten der Attribute gibt es an manchen Stellen mehrere Möglichkeiten. Im Beispiel (und auch bei der Messung) wurden die Partitionierungsattribute so gewählt, daß Partitionierung nach `linenumber` möglichst vermieden wird, da dieses Attribut einen sehr kleinen Wertebereich hat. Leider konnte das nicht vollständig vermieden werden, denn z. B. die Attributbeziehungen 40[6] und 24[7] ließen keine andere Wahl. Ein weiterer kleiner Schwachpunkt sind die Attributbeziehungen

5. Desweiteren ist die *Range*-Partitionierung nach vielen Attributen aufgrund des SQL-Vektorvergleichs nicht sinnvoll.
6. $P_T \subset IS(\neg\mathrm{P})$, Index n. p. $\Longrightarrow P_I = \emptyset, P_T = (\mathrm{ln})$
7. $P_I \subset IS(\neg\mathrm{P}), P_T = P_I \Longrightarrow P_I = (\mathrm{ln}), P_T = (\mathrm{ln})$

IS = ok, ln

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	P_T=(pk,sk)	—	P_I=(ok,ln) P_T=(pk,sk)	P_I=(ok) P_T=(pk,sk)	—	—	—	—	—
$P_T = P_I$	P_T=(ok,ln)	P_I=(pk,sk) P_T=(pk,sk)	P_I=(ok,ln) P_T=(ok,ln)	P_I=(ok) P_T=(ok)	P_I=(ln) P_T=(ln)	P_I=(ok,ln,pk) P_T=(ok,ln,pk)	P_I=(pk,ok,ln) P_T=(pk,ok,ln)	P_I=(ok,pk) P_T=(ok,pk)	P_I=(pk,ln) P_T=(pk,ln)
$P_T \subset P_I$ (Präfix)	P_T=(ok)	—	P_I=(ok,ln) P_T=(ok)	P_I=(ok) P_T=$\emptyset$	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	P_T=(ln)	—	P_I=(ok,ln) P_T=(ln)	P_I=(ok) P_T=$\emptyset$	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	P_T=(ok,ln,pk)	—	P_I=(ok,ln) P_T=(ok,ln,pk)	P_I=(ok) P_T=(ok,pk)	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	P_T=(pk,ok,ln)	—	P_I=(ok,ln) P_T=(pk,ok,ln)	P_I=(ok) P_T=(pk,ok)	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	P_T=(ok,pk)	—	P_I=(ok,ln) P_T=(ok,pk)	P_I=(ok) P_T=(pk)	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	P_T=(pk,ln)	—	P_I=(ok,ln) P_T=(pk,ln)	P_I=(ok) P_T=(pk)	—	—	—	—	—

Tabelle 5.1: Partitionierungsattribute. Die Abkürzungen ok, pk, sk, ln stehen für `orderkey`, `partkey`, `suppkey` und `linenumber`

33[8] und 34[9], die sich nicht voneinander unterscheiden. Einziger Ausweg wären auch hier Schemata mit einem Indexschlüssel aus drei Attributen, was aus den bereits erwähnten Gründen nachteilig ist.

Die Wahl der Partitionierungsattribute in anderen Partitionierungsschemata verläuft analog und soll hier nicht weiter behandelt werden.

5.2 Anfragen

Nachdem die verschiedenen Partitionierungsschemata erzeugt worden sind, muß nun herausgefunden werden, welche davon leistungsmäßige Vorteile erbringen. Dazu wird das Partitionierungsschema in der Datenbank erzeugt und drei verschiedene Anfragen werden wiederholt an die Datenbank gestellt.

a) Punktanfrage Bei der Punktanfrage (Beispiel siehe Abbildung 5.3) werden die beiden Indexattribute, also `orderkey` und `linenumber` bei B*-Baum Indexen und `quantity` und `discount` bei Bitmap Indexen, mit einem Wert belegt und alle Tupel, die dazu passen, aus der Datenbank geholt. Das ergibt eine Ergebnismenge mit keinem oder einem Tupel bei B*-Baum Indexen und größere Ergebnismengen bei Bitmap Indexen, da aufgrund der kleineren Wertebereiche der bei dieser Messung verwendete Attribute die Selektivität schlechter ist.

Nach einem Vorspann von drei Befehlen, die *Intra-Query*-Parallelität ermöglichen, das *Tracing* einschalten und der Anfrage eine eindeutige ID zuweisen, um die *Trace Files* später auffindbar zu machen, beginnt eine Folge aus 50 `select`-Anweisungen. Die Selektionskriterien werden mit Hilfe eines selbstgeschriebenen Pseudo-Zufallszahlengenerators erzeugt, der für alle Messungen die gleichen Anfragen erzeugt. Das Attribut `quantity` in der `select`-Klausel wurde gewählt, um bei der Anfrageauswertung einen Zugriff auf die Datenbank zu erzwingen. Könnte Oracle8i die Anfragen rein aus dem Index beantworten, würde die Partitionierung der Tabelle keinen Einfluß mehr haben und das Zusammenspiel zwischen Index- und Tabellenpartitionierung könnte nicht mehr erfaßt werden.

b) Bereichsanfrage Die Bereichsanfrage (Beispiel siehe Abbildung 5.4) unterscheidet sich wenig von der Punktanfrage, so daß alles im vorhergehenden Punkt gesagte auch hier zutrifft. Die Anfragen werden so gewählt, daß die Selektivität pro Partition kleiner als 5 % bleibt, da sonst der Oracle-Optimierer in jedem Fall den *Partition Scan* als Ausführungsstrategie vorzieht und der Index nicht genutzt wird (siehe auch 5.3.1). Diese Anfrage wird pro Partitionierungsschema nur 10 mal wiederholt, um die Meßzeit auf der verwendeten Hardware im Rahmen zu halten. Dabei sind diese 10 Anfragen so gewählt, daß die zu selektierenden Bereiche manchmal komplett innerhalb einer Partition liegen und manchmal Partitionsgrenzen überschreiten, so

8. $P_I \subset IS(\mathrm{P}), P_T \subset P_I(\mathrm{P}) \Longrightarrow P_I = (\mathrm{ok}), P_T = \emptyset$
9. $P_I \subset IS(\mathrm{P}), P_T \subset P_I(\neg\mathrm{P}) \Longrightarrow P_I = (\mathrm{ok}), P_T = \emptyset$

```
alter session enable parallel dml;
alter session set sql_trace=true;
select 'id:xxxxxxxx' from dual;
select l_quantity
  from li$p
  where l_orderkey=5041126 and l_linenumber=2;
...
```

Abbildung 5.3: Punktanfrage

```
...
select sum(l_quantity)
  from li$p
  where l_orderkey between 4789070 and 5089070
    and l_linenumber between 2 and 5;
...
```

Abbildung 5.4: Bereichsanfrage

daß die Reaktion von Oracle8i für verschiedene Konstellationen gemittelt wird. Allerdings liegt ein Bereich in niemals mehr als zwei Partitionen, da eine größere Zahl die vorhandene Hardware-Parallelität übersteigen würde (siehe 5.4).

c) Verbundanfrage Die Verbundanfrage (Beispiel in Abbildung 5.5) unterscheidet sich von den beiden vorhergehenden Anfragen dadurch, daß in ihr die partitionierte Tabelle `lineitem` mit einer nichtpartitionierten Tabelle verknüpft wird. Diese Anfrage entspricht damit einem typischen *Data Warehouse*-Szenario.

Neben der partitionierten `lineitem`-Tabelle wird die Tabelle `orders` (bei B*-Baum Indexen) bzw. die Tabelle `part` (bei Bitmap Indexen) herangezogen. Der Ausdruck in der `select`-Klausel ergibt zwar keinen „sinnvollen“ Wert, sorgt aber dafür, daß bei der Anfragebearbeitung neben den Indexen auch beide Tabellen gelesen werden. Die zweite Tabelle ist jeweils nicht partitioniert und hat einen Index für das angefragte Attribut. Auch bei dieser Anfrage wird die Selektivität auf unter 5 % pro `lineitem`-Partition verringert, um den generellen Gebrauch von *Partition Scans* durch den Optimierer zu verhindern.

```
...
select sum(o_totalprice)-sum(l_extendedprice)
  from li$p, orders
  where l_orderkey=o_orderkey
    and l_orderkey between 990715 and 1014378;
...
```

Abbildung 5.5: Verbundanfrage

5.3 Einfluß des Optimierers

Im Verlauf der Meßvorbereitung hat sich herausgestellt, daß der Oracle-Optimierer einen wesentlichen Einfluß auf die Leistungsfähigkeit der Anfragebearbeitung hat. Es liegt z. B. am Optimierer, Möglichkeiten des Partitionsausschlusses zu erkennen und zu nutzen. Daher soll im folgenden das Verhalten des Optimierers bei den benutzten Daten und Anfragen beleuchtet werden.

5.3.1 Indexzugriff oder *Table Scan*

Wie in 2.3 bereits angedeutet, hat der Optimierer bei jeder Selektion eine Entscheidung zwischen Indexnutzung (direkter Indexzugriff oder *Scan*) und *Table (Partition) Scan* zu treffen. Entscheidendes Kriterium dafür ist die Selektivität der Anfrage: ab einem bestimmten Punkt ist es schneller, die ganze Tabelle bzw. Partition sequentiell einzulesen, als den Lese-Schreib-Kopf des Plattenlaufwerks hin- und herzubewegen.

Aufgrund eingesetzter Techniken wie *Multi-Block I/O* [Bau99] bewertet der Optimierer *Scans* sehr günstig. Experimentell hat sich ergeben, daß der Optimierer auf dem verwendeten Rechner in der Standardeinstellung schon ab einer Selektivität von ca. 5 % den *Table Scan* gegenüber dem Indexzugriff vorzieht. Das liegt daran, daß Oracle8i durch die *Multi-Block I/O* Technik viele Blöcke in einer Operation einlesen kann, so daß aus Sicht des Optimierers das Lesen eines einzigen Indexknotens und eines 32 kB Datenblocks einer Tabelle gleich teuer ist. Allerdings ist zu bezweifeln, ob diese Annahme wirklich praxisgerecht ist. In den beiden *Profiles* in Abbildung 5.6 und 5.7 wurde die gleichen Bereichsanfragen mit relativ guter Selektivität gestellt. Im ersten Fall konnte der Optimierer frei über den Ausführungsplan entscheiden, während im zweiten Fall die Indexnutzung vorgegeben wurde. Die unterschiedlichen Antwortzeiten (*elapsed*) von 16,81 s und 2,59 s legen nahe, daß der *Scan* in Wirklichkeit teurer ist, als vom Optimierer angenommen und daher die Indexnutzung häufiger zu bevorzugen ist, als dies in der gegenwärtigen Implementierung der Fall ist.

In einigen Fällen verzichtet der Optimierer sogar generell auf Indexnutzung. So ist bei Voruntersuchungen zur Messung aufgefallen, daß zumindest in der benutzten Version Oracle8i 8.1.5 Indexe, die nicht mit `unique` gekennzeichnet sind, generell bei der Auswertung von Bereichsanfragen nicht berücksichtigt werden, obwohl das Kostenargument für die Indexnutzung sprechen würde. Es entstand sogar die Situation, daß der Optimierer bei einer Anfrage des Typs

```
select a from t where a = x;
```

einen Indexzugriff benutzte, während die Anfrage

```
select a from t where a between x and x;
```

mittels *Table Scan* ausgeführt wurde. Offensichtlich liegt hier eine Optimiererregel vor, die besagt, daß Bereichsanfragen egal welcher Art nicht über `non unique` Indexe ausgeführt werden dürfen. Weiterhin werden nichteindeutige Indexe im Zusammenhang mit Partitionierung gar nicht benutzt, selbst bei Punktanfragen nicht.

```
select sum(l_quantity) from lineitem
where l_partkey between 799 and 2799
  and l_suppkey between 106 and 116

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.02       0.49          0          0          0           0
Execute      1      0.00       0.06          0          0          0           0
Fetch        2      3.67      16.26       9605       9604          4           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      3.69      16.81       9605       9604          4           1

Misses in library cache during parse: 1
Optimizer goal: ALL_ROWS
Parsing user id: 25  (CKAUHAUS)

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE
    640   TABLE ACCESS FULL LINEITEM


Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: ALL_ROWS
      1   SORT (AGGREGATE)
    640    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'LINEITEM'
```

Abbildung 5.6: Bereichsanfrage ohne *Hint*

```
select /*+ index(lineitem) */ sum(l_quantity) from lineitem
where l_partkey between 799 and 2799
  and l_suppkey between 106 and 116

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.03          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.37       2.56        786        800          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.37       2.59        786        800          0           1

Misses in library cache during parse: 1
Optimizer goal: ALL_ROWS
Parsing user id: 25  (CKAUHAUS)

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE
    640   TABLE ACCESS BY INDEX ROWID LINEITEM
    641    INDEX RANGE SCAN (object id 8269)


Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: ALL_ROWS
      1   SORT (AGGREGATE)
    640    TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'LINEITEM'
    641     INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'L_P_S' (NON-UNIQUE)
```

Abbildung 5.7: Bereichsanfrage mit *Hint*

Die Kostenabschätzung zwischen Indexzugriff und *Scan* läßt sich im Wesentlichen durch zwei Oracle8i-Parameter beeinflussen [Bau99]:

- Mit `DB_FILE_MULTIBLOCK_READ_COUNT` läßt sich die Zahl der in einem *Multi-Block I/O* Vorgang eingelesenen Blöcke beeinflussen und damit indirekt über die Zahl der für eine Tabelle notwendigen Leseoperationen auch die Gewichtung des *Table* oder *Index Scan*. Bei der verwendeten Oracle8i-Installation hat dieser Parameter den Wert 32, d. h. mit einer Leseoperation werden 32 Blöcke eingelesen.

- Der Parameter `OPTIMIZER_INDEX_COST_ADJ` beeinflußt als Faktor direkt die Gewichtung der Indexzugriffskosten für den Optimierer. Mit Hilfe dieses Parameters kann der Indexzugriff generell billiger oder teurer bewertet werden, egal, ob es sich dabei um partitionierte Tabellen und Indexe handelt oder nicht. Bei Voruntersuchungen konnte eine realistischere Bewertung der Indexzugriffskosten durch Verstellen des Parameters von 1.0 auf 0.1 erreicht werden.

Im Rahmen dieser Arbeit werden alle Parameter jedoch bei ihren Standardwerten belassen, um das Verhalten von Oracle8i *„out of the box“* zu bewerten.

5.3.2 *Hints*

Als weitere Möglichkeit zur Einflußnahme auf Optimiererentscheidungen bietet sich der Einsatz von *Hints* an. *Hints* sind in Anfragen eingebaute Anweisungen, die den Ausführungsplan für bestimmte Anfrageteile festlegen bzw. beeinflussen. *Hints* bieten sehr weitgehende Möglichkeiten zur Einflußnahme und sind deshalb ein mächtiges Mittel zur Leistungssteigerung.

Dem stehen allerdings auch ernstzunehmende Nachteile gegenüber. Zum einen geht die Unabhängigkeit der Anfragen vom physischen Entwurf verloren, da diese Kenntnisse direkt in die Anfragen einfließen müssen. Insbesondere müssen bei Änderungen des physischen Entwurfs auch die Anfragen angepaßt werden (Anlegen/ Löschen eines Indexes etc.), was nicht wünschenswert ist. Zum anderen sind *Hints* datenabhängig. Die Indexnutzung zu erzwingen, wenn eine Anfrage eine gute Selektivität hat, mag sinnvoll sein. Allerdings ist nicht garantiert, daß dieselbe Anfrage auch in Zukunft noch eine gute Selektivität hat, da sich der Datenbestand ändern kann. Deshalb ist es eigentlich Aufgabe des Optimierers, die Ausführungspläne auch aufgrund vorliegender Statistiken über die Datenverteilung zu erstellen. Mit *Hints* nimmt man dem Optimierer diese Aufgabe teilweise ab und überträgt sie auf den menschlichen Datenbankprogrammierer.

Aus diesen Gründen wird im Rahmen dieser Arbeit auf *Hints* verzichtet.

5.3.3 Partitionsbezogene Optimierung

Durch Partitionierung erhält der *Scan* dadurch eine weitere Bevorzugung, weil bei partitionierten Tabellen und Indexen die Selektivität einer Anfrage pro Partition

schlechter wird. Daher wird bezogen auf die gesamte Tabelle natürlich auch die kritische Selektivität, ab der ein *Scan* bevorzugt wird, kleiner; sie teilt sich durch die Anzahl der Partitionen. Nun ist die partitionsbezogene Selektivität natürlich nicht in jeder Partition gleich groß, denn bei Bereichsanfragen können drei Fälle auftreten: der angefragte Bereich liegt entweder vollständig innerhalb, vollständig außerhalb oder teilweise in einer Partition. Der Optimierer entscheidet die Ausführungsstrategie aber nicht für jede Partition einzeln, sondern trifft nur eine globale Entscheidung, die gleichermaßen für jede Partition umgesetzt wird. Daher würde eine Bereichsanfrage mit insgesamt zu schlechter Selektivität, die eine Partition vollständig und zwei Partitionen teilweise (mit guter partitionsbezogener Selektivität) umfaßt, in allen drei Partitionen mittels eines *Table Partition Scan* ausgeführt werden, selbst wenn für die Partitionen lokale Indexe vorliegen und die Ausführung in den beiden teilweise berührten Partitionen per Indexzugriff wesentlich günstiger wäre. Auf der anderen Seite muß beachtet werden, daß die wünschenswerte Eigenschaft der partitionsbezogenen Optimierung einen um die Zahl der Partitionen höheren Optimierungsaufwand verursachen würde.

5.4 Meßmethode

Nachdem die einzelnen zu untersuchenden Partitionierungsschemata und die Anfragen geklärt sind, bleibt die Frage, wie die Leistungsfähigkeit eines bestimmten Partitionierungsschemas gemessen werden soll.

Der wichtigste Leistungsparameter ist die Antwortzeit T, also die Zeit, die von der Anfragestellung bis zum Erscheinen der Antwort vergeht. Weiterhin spielt der Durchsatz D eine Rolle, der aber aufgrund des verwendeten Rechners nicht gemessen werden konnte.

Um die Antwortzeit einer Datenbankanfrage zu bestimmen, gibt es grundsätzlich zwei Ansätze:

1. Aus den durchgeführten Plattenzugriffen und verbrauchten CPU-Sekunden (diese sind exakt bestimmbar) wird anhand eines Kostenmodells ein Wert ermittelt.

   Diese Vorgehensweise hat den schwerwiegenden Nachteil, daß zuerst ein Kostenmodell aufgestellt werden muß, was ein nichttrivialer Vorgang ist. Wie hoch sind z. B. die CPU-Kosten im Gegensatz zu den Plattenzugriffskosten? Weiterhin können speziell durch Partitionierung erzielte Effekte wie z. B. *Intra-Query*-Parallelität durch ein solches Modell nur schwer erfaßt werden.

2. Die vergangene Echtzeit zwischen Anfragestellung und -beantwortung wird gemessen und als Antwortzeit T direkt verwendet.

   Diese Methode hat den Vorteil, daß sie die Ausführungskosten prinzipiell besser erfaßt und zwei Nachteile: Auf dem zur Messung verwendeten Rechner arbeiten viele Benutzer im *Multi-User*-Betrieb. Dadurch kann die Antwortzeiten durch kurzzeitige Belastungsschwankungen verfälscht werden. Weiterhin spei-

chert Oracle8i benutzte Seiten im Puffer ab, um wiederholte Plattenzugriffe zu vermeiden. Demnach laufen dieselben Anfragen auf die Dauer schneller ab.

Nach sorgfältiger Abwägung wurde im Rahmen dieser Arbeit die zweite Vorgehensweise gewählt. Zusätzlich wurden Vorkehrungen getroffen, um die geschilderten Nachteile abzuschwächen bzw. zu vermeiden:

- Um Verfälschungen durch den *Multi-User*-Betrieb gering zu halten, wurden keine absoluten Antwortzeiten gemessen. Stattdessen wurden für jedes Partitionierungsschema zwei Messungen durchgeführt: eine auf einer nicht partitionierten Referenzdatenbank und eine auf der partitionierten Datenbank. Als Meßergebnis wird das Verhältnis zwischen diesen Antwortzeiten ausgewertet.

  Leider war es nicht möglich, diese beiden Messungen gleichzeitig laufen zu lassen, da das die vorhandene Hardware-Parallelität von zwei unabhängigen Plattensystemen (s. u.) bereits ausgenutzt und keinen Spielraum für Leistungssteigerungen durch *Intra-Query*-Parallelität gelassen hätte. Die beiden Messungen wurden direkt nacheinander ausgeführt in der Annahme, daß innerhalb kurzer Zeiten die Last auf dem Rechner nicht stark wechselt. Diese Annahme hat sich bei dem Vergleich der Antwortzeiten der Referenzmessungen untereinander weitgehend als richtig erwiesen. Wich die Antwortzeit für die Referenzmessung zu stark ab, wurde diese Messung wiederholt.

  Stichprobenartig wurde der Trend der Ergebnisse auf einem für kurze Zeit zur Verfügung stehenden separaten Rechner, auf dem keine weiteren *User* arbeiteten, nachgeprüft. Dabei wurden die Meßergebnisse bestätigt.

- Pufferbedingte Effekte wurden weitgehend ausgeschlossen, indem jede Anfrage 10 bzw. 50 Mal ausgeführt wurde. Da die Anfragen b (Bereich) und c (Verbund) wegen der längeren Einzellaufzeit nur 10 mal pro Messung ausgeführt wurden, begann jede Messung mit einer Warmlaufphase von einer Anfrage jedes Typs, um den Puffer vorzubelegen.

Die Messungen wurden auf einem Digital Alpha Rechner unter Tru64 Unix mit Oracle 8.1.5 durchgeführt. Zur Messung standen zwei unabhängige Plattensysteme mit SCSI-Schnittstelle zur Verfügung. Um Effekte durch *Intra-Query*-Parallelität zu erfassen, wurden die Partitionen so angeordnet, daß sie abweichend auf den beiden Platten liegen. Dadurch ist sichergestellt, daß bei Anfragen, die aus *range*-partitionierten Tabellen bzw. Indexen beantwortet werden, immer beide Plattensysteme genutzt werden, sofern der Zugriff auf mehr als eine Partition zur Beantwortung der Anfrage nötig ist. Bei *hash*-partitionierten Tabellen und Indexen ist eine so genaue Festlegung der benutzten Partitionen nicht möglich, allerdings werden bei dieser Partitionierungsart immer viele bzw. alle Partitionen benutzt, so daß sich auch hier Effekte durch *Intra-Query*-Parallelität auswirken können.

Zusammengefaßt hat jede Messung den folgenden Ablauf:

1. Das Partitionierungsschema wird mit Indexart, Attributzahl, Partitionierungsart und Attributbeziehung festgelegt (Abschnitte 3.1 bis 3.4).

2. Die Zulässigkeit des Partitionierungsschemas unter Oracle8i wird überprüft (Abschnitt 4.2). Falls das Partitionierungsschema unzulässig ist, wird mit dem nächsten Partitionierungsschema fortgefahren.

3. Die Meßdatenbank wird aus der Referenzdatenbank (die immer gleich bleibt) entsprechend des Partitionierungsschemas erzeugt (Abschnitt 5.1).

4. Für jede der drei Anfragen (a, b und c) wird der Ausführungsplan erfragt und abgespeichert (Abschnitt 5.2).

5. Die Anfragen a, b und c werden nach einem Durchgang zum Warmlaufen 50 bzw. 10 mal sowohl auf der Referenzdatenbank als auch auf der partitionierten Datenbank ausgeführt und die Laufzeitdaten in einem *Trace File* festgehalten (Abschnitt 5.4).

6. Die Antwortzeiten werden aus den Laufzeitdaten extrahiert, zusammengefaßt und mit den Ausführungsplänen zusammen gespeichert.

6 Meßergebnisse

Mit der im letzten Abschnitt beschriebenen Meßmethode wurden die verschiedenen im Abschnitt 3 vorgestellten Index- und Tabellenpartitionierungs-Schemata untersucht, soweit dies innerhalb der im Abschnitt 4 gezeigten Implementierungsgrenzen möglich war. Die hier gewonnenen Ergebnisse sind stark von verschiedenen Faktoren der Meßumgebung abhängig. So wären die Daten möglicherweise bei höherer Hardware-Parallelität anders ausgefallen, da im Rahmen dieser Arbeit nur eine zweifache Hardware-Parallelität (siehe 5.4) ausgenutzt werden konnte. Auch kann es sein, daß eine andere Oracle8i-Version als die hier verwendete Version 8.1.5 mit verschiedenen Partitionierungskonstellationen anders und besser umgehen kann, da gerade die Partitionierung ein Bereich ist, an dem in den letzten Oracle8i-Versionen intensiv seitens Oracle gearbeitet worden ist. Demnach sind die hier vorgestellten Ergebnisse wahrscheinlich nicht stabil gegenüber einer Portierung in andere Umgebungen.

Da das Datenmaterial recht umfangreich ist, soll es wie folgt gegliedert werden. Zuerst sollen die Ergebnisse nach der *Partitionierungsart* (3.3) unterteilt werden, d. h. ob Index und Tabelle *range*- oder *hash*-partitioniert sind. Innerhalb dieser Abschnitte sollen die Ergebnisse nach *Indexart* (3.1) und *Attributzahl* (3.2) gegliedert sein, da in der Messung B*-Baum Indexe mit einem oder zwei Attributen und Bitmap Indexe mit zwei Attributen berücksichtigt werden. Wenn Partitionierungsart, Indexart und Attributzahl feststehen, wird weiterhin nach *Anfrage* (5.2) unterteilt: Punkt-, Bereichs- und Verbundanfrage. Zu jeder der genannten Kombinationen gibt es dann eine Matrix, in der die Meßwerte für die verschiedenen *Attributbeziehungen* (3.4) eingetragen sind. Diese Matrix entspricht dem in Abbildung 3.1 auf Seite 17 vorgestellten Schema, wobei zusätzlich als 0. Spalte eine reine Tabellenpartitionierung ohne Indexpartitionierung aufgenommen ist.

Die Meßwerte werden in den Diagrammen als eine Matrix dargestellt. Die in der Matrix enthaltenen Werte entsprechen dem durchschnittlichen Antwortzeitverhältnis relativ zur Referenzanfrage (5.4), wobei dieses Verhältnis in den Diagrammen durch einen Balken mit logarithmischem Maßstab dargestellt sind. Balken auf der Nullebene bedeuten, daß die Anfrage im partitionierten Fall und die Referenzanfrage gleich schnell liefen. Balken nach oben bedeuten, daß die Anfrage im partitionierten Fall schneller lief als die Referenzanfrage, Balken nach unten umgekehrt. Neben den Diagrammen, die sich in diesem Kapitel befinden, sind die Verhältnisse der Antwortzeiten in den Tabellen im Anhang A direkt aufgeführt. Dort bezeichnet die erste Zahl die Antwortzeit der Referenzanfrage und die zweite Zahl die Antwortzeit der Meßanfrage. Danach folgt ein ein Kennwert, der aus der Analyse der Anfrageausführungspläne hervorgeht: Die Buchstaben T und I geben an, ob die der Anfrageausführung zugrundeliegende Funktion mit einem Index- oder Tabellenzugriff begonnen hat. Wenn die zugrundeliegende Funktion ein Indexzugriff war, bedeutet das natürlich, daß ein Tabellenzugriff folgte, da die Meßanfragen nicht allein aus dem Index beantwortet werden können. Die darauf folgende Zahl bezeichnet die Anzahl der zugegriffenen Partitionen. Da insgesamt 7 Partitionen benutzt werden, ist 7 der schlechteste und 1 der beste Wert. Weiterhin gibt es noch die Möglichkeit, daß nur *?* vermerkt ist. In

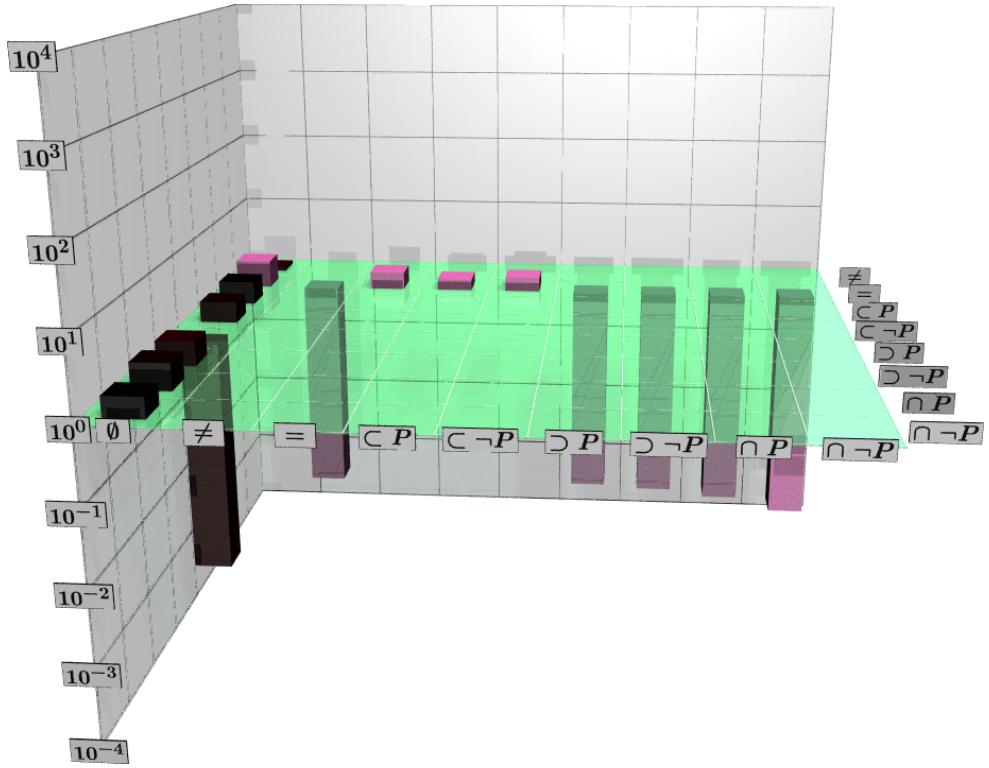

Abbildung 6.1: B*-Baum Index, 2 Attribute, HH Partitionierung, Punktanfrage

diesem Fall gab das *Trace File* keine Auskunft zur konkreten Anfrageauswertung, weil im DBMS Listenoperatoren verwendet worden sind, die eine konkrete Ermittlung der benutzten Partitionen nicht ohne weiteres zulassen.

6.1 Tabelle: *Hash*, Index: *Hash* (HH)

In der ersten Messung sind sowohl Tabelle als auch Index *hash*-partitioniert. Das heißt, daß bei beiden eine möglichst gleichmäßige Verteilung anhand der *Hash*-Funktion vorgenommen wird, die nicht lokalitätserhaltend ist. Zu erwarten ist, daß die Punktanfragen profitieren, da in diesem Fall ein Partitionsausschluß durchgeführt werden kann. Leistungsverbesserungen bei den Bereichsanfragen sind eher nicht zu erwarten, da die angefragten Bereiche sowohl bei der Tabelle als auch beim Index wegen der *Hash*-Partitionierung alle Partition berühren und somit keine Reduktion der Datenmenge möglich ist. Bei den Verbundanfragen können zusätzlich aber leichte Verbesserungen durch eine partitionsweise ausgeführte Verbundoperation eintreten.

6.1.1 B*-Baum Indexe über zwei Attribute

Die Meßwerte der Punktanfragen (Abbildung 6.1) bestätigen die Erwartung nur teilweise. Es fällt auf, daß die größte Verbesserung bei der Attributkombination 20 (Attributkombinationen siehe Abbildung 3.1 auf Seite 17 und Tabelle 5.1 auf Seite 27) auftritt, bei der die Tabelle nach dem Schlüssel *hash*-partitioniert ist und der Index

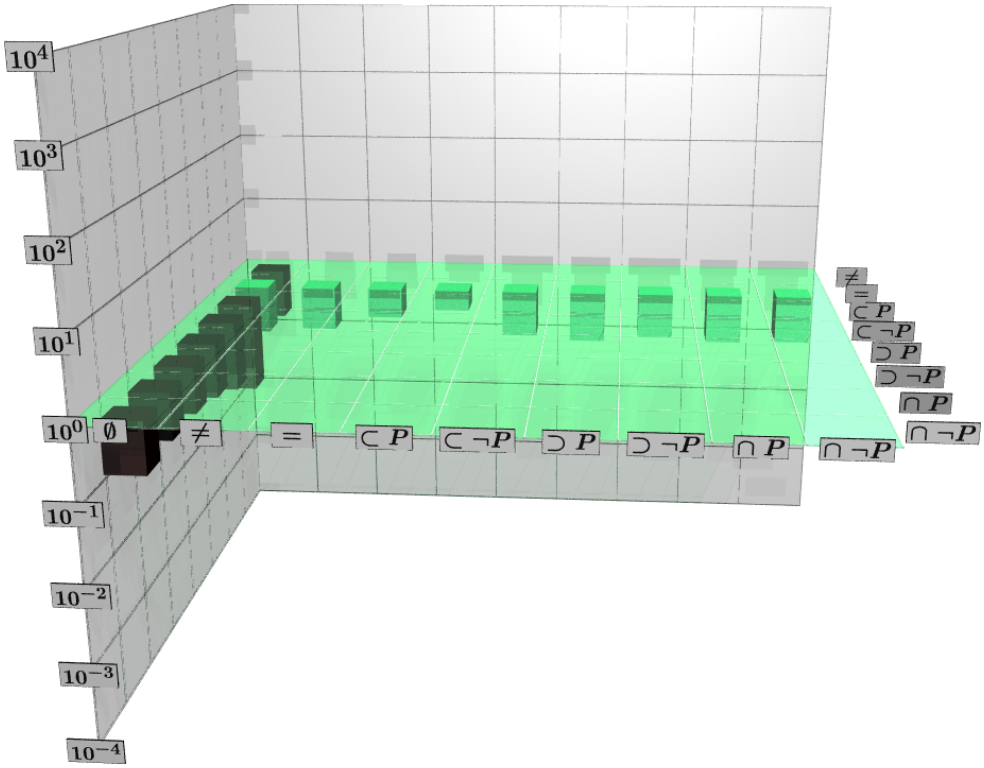

Abbildung 6.2: B*-Baum Index, 2 Attribute, HH Partitionierung, Bereichsanfrage

nicht partitioniert ist. In diesem Fall wird allerdings der Index *nicht* benutzt, wie Tabelle A.1 auf Seite 63 zeigt. Dem DBMS gelingt es anscheinend sehr effizient, anhand des Schlüssels die richtige Tabellenpartition zu finden und diese zu durchsuchen. Dem gegenüber müßte der nichtpartitionierte Index geladen werden, was aus Sicht des Optimierers einen höheren Aufwand verursachen würde. Offensichtlich scheint das DBMS die Puffereffekte bei der Kostenabschätzung nicht zu berücksichtigen, da man annehmen kann, daß bei häufiger Wiederholung der Anfrage, wie es bei der Messung der Fall gewesen ist, der Index im Puffer gehalten wird und Indexnutzung signifikant billiger werden würde. Die drei nächstbesten Attributkombinationen sind 22 bis 24. Dort kann anhand des Schlüssels und der *Hash*-Funktion die relevante Indexpartition bestimmt und dort die Position des angefragten Tupels ermittelt werden. Offensichtlich schätzt der Optimierer den Aufwand, einen partitionierten Index zu benutzen, dann gering ein, wenn die Indexpartition im Voraus bestimmt werden kann. In einigen weiteren Fällen kommt es zu einem völlig unnötigen *Full Table Scan*. Wahrscheinlich ist das auf einen Fehler in der verwendeten Oracle8i-Version zurückzuführen, der möglicherweise beim *Query Rewriting* entsteht.[10]

10. Beim *Query Rewriting* ersetzt der Optimierer die Anfrage durch eine äquivalente, aber effizientere Anfrage. Bei Partitionierung wird die Anfrage in eine Reihe von partitionsbezogenen Teilanfragen aufgeteilt. Es ist zu vermuten, daß im Zusammenhang mit diesem Prozeß Fehler bei der Selektivitätsabschätzung und Kostenbewertung auftreten.

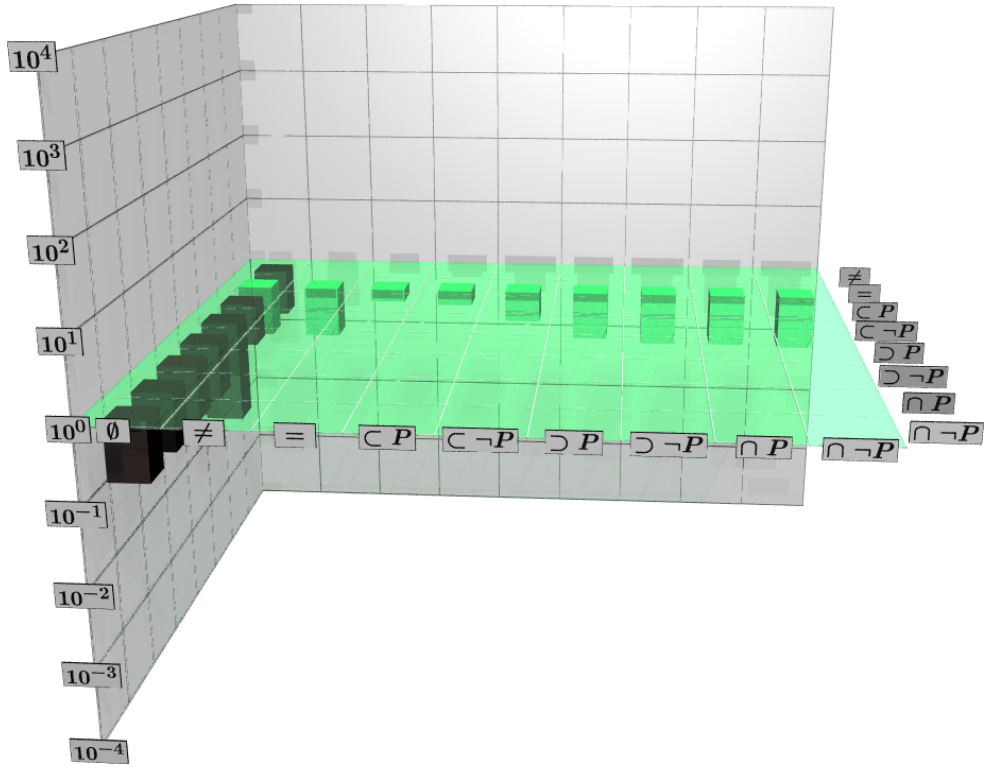

Abbildung 6.3: B*-Baum Index, 2 Attribute, HH Partitionierung, Verbundanfrage

Bei den Bereichs- (Abbildung 6.2 auf der gegenüberliegenden Seite) und Verbundanfragen (Abbildung 6.3) können keine Verbesserungen des Antwortzeitverhaltens gemessen werden. Da die *Hash*-Funktion die Tupel gleichmäßig über alle Partitionen verteilt, ist kein Partitionsausschluß möglich. Stattdessen entsteht ein Mehraufwand durch die Verwendung mehrerer Partitionen, der auch durch den Einsatz von *Intra-Query*-Parallelität nicht aufgewogen werden kann. Die Benutzung eines *Index Scans* anstelle eines *Table Scans* bei den Attributkombinationen 22 bis 24 schlägt sich nur bei der Verbundanfrage in den Meßergebnissen nieder, da dort die Verknüpfung mit der zweiten Tabelle bereits mit den Daten, die aus dem Index gelesen werden, vollzogen werden kann. Da in allen Fällen dennoch ein Tabellenzugriff durchgeführt werden muß, bleiben alle Antwortzeiten über denen der Referenzmessung ohne Partitionierung. An dieser Stelle wären sicher Messungen mit größeren Datenmengen, mehr Partitionen und einem höheren Parallelitätsgrad aufschlußreich.

6.1.2 B*-Baum Indexe über ein Attribut

Die Messung mit Indexen und Anfragen über einem Schlüssel, der nur aus einem Attribut besteht, unterscheidet sich von der vorhergehenden in zwei Weisen. Erstens gibt es in diesem Fall nicht so viele Attributkombinationen, da bei Schlüsseln aus einem Attribut echte Teil- und Schnittmengenbeziehungen nicht möglich sind. Demnach bleiben von der Matrix der Attributbeziehungen (Abbildung 3.1 auf Seite 17 und Tabelle 5.1 auf Seite 27) nur die Spalten 0 bis 2 und die Zeilen 1 und 2 übrig.

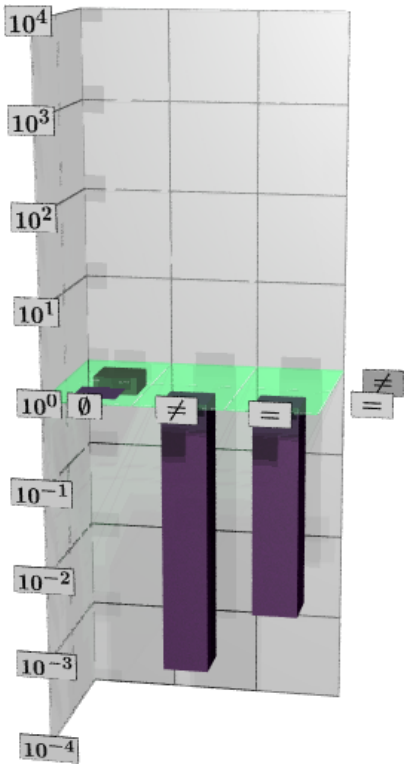

Abbildung 6.4: B*-Baum Index, 1 Attribut, HH Partitionierung, Punktanfrage

Die Meßergebnisse sollten dem entsprechenden Ausschnitt der Messung mit Schlüsseln aus mehreren Attributen (6.1.1) entsprechen. Zweitens ergibt sich ein wichtiger Unterschied jedoch daraus, daß Schlüssel, die aus einem Attribut bestehen, bei dem zugrundeliegenden Schema nicht eindeutig sind, da der Primärschlüssel der Tabelle `lineitem` aus zwei Attributen besteht. Es kommt zu einer Verschlechterung der Selektivität bei allen Anfragen.

Die Meßergebnisse der Punktanfragen entsprechen im Prinzip den Erwartungen, zumindest im Fall von nichtpartitionierten Indexen. (Abbildung 6.4, Tabelle A.4 auf Seite 66). In den Spalten 1 und 2, in denen der Index partitioniert ist, scheint der Oracle8i-Optimierer für partitionierte Tabellen und Indexe eine Regel starr zu befolgen, die besagt, daß nichteindeutige Indexe in diesem Fall *grundsätzlich* nicht zu benutzen sind. Daher wird bei der Attributkombination 22 (Tabelle nach Indexschlüssel und damit auch nach dem in der Anfrage verwendeten Attribut partitioniert) ein *Table Partition Scan* und in der Kombination 21 (Tabelle nicht nach Indexschlüssel partitioniert) sogar ein *Table Full Scan* durchgeführt. Diese Anfrageausführungsstrategie scheint regelbasiert zu sein, da das Kostenargument keinesfalls dafür spricht. Im ungünstigsten Fall haben 7 Tupel der Tabelle den gleichen Wert für das als Indexschlüssel benutzte Attribut `orderkey`, so daß bei der Punktanfrage eine Selektivität von bis zu 7/6 000 000 auftreten kann. Diese Selektivität ist immer noch klein genug, um eine Indexnutzung zu rechtfertigen. Immerhin ist die Ausführung der Referenzan-

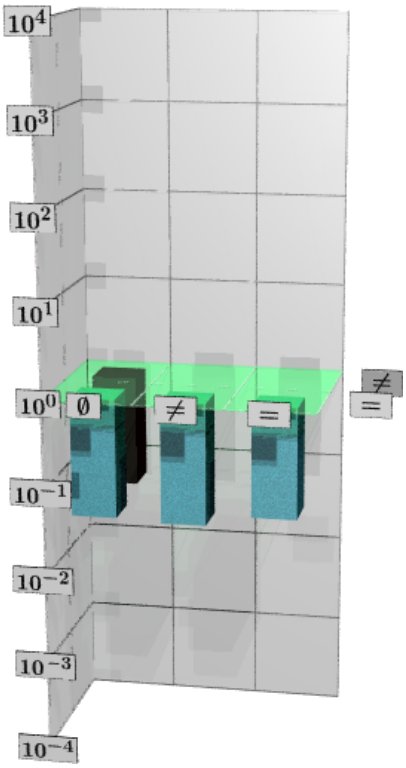

Abbildung 6.5: B*-Baum Index, 1 Attribut, HH Partitionierung, Bereichsanfrage

frage im nichtpartitionierten Fall, bei der generell der Index benutzt wird, um bis zu 1 000 mal schneller.

Bei den Bereichs- (Abbildung 6.5) und Verbundanfragen (Abbildung 6.6 auf der nächsten Seite) ergibt sich ein ähnliches Bild. In allen Fällen entscheidet sich der Optimierer aufgrund der Unmöglichkeit von Partitionsausschluß und der starren Optimiererregel für einen *Full Table Scan*, was sich in entsprechend schlechten Leistungswerten niederschlägt. Es scheint, daß die regelbasierte Optimierung hier für den Fall, daß viele kleine Partitionen existieren, gemacht worden ist. Bei der im Rahmen dieser Arbeit verwendeten Konstellation von sieben relativ großen Partitionen versagt die Optimiererregel daher. Ein kostenbasiertes Verhalten des Optimierers konnte trotz des Vorhandenseins aller notwendigen Statistiken und Histogramme ohne den Einsatz von *Hints* (siehe 5.3.2) nicht erreicht werden.

6.1.3 Bitmap Indexe

Bei der Verwendung von Bitmap Indexen ergibt sich eine leicht veränderte Ausgangslage. Im Gegensatz zu B*-Baum Indexen, die als Baumstruktur mit hohem Verzweigungsgrad aufgebaut sind und bei denen die Partitionierung in der Regel nur eine Verringerung der Baumhöhe um wenige (bzw. eine) Ebenen bewirkt, sind Bitmap Indexe als Listen gespeichert. Da durch den Einsatz von Partitionierung die zu durchsuchende Datenmenge signifikant verringert wird (sofern Partitionsausschluß möglich ist), kann eine Verbesserung des Antwortzeitverhaltens durch die Partitionierung von Bit-

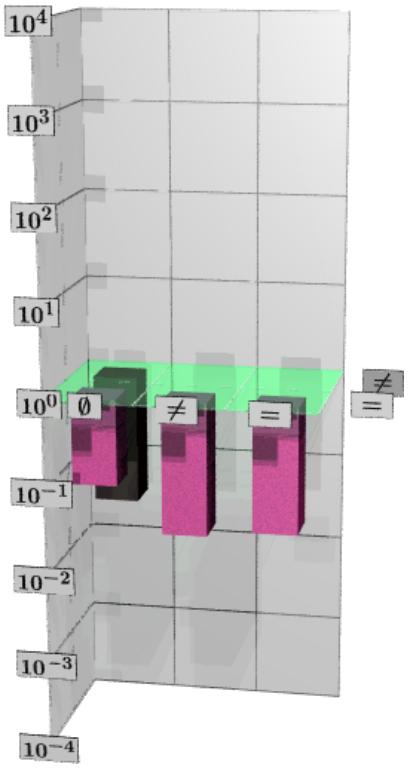

Abbildung 6.6: B*-Baum Index, 1 Attribut, HH Partitionierung, Verbundanfrage

map Indexen erreicht werden. Die kritische Frage ist, ob Partitionsausschluß aufgrund des Verhältnisses zwischen Indexschlüssel, Indexpartitionierung, Tabellenpartitionierung und Anfrage möglich ist oder nicht.

Entsprechend der Erwartungen fallen die Ergebnisse der Punktanfragen (Abbildung 6.7 auf der gegenüberliegenden Seite, Tabelle A.7 auf Seite 67) bei den Attributkombinationen 22 bis 24, bei denen die Partition direkt durch die Anwendung der *Hash*-Funktion auf den Schlüsselwert bestimmt werden kann, günstig aus. Der Zugriff über den partitionierten Bitmap Index bewirkt hier eine signifikante Verbesserung des Antwortzeitverhaltens. In den anderen Fällen (Attributkombinationen 21 und 25 bis 28) kann die Indexpartition nicht bereits aus der Anfrage bestimmt werden und deshalb müssen alle Partitionen durchsucht werden. Das bedeutet, da die Indexpartitionierung in diesem Fall einen Mehraufwand verursacht, eine leichte Leistungsverschlechterung.

Bei den Bereichsanfragen (Abbildung 6.8 auf der gegenüberliegenden Seite) kommen diese Vorteile nicht mehr zum Tragen, da aufgrund der *Hash*-Partitionierung auf alle Partitionen zugegriffen werden muß. Die Interpretation der Meßergebnisse bei den Attributkombinationen 22 bis 24 ist schwierig, da das DBMS dort intern Listenoperatoren verwendet, über die keine konkrete Information zu erhalten ist. In den anderen Fällen ist weder eine Verbesserung noch eine Verschlechterung eingetreten, kleine Abweichungen nach oben oder unten liegen noch im Rahmen der Meßungenauigkeit.

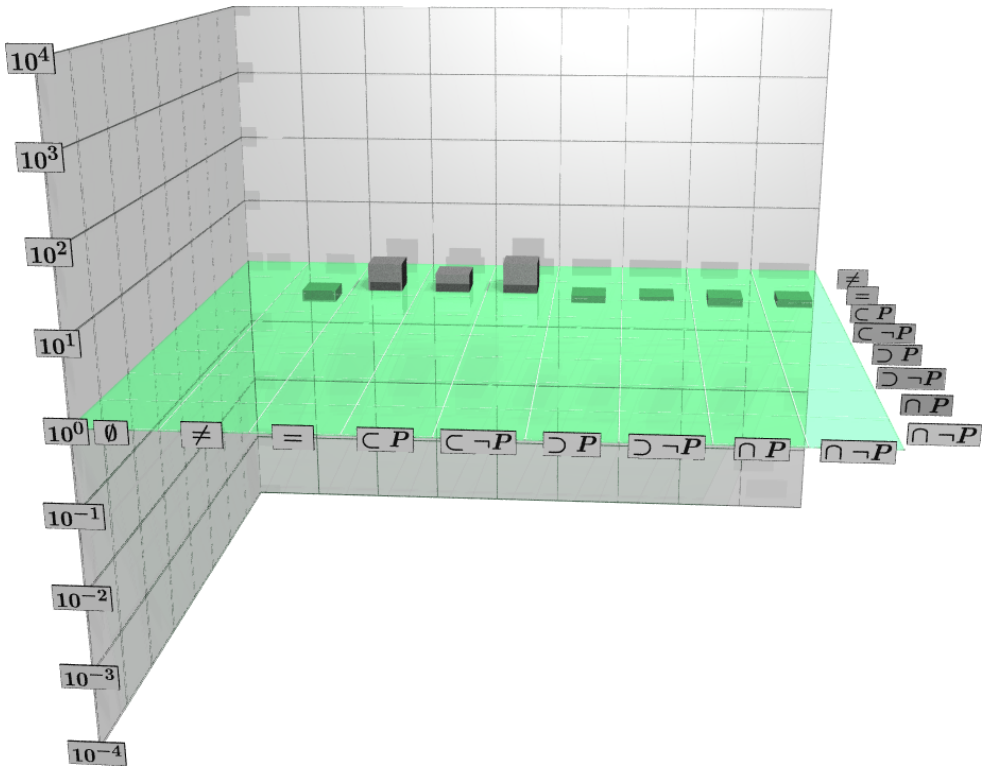

Abbildung 6.7: Bitmap Index, 2 Attribute, HH Partitionierung, Punktanfrage

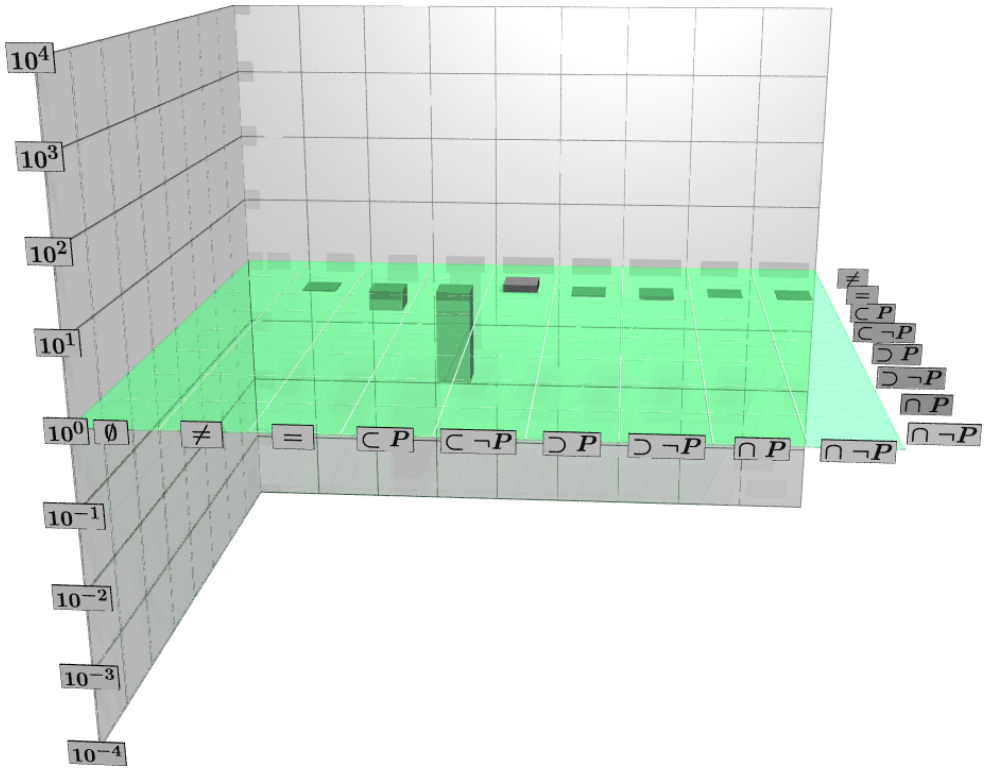

Abbildung 6.8: Bitmap Index, 2 Attribute, HH Partitionierung, Bereichsanfrage

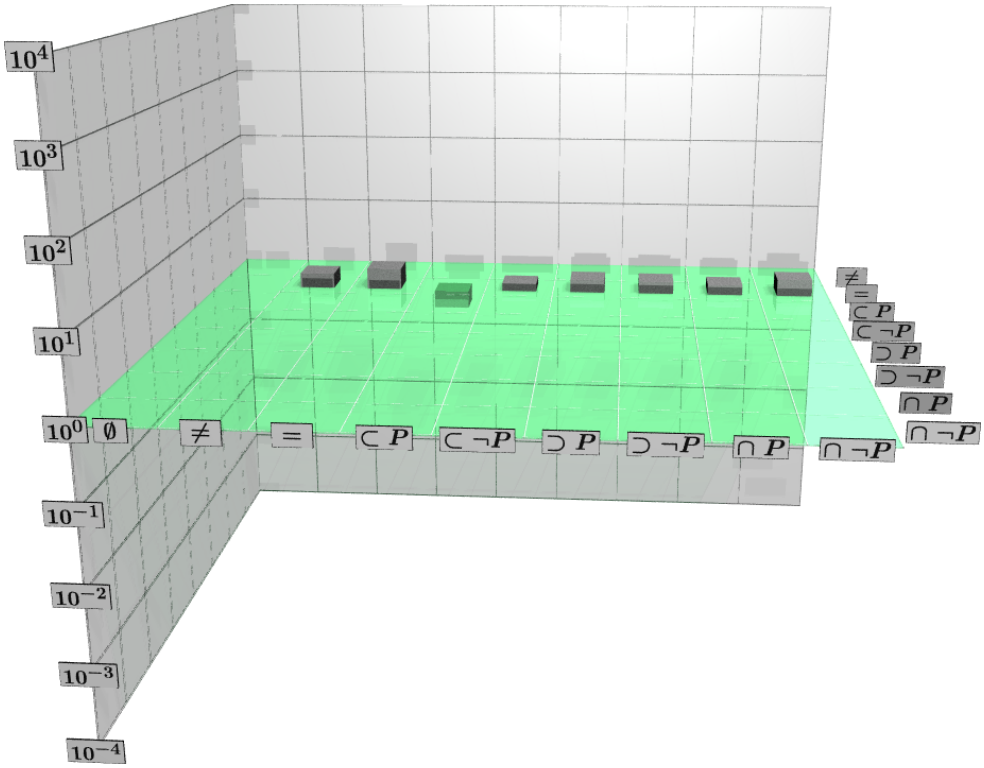

Abbildung 6.9: Bitmap Index, 2 Attribute, HH Partitionierung, Verbundanfrage

Die Verbundanfragen (Abbildung 6.9) können von der *Hash*-Partitionierung der Bitmap Indexe hingegen wieder profitieren. Das entspricht auch den Erwartungen, da die *Intra-Query*-Parallelität bei Verbünden gut ausgenutzt werden kann, indem ein partitionsweiser *Hash Join* durchgeführt wird. Warum bei der Attributkombination 23, bei der das Partitionierungsattribut von Tabelle und Index nur `orderkey` ist, eine Verschlechterung eintritt, ist nicht klar. Bei der Attributkombination 24, bei der Tabelle und Index nur nach dem Attribut `linenumber` partitioniert sind, entscheidet sich der Optimierer aufgrund der schlechten Verteilungseigenschaft der *Hash*-Funktion bei Schlüsseln mit kleinem Wertebereich (siehe 4.1.5) gegen die Indexnutzung, und das offenbar ohne Leistungseinbußen, die in vielen ähnlichen Fällen auftreten. Es ist anzunehmen, daß der Wert bei der Attributkombination 24 ungefähr die Größenordnung der hypothetischen Antwortzeiten für nichtpartitionierte Bitmap Indexe angibt, deren Werte durch Beschränkungen von Oracle8i (siehe 4.1.4) nicht gemessen werden konnten.

6.2 Tabelle: *Hash*, Index: *Range* (HR)

In der zweiten Messung sind Tabelle und Index nach unterschiedlichen Arten partitioniert: die Tabelle ist *hash*-partitioniert, der Index *range*-partitioniert. Dadurch, daß hier Tabelle und Index immer unterschiedlich partitioniert sind, ist lokale Partitionierung nicht möglich. Die Partitionierung des Indexes ist lokalitätserhaltend, d.h. die

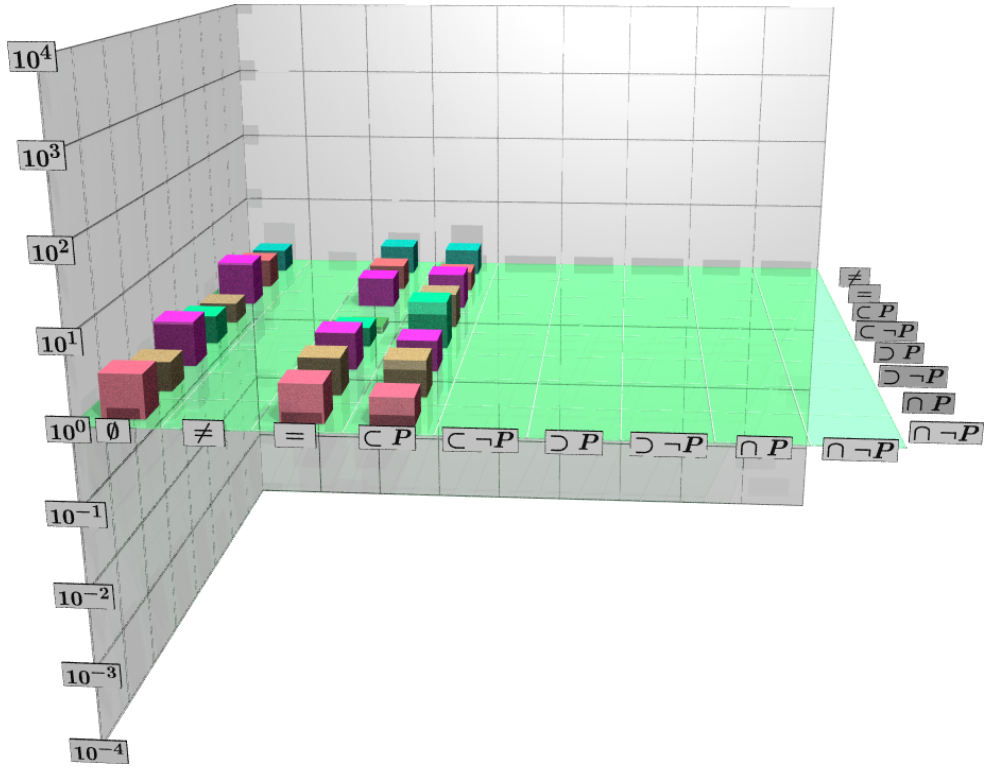

Abbildung 6.10: B*-Baum Index, 2 Attribute, HR Partitionierung, Punktanfrage

angefragten Bereiche liegen in der Regel auf derselben Indexpartition und zusätzlich zu den Punktanfragen kann auch bei den Bereichsanfragen ein Partitionsausschluß durchgeführt werden. Auf der *hash*-partitionierten Tabelle bedeutet ein Bereichszugriff jedoch, da keine lokale Partitionierung vorliegt, einen Zugriff auf alle Partitionen. Theoretisch ist bei allen Anfragen eine Leistungsverbesserung zu erwarten. Bei Punktanfragen kann sowohl beim Index als auch bei der Tabelle ein Partitionsausschluß durchgeführt werden. Bei Bereichs- und Verbundanfragen kann schneller im *range*-partitionierten Index gesucht werden, da die Indexpartitionen klein sind. Bei Verbundanfragen sind allerdings keine großen Leistungssteigerungen zu erwarten, da aufgrund der *Range*-Partitionierung alle angefragten Tupel in derselben Indexpartition liegen und demnach beim Indexzugriff keine *Intra-Query*-Parallelität ausgenutzt werden kann.

Die Kombination aus *hash*-partitionierter Tabelle und *range*-partitioniertem Index wäre auch bei Bitmap Indexen sehr interessant, da Bitmap Indexe stärker als B*-Baum Indexe von der Indexpartitionierung profitieren können. Aufgrund der hier fehlenden lokalen Partitionierung ist das Anlegen von Bitmap Indexen jedoch nicht möglich. Daher sind die Messungen in diesem Abschnitt auf B*-Baum Indexe über einem und zwei Attributen beschränkt.

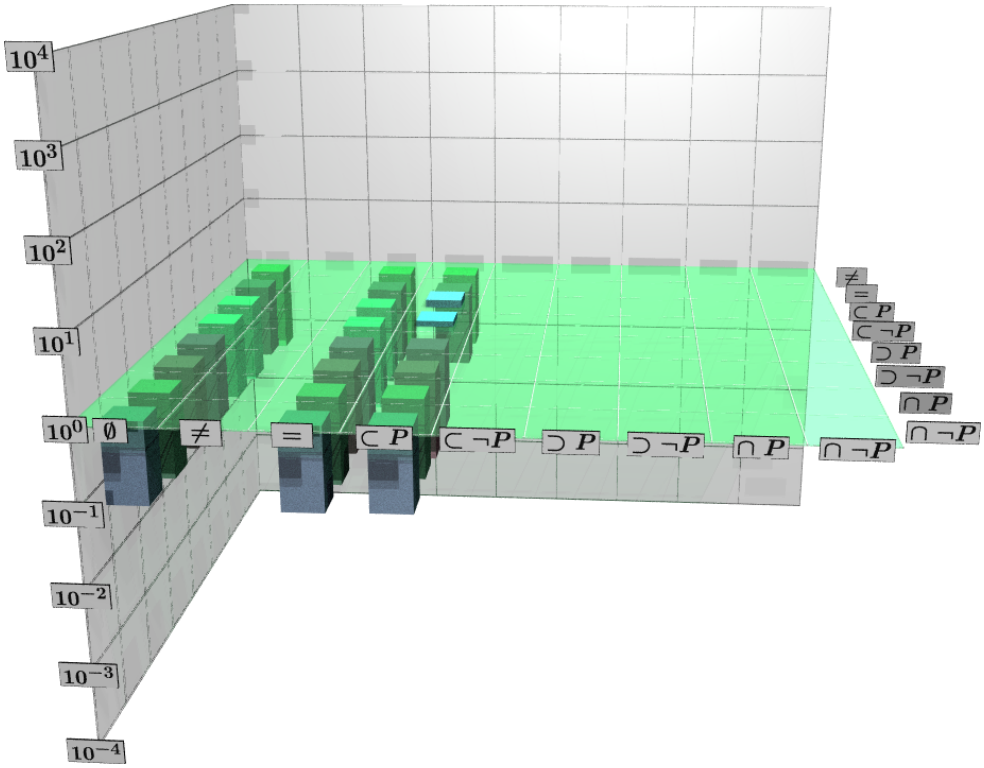

Abbildung 6.11: B*-Baum Index, 2 Attribute, HR Partitionierung, Bereichsanfrage

6.2.1 B*-Baum Indexe über zwei Attribute

Die Punktanfragen (Abbildung 6.10 auf der vorherigen Seite, Tabelle A.10 auf Seite 70) können wie erwartet von der Partitionierung profitieren. In allen Fällen mit partitioniertem Index (Spalten 2 und 3 der Attributkombinationen) nutzt der Optimierer den Index. In der Spalte 0, also bei den Kombinationen mit nichtpartitioniertem Index, scheint der Indexzugriff zu teuer gegenüber einem *Table Scan*. In einigen Fällen (Kombinationen 20, 30 und 40) kann aufgrund der Beziehung zwischen Schlüssel und Partitionierungsattributen ein Partitionsausschluß aufgrund des Schlüsselwertes durchgeführt werden. In den anderen Fällen kann zwar prinzipiell auch Partitionsausschluß genutzt werden, allerdings ist in Tabelle A.10 nur der Listenoperator des DBMS zu erkennen. Die Meßwerte zeigen jedoch an, daß auch in diesem Fall Partitionen ausgeschlossen worden sein müssen. Allerdings wird aus den Meßdaten auch deutlich, daß die Partitionierung des Indexes gegenüber den Anfragen mit nichtpartitioniertem Index keinen Gewinn darstellt. Im Gegenteil, die Anfrageausführung mit partitionierten Indexen ist in vielen Fällen langsamer als ein *Table Partition Scan*. Es kann davon ausgegangen werden, daß insbesondere eine Partitionierung mit vielen kleinen Partitionen und funktionierendem Partitionsausschluß als Ersatz für Indizierung benutzt werden kann. Jedoch müßten die genauen Umstände und Kriterien die ein solches Vorgehen sinnvoll machen, in weiterführenden Arbeiten genau untersucht werden.

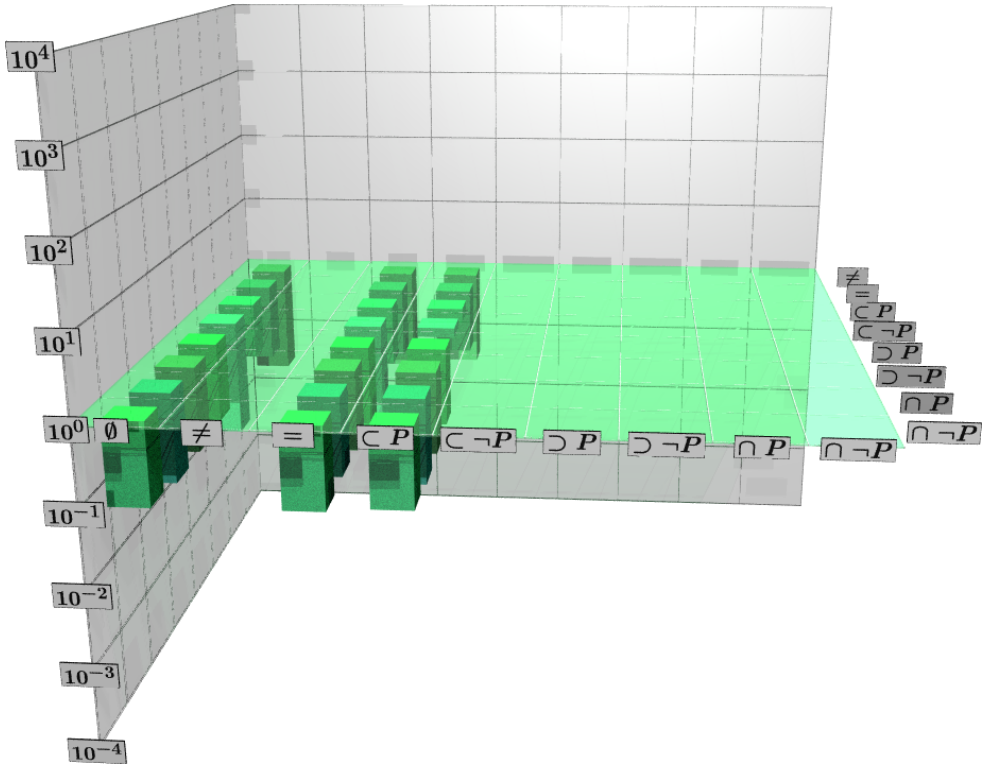

Abbildung 6.12: B*-Baum Index, 2 Attribute, HR Partitionierung, Verbundanfrage

Bei den Bereichs- (Abbildung 6.11 auf der gegenüberliegenden Seite) und Verbundanfragen (Abbildung 6.12) sehen die Meßwerte deutlich ungünstiger aus. In fast allen Fällen nutzt der Optimierer den Index nicht und weicht stattdessen auf den *Full Table Scan* aus. Das ergibt gegenüber der Referenzanfrage, bei der der Index benutzt wird, eine Leistungsverschlechterung ca. um den Faktor 10. Es ist unklar, warum der Optimierer hier den Index nicht benutzt. Eine wichtige Ausnahme sind die Attributkombinationen 33 und 34. Bei diesen beiden Kombinationen ergibt es sich, daß die Tabelle nichtpartitioniert und nur der Index partitioniert ist (vgl. Tabelle 5.1 auf Seite 27). Weiterhin benutzt der Optimierer hier den Index zur Anfragebearbeitung. Das heißt, daß in diesem Fall nachvollzogen werden kann, welchen Einfluß die reine Indexpartitionierung ohne gleichzeitige Tabellenpartitionierung hat. Bei den vorliegenden Meßwerten traten an dieser Stelle Antwortzeitverbesserungen von ca. 20% auf, sowohl bei der Bereichs- als auch bei der Verbundanfrage. Natürlich ist dieser Wert theoretisch, da Indexpartitionierung ohne Tabellenpartitionierung keinen praktischen Sinn hat.

6.2.2 B*-Baum Indexe über ein Attribut

Wenn die Indexe und Anfragen nur einen Schlüssel aus einem (nichteindeutigen) Attribut benutzen, sollten die Ergebnisse theoretisch den vorhergehenden entsprechen. Allerdings treten auch bei dieser Messung die bekannten Effekte auf: Der Optimierer läßt den Index grundsätzlich außer Acht, obwohl die Selektivität der Anfrage durchaus

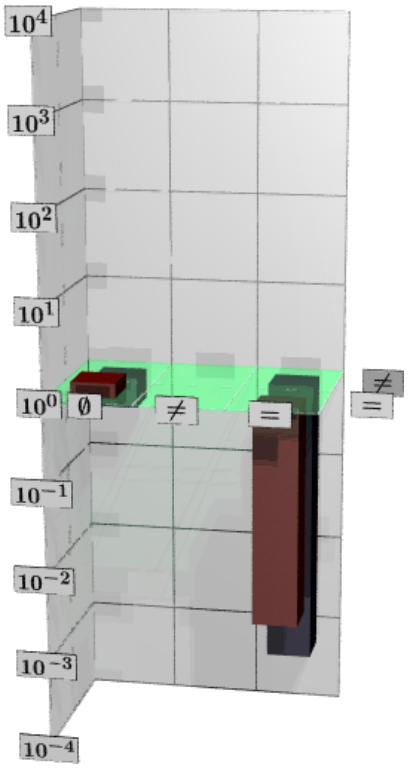

Abbildung 6.13: B*-Baum Index, 1 Attribut, HR Partitionierung, Punktanfrage

Indexnutzung rechtfertigen würde. Bei den Punktanfragen (Abbildung 6.13, Tabelle A.13 auf Seite 73) fällt weiterhin auf, daß obwohl der Index nicht benutzt wird, zwischen den Attributkombinationen ohne Indexpartitionierung (10 und 20) und denen mit Indexpartitionierung (12 und 22) ein sehr großer Unterschied besteht. Dies ist aus den Ausführungsprotokollen nicht erklärbar; möglicherweise liegt ein Fehler beim *Query Rewriting* des verwendeten Oracle8i vor.

Die Bereichs- (Abbildung 6.14 auf der gegenüberliegenden Seite) und Verbundanfragen (Abbildung 6.15 auf der gegenüberliegenden Seite) weisen in abgeschwächter Form dieselbe Anomalie auf. Abgeschwächt dadurch, daß aufgrund der *Hash*-Partitionierung der Tabelle beim Bereichszugriff kein Partitionsausschluß durchgeführt werden kann und sich durch den dadurch bedingten *Table Scan* sowieso schlechtere Antwortzeiten ergeben. Die Verbundanfragen können von der partitionierten Tabelle profitieren, da der *Hash Join* partitionsweise ablaufen kann. Insgesamt bleiben aber alle Ergebnisse hinter denen der Referenzmessung im komplett nichtpartitionierten Fall weit zurück, selbst bei partitionierter Tabelle und nichtpartitioniertem Index.

6.3 Tabelle: *Range*, Index: *Range* (RR)

In der dritten Messung, in der sowohl Tabelle als auch Index *range*-partitioniert sind, sind viele Attributkombinationen möglich, was die Interpretation der Ergebnisse kom-

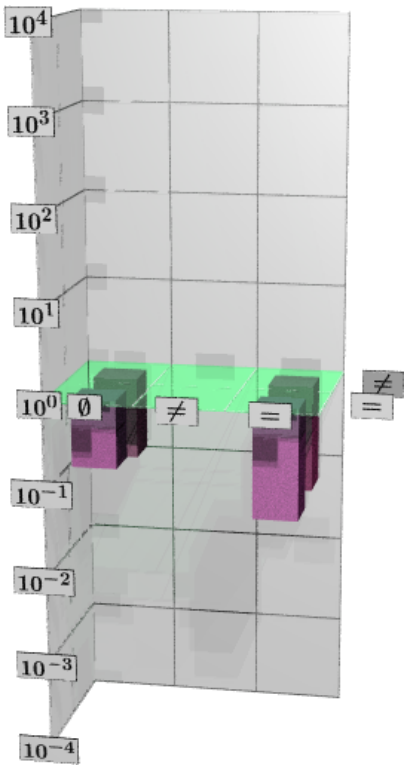

Abbildung 6.14: B*-Baum Index, 1 Attribut, HR Partitionierung, Bereichsanfrage

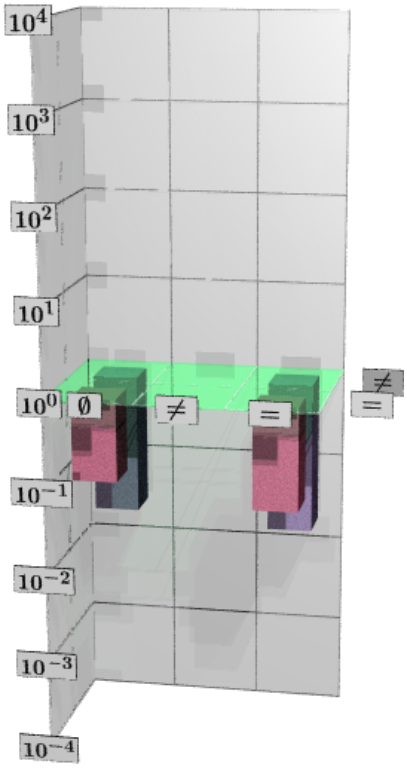

Abbildung 6.15: B*-Baum Index, 1 Attribut, HR Partitionierung, Verbundanfrage

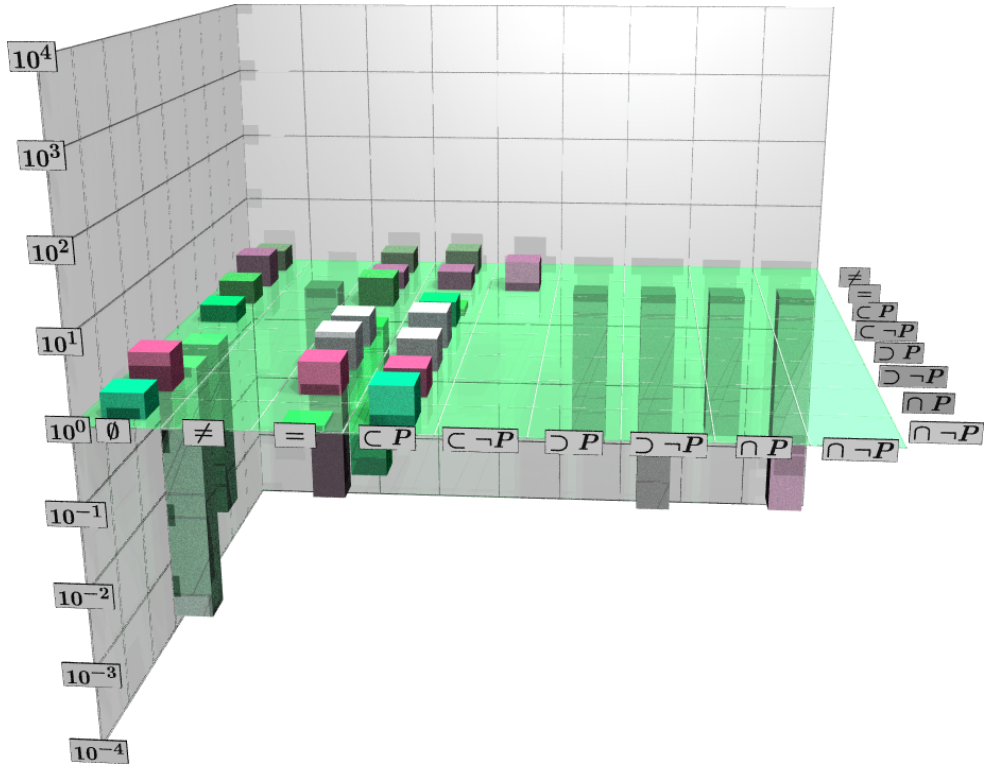

Abbildung 6.16: B*-Baum Index, 2 Attribute, RR Partitionierung, Punktanfrage

plizierter macht. Diese Partitionierungsart ist bei allen Anfragen sowohl beim Index als auch bei der Tabelle lokalitätserhaltend. Das heißt, daß eine Reduzierung der Datenmenge immer stattfinden kann. Dagegen ist nicht zu erwarten, daß Leistungsverbesserungen durch *Intra-Query*-Parallelität auftreten, denn die gesuchten Tupel liegen fast immer in einer Partition. Aus demselben Grund sind auch spezielle Ausführungsstrategien für die Verbundanfrage, wie z. B. der partitionsweise *Hash Join*, ausgeschlossen.

6.3.1 B*-Baum Indexe über zwei Attribute

Bei den Punktanfragen (Abbildung 6.16, Tabelle A.16 auf Seite 74) können wie erwartet in vielen Fällen Antwortzeitverbesserungen erzielt werden. Bei nichtpartitionierten Indexen (Spalte 0 der Attributkombinationen) kann das DBMS in fast allen Fällen durch Partitionsausschluß eine Leistungsverbesserung gegenüber der Referenzmessung erzielen. Die größte Verbesserung von ca. Faktor 3 tritt bei der Kombination 20 auf (Attributkombinationen siehe Abbildung 3.1 auf Seite 17 und Tabelle 5.1 auf Seite 27). Bei den Kombinationen 50 und 60, wo offenbar zu viele Partitionierungsattribute im Spiel sind (`orderkey`, `linenumber` und `partkey`), kann der Optimierer die zu durchsuchenden Partitionen nicht mehr eingrenzen und weicht teilweise auf *Full Table Scan* aus. In diesem Fall könnte der Indexzugriff als eine deutlich bessere Ausweichstrategie benutzt werden. Ähnliches gilt für die lokalen Indexpartitionierungen (Kombinationen 21 bis 28). Hier wird in den drei Fällen 22 bis 24, in denen die

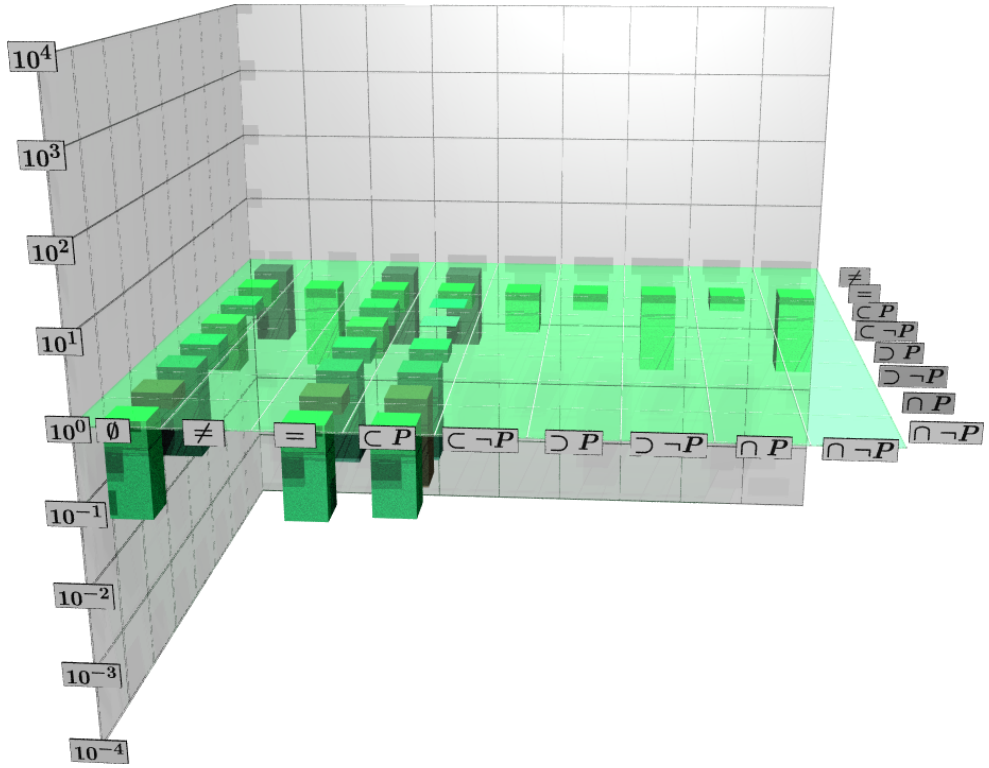

Abbildung 6.17: B*-Baum Index, 2 Attribute, RR Partitionierung, Bereichsanfrage

Partitionierungsattribute des Indexes einer Teilmenge (echt oder unecht) des Schlüssels entsprechen, der Index benutzt und eine ähnliche Leistungssteigerung wie in der Kombination 20 erzielt. In anderen lokalen Partitionierungen jedoch weicht das DBMS auf *Table Partition Scan* und sogar auf *Full Table Scan* aus. Daher ist bei der Auswahl der Partitionierungsattribute Vorsicht angeraten. Die übrigen globalen Partitionierungen (Spalten 2 und 3 der Attributkombinationen) entsprechen auch den Erwartungen. In fast allen Fällen kann ein Partitionsausschluß durchgeführt werden, d. h. der Optimierer verhält sich robust gegenüber einem wechselhaften Verhältnis zwischen Indexschlüssel, Partitionierungsattributen und Attributen in den Selektionsprädikaten, wie es in der Praxis häufig der Fall ist. Insgesamt kommt aber keine Kombination mit Indexpartitionierung auf einen wesentlich besseren Wert als die beste Kombination ohne Indexpartitionierung.

Das positive Bild der Punktanfragen läßt sich nicht auf die Bereichs- (Abbildung 6.17) und Verbundanfragen (Abbildung 6.18 auf der nächsten Seite) übertragen. Hier nutzt das DBMS den Index nur unzureichend und kann oftmals auch keinen Partitionsausschluß nutzen. Generell bringen die Attributkombinationen, bei denen die Partitionierungsattribute relativ gut mit dem Indexschlüssel bzw. den Attributen im Selektionsprädikat übereinstimmen, eine noch relativ gemäßigte Verschlechterung. In den anderen Fällen entscheidet sich der Oracle8i-Optimierer oftmals für *Full Table Scans*, mit entsprechend negativen Leistungsauswirkungen. Weiterhin machen sich

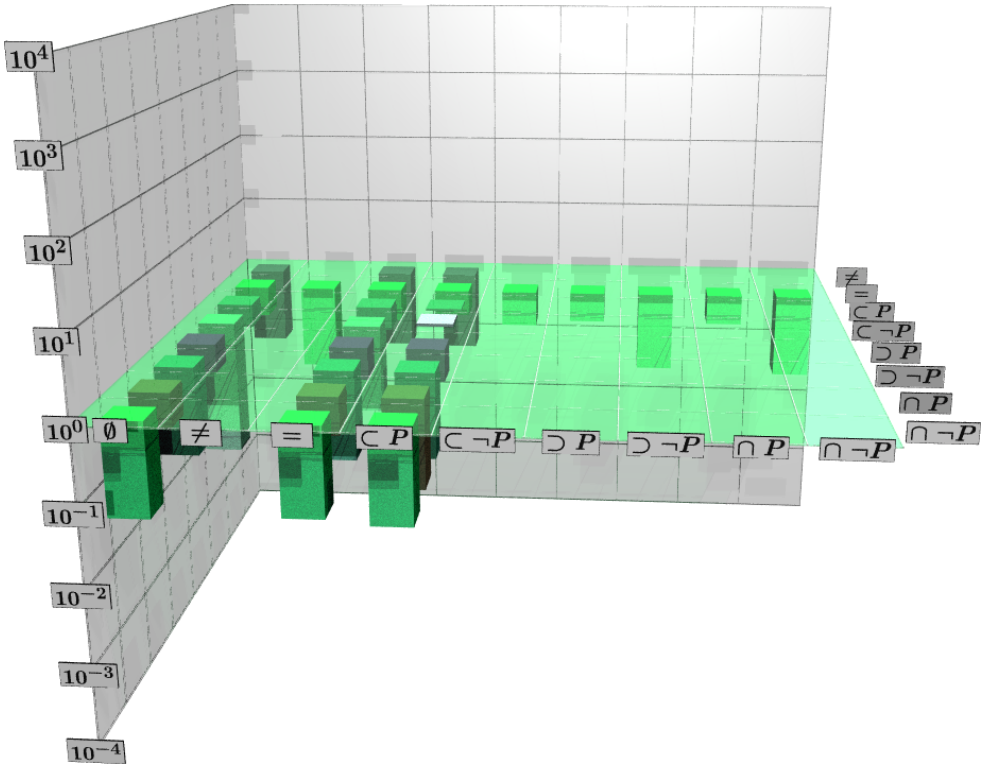

Abbildung 6.18: B*-Baum Index, 2 Attribute, RR Partitionierung, Verbundanfrage

fehlende *Intra-Query*-Parallelität und partitionsweiser Verbund aufgrund der Lokalitätserhaltung bei *Range*-Partitionierung bemerkbar.

6.3.2 B*-Baum Indexe über ein Attribut

Werden Indexschlüssel, Partitionierungsattribut und Selektionsprädikat jeweils auf ein Attribut beschränkt, ergibt sich das von den anderen Messungen mit Indexen über einem Attribut gewohnte Bild. Bei der Punktanfrage (Abbildung 6.19 auf der gegenüberliegenden Seite, Tabelle A.19 auf Seite 77) wird der Index vom DBMS völlig außer Acht gelassen, mit entsprechenden Leistungseinbußen. Weiterhin ist hier unklar, warum sich die Werte in den Attributkombinationen 20 und 22 so stark voneinander unterscheiden. Da kein Index benutzt wird und die Tabellenpartitionierung gleich ist, sollten auch gleiche Werte gemessen werden. Möglicherweise handelt es sich um einen Fehler beim *Query Rewriting*.

Dieser Fehler tritt bei den Bereichs- (Abbildung 6.20 auf der gegenüberliegenden Seite) und Verbundanfragen (Abbildung 6.21 auf Seite 54) nicht auf. Dort ergibt sich ein weitgehend symmetrisches Bild, wie es theoretisch zu erwarten ist. Allerdings sind die Meßwerte alle etwas schlechter als bei den anderen beiden Partitionierungsarten mit *hash*-partitionierter Tabelle, da hier keine *Intra-Query*-Parallelität und kein partitionsweiser Verbund benutzt werden kann.

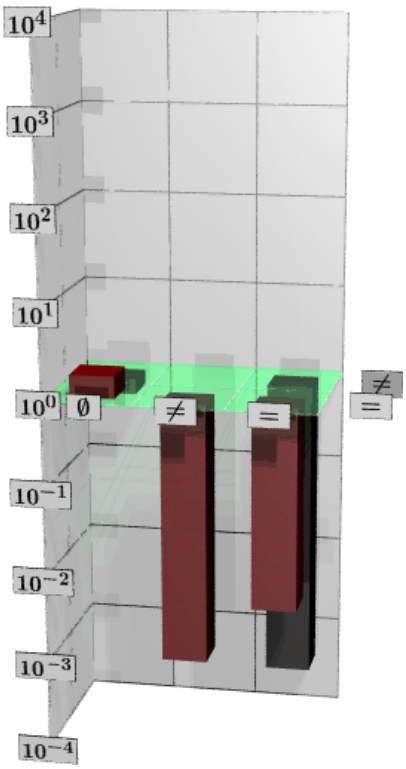

Abbildung 6.19: B*-Baum Index, 1 Attribut, RR Partitionierung, Punktanfrage

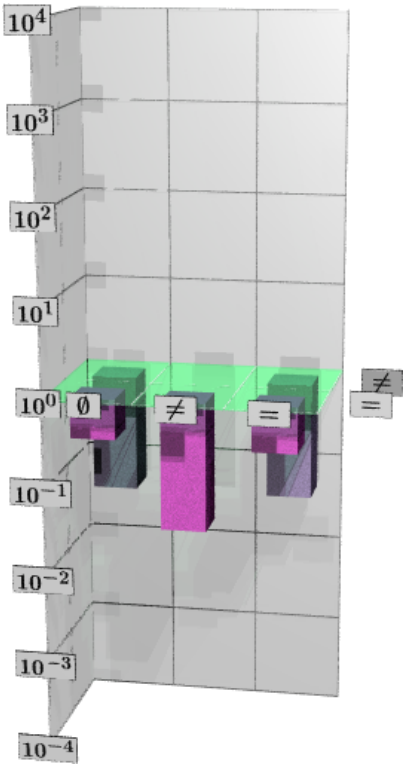

Abbildung 6.20: B*-Baum Index, 1 Attribut, RR Partitionierung, Bereichsanfrage

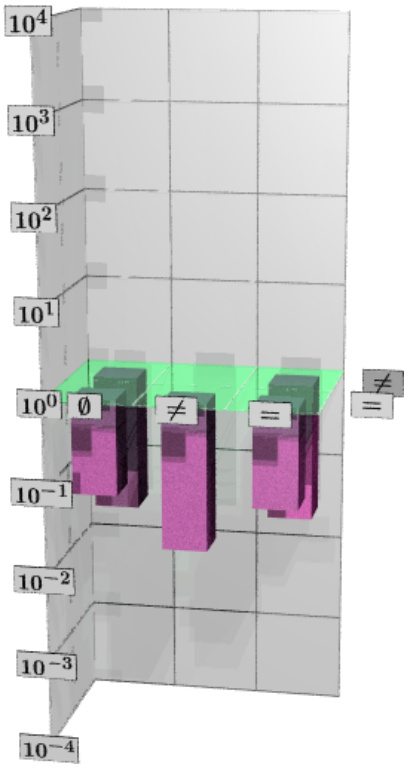

Abbildung 6.21: B*-Baum Index, 1 Attribut, RR Partitionierung, Verbundanfrage

6.3.3 Bitmap Indexe

Bitmap Indexe können von der *Range*-Partitionierung besonders profitieren, da die Suchzeit in Indexpartitionen hier stärker von der Partitionsgröße abhängig ist als in B*-Baum Indexen (vgl. 6.1.3).

Bei den Punktanfragen (Abbildung 6.22 auf der gegenüberliegenden Seite, Tabelle A.22 auf Seite 78) kann der Index bei fast allen Attributkombinationen mit Partitionsausschluß genutzt werden. Lediglich bei den Kombinationen 21, 26 und 28, bei denen aufgrund des SQL-Vektorvergleichs die Partition nicht direkt aus der Anfrage bestimmt werden kann, muß ein *Full Index Scan* durchgeführt werden, der aufgrund des höheren Verwaltungsaufwands langsamer läuft als die Referenzmessung. Dieses Verhalten ist durch die Semantik des SQL-Vektorvergleichs, die besagt, daß das an weitesten links stehende Attribut höchste Priorität genießt, bestimmt und hat bei Bitmap Indexen eigentlich keine sachliche Rechtfertigung, da Bitmap Indexen die hierarchische Baumstruktur fehlt. An dieser Stelle sind demnach in Oracle8i die Annahmen, die für B*-Baum Indexe gemacht worden sind, zu einfach auf Bitmap Indexe übertragen worden. Insbesondere im Hinblick auf die häufige Anwendung von Bitmap Indexen im *Data Warehouse* Bereich, der mit den anfallenden Datenmengen am stärksten von Indexpartitionierung profitieren könnte, muß die Implementierung der Partitionierung von Bitmap Indexen als nicht konsequent beurteilt werden. Ein wirklich mehrdimensionales Partitionierungsverfahren wäre wünschenswert.

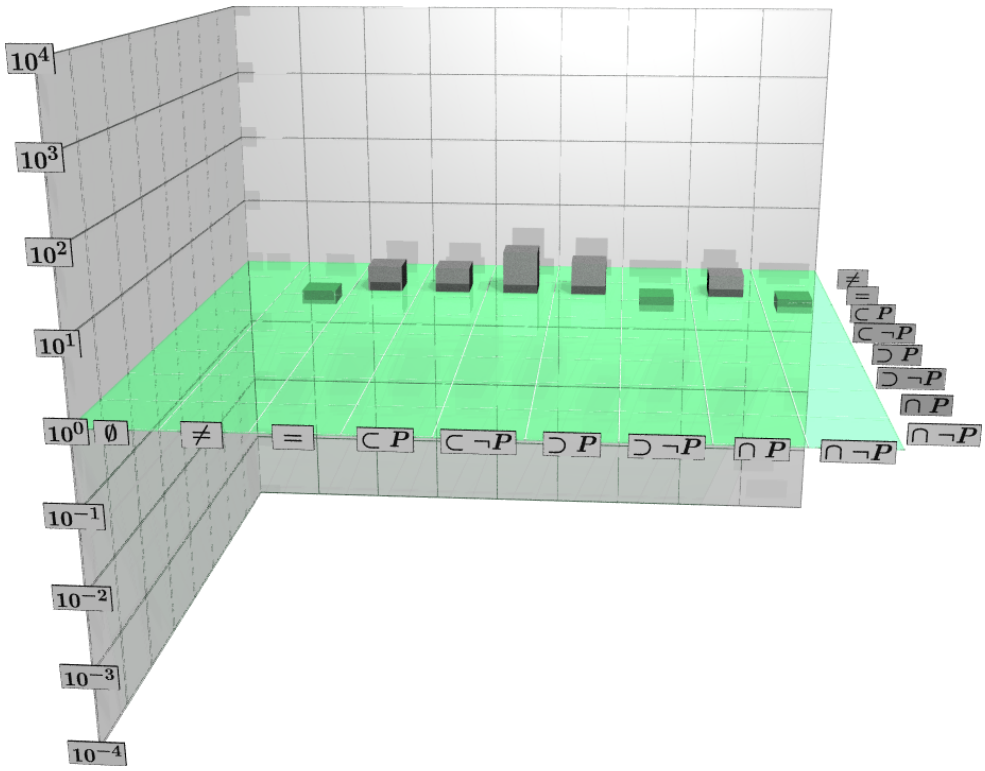

Abbildung 6.22: Bitmap Index, 2 Attribute, RR Partitionierung, Punktanfrage

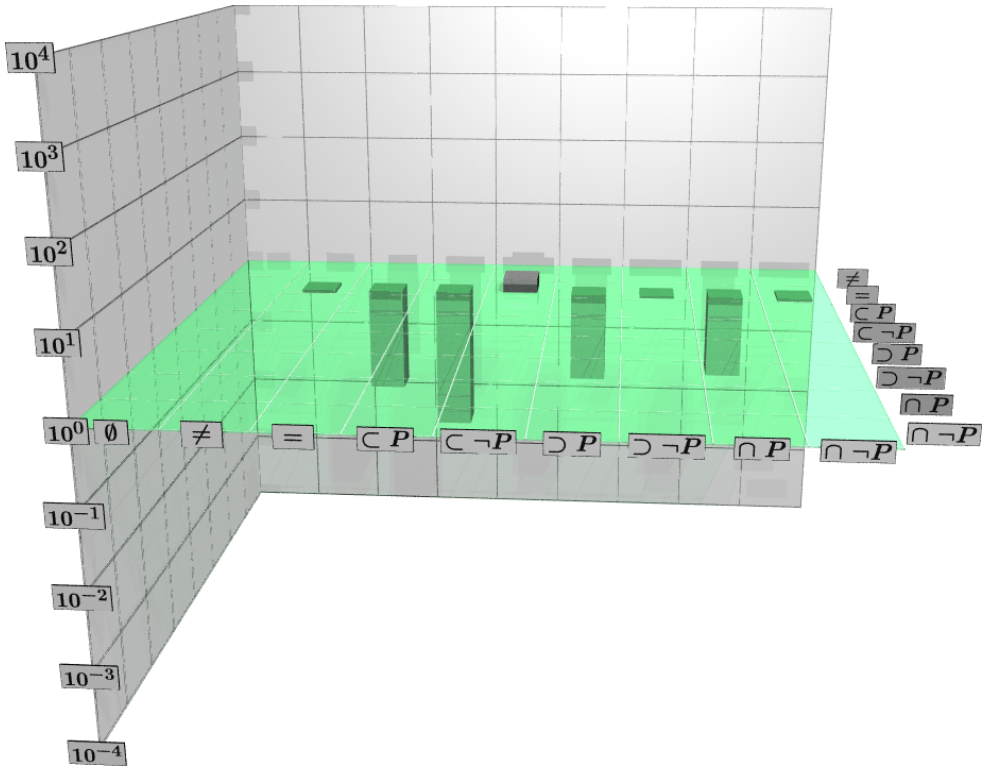

Abbildung 6.23: Bitmap Index, 2 Attribute, RR Partitionierung, Bereichsanfrage

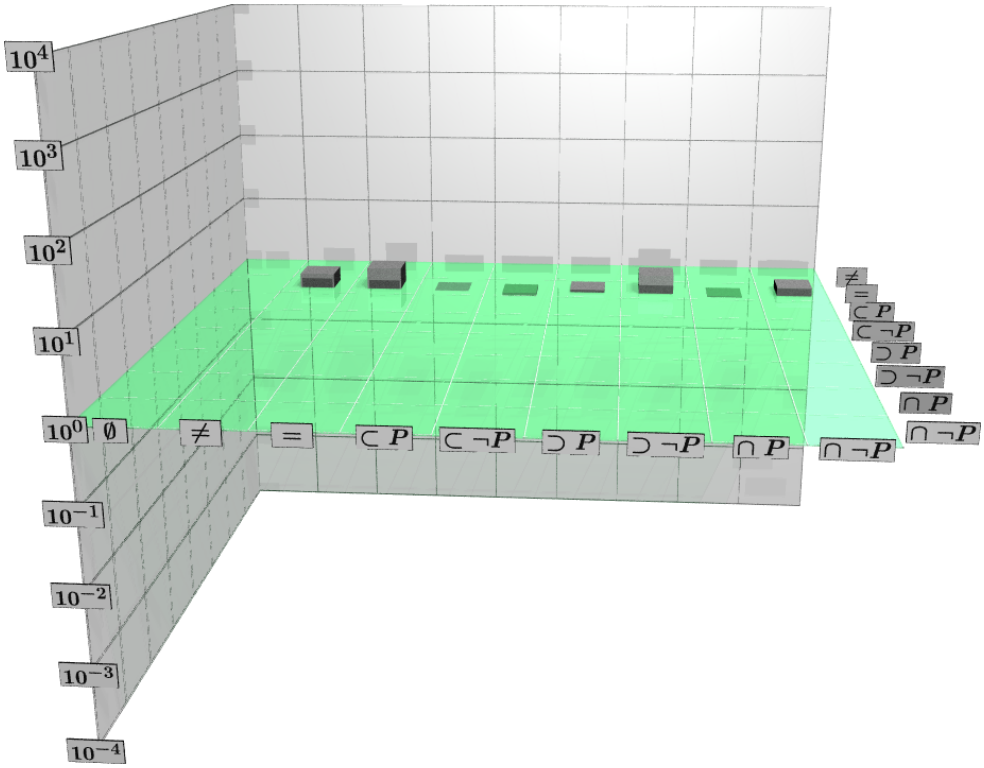

Abbildung 6.24: Bitmap Index, 2 Attribute, RR Partitionierung, Verbundanfrage

Die Meßwerte der Bereichsanfragen (Abbildung 6.23 auf der vorherigen Seite) sind im Rahmen dieser Messung außergewöhnlich. In fast allen Fällen ist in den Protokollen nur ein Hinweis auf den Listenoperator und die Fälle, die bei den Punkt- und Verbundanfragen am problematischsten waren (Kombinationen 21, 26 und 28), ergeben hier noch die besten Ergebnisse mit weder Verbesserung noch Verschlechterung. Möglicherweise wird bei Bereichsanfragen in der untersuchten Oracle8i-Version 8.1.5 ein noch nicht ausgereifter Programmbereich berührt. Dazu paßt, daß es abhängig von der konkreten Formulierung der Bereichsanfrage zu Serverabstürzen kam. Insbesondere mußten Formulierungen wie z. B. `where quantity between 40 and 43` durch äquivalente Ausdrücke wie `where quantity in (40, 41, 42, 43)` ersetzt werden. Diese Probleme traten nur dann auf, wenn im Selektionsprädikat mehr als ein Attribut eingeschränkt wurde.

Bei den Verbundanfragen (Abbildung 6.24) kommt ein deutlich stabileres Verhalten zum Vorschein. Es tritt gegenüber der Referenzmessung keine wesentliche Verschlechterung und in einigen Fällen sogar eine Verbesserung ein. Da der Optimierer bei den Attributkombinationen 21 bis 23, 25 und 27 den Index benutzt, kann hier der Vorteil des Partitionsausschlusses ausgenutzt und die theoretisch mögliche Verbesserung der Antwortzeit durch Datenreduktion praktisch umgesetzt werden. Allerdings ist zu bemerken, daß der Partitionsausschluß aufgrund der lokalen Partitionierung gleichermaßen auf der Tabelle möglich ist, so daß hier nicht klar ist, wieviel der Verbesserung wirklich auf die Indexpartitionierung zurückgeht, da eine Messung bei

nichtpartitioniertem Index bei der verwendeten Oracle8i-Version nicht möglich ist. Allerdings gibt der Meßwert bei Kombination 24, bei der direkt auf die Tabelle mit Partitionsausschluß zugegriffen wird, einen Hinweis darauf, daß die Indexpartitionierung schon zur Verbesserung der Antwortzeit beiträgt, denn der Partitionsausschluß auf der Tabelle allein bringt keine Verbesserung.

7 Fazit

7.1 Zusammenfassung der Ergebnisse

In dieser Arbeit wurde der Leistungsgewinn beim Einsatz von Indexpartitionierung anhand einer praktischen Messung untersucht. Dabei muß noch einmal darauf hingewiesen werden, daß die ermittelten Ergebnisse stark von der verwendeten Hard- und Software abhängig sind. Eine Übertragung der Messung in andere Umgebungen führt sehr wahrscheinlich zu anderen Ergebnissen. Die mit der benutzten Oracle8i-Version gewonnenen Ergebnisse lassen allerdings einige prinzipielle Zusammenhänge erkennen.

Grundsätzlich läßt sich sagen, daß die Leistungsverbesserung durch Indexpartitionierung der durch Tabellenpartitionierung deutlich nachsteht. Theoretisch ergeben sich Möglichkeiten zur Leistungssteigerung durch Ausnutzung von *Intra-Join*-Parallelität, partitionsweisen Verbundoperationen und vor allem bei Bitmap Indexen durch Beschleunigung der Suchoperationen durch Partitionsausschluß. Praktisch können in vielen Fällen jedoch nur geringe Leistungsverbesserungen gegenüber einer reinen Tabellenpartitionierung ohne Indexpartitionierung gewonnen werden, und das immer nur bei bestimmten Anfragen. Dagegen gibt es sehr viele Fälle, bei denen der Einsatz von Indexpartitionierung beim verwendeten DBMS zu einer Leistungsverschlechterung führt, teilweise um mehrere Größenordnungen. Dies ist im Wesentlichen auf Probleme des Oracle8i-Optimierers im Umgang mit Indexpartitionierung zurückzuführen.

Im Gegensatz zu B*-Baum Indexen, die als problematisch anzusehen sind, können Bitmap Indexe teilweise von Indexpartitionierung profitieren. Das hängt maßgeblich mit dem unterschiedlichen inneren Aufbau von Bitmap Indexen zusammen. Während bei B*-Baum Indexen durch die Partitionierung keine signifikante Veränderung der zu durchsuchenden Baumhöhe erreicht werden kann, wird die zu durchsuchende Datenmenge bei Bitmap Indexen durch Partitionierung wirkungsvoll eingeschränkt. Da in der untersuchten Oracle8i-Version nichtpartitionierte Bitmap Indexe bei partitionierten Tabellen nicht möglich sind, sind partitionierte Bitmap Indexe die einzige Möglichkeit, Bitmap Indexe überhaupt bei partitionierten Tabellen einzusetzen. Da das in der Regel nicht mit Leistungseinbußen verbunden ist (ausgenommen Bereichsanfragen), sollte der Einsatz von partitionierten Bitmap Indexen in Betracht gezogen werden.

Von den verschiedenen Anfragetypen profitiert die Punktanfrage am meisten. Dort kann am ehesten aufgrund der Anfragewerte ein Partitionsausschluß durchgeführt werden, so daß sich die zu betrachtende Datenmenge signifikant verkleinert. Daher ist es bei der Frage, ob Indexpartitionierung benutzt werden soll oder nicht, entscheidend, die Anfragen genau zu analysieren. Unterstützen die Indexe vor allem Punktanfragen, so kann Indexpartitionierung eine Verbesserung bringen. Kommen zusätzlich auch nennenswert Bereichsanfragen vor, so ist hingegen von einem Einsatz der Indexpartitionierung stark abzuraten bzw. die Nutzung eines partitionierten Indexes zu verhindern. Das bei dieser Messung verwendete DBMS zeigte bei Bereichsanfragen

ernsthafte Schwächen in der Anfrageauswertung. Oft wurde nicht nur die Partitionierung nicht ausgenutzt, sondern zusätzlich der ganze Index nicht benutzt und auf die verschiedenen Methoden des *Table Scans* zurückgegriffen. Bei der Messung wurden viele Hinweise auf eine noch nicht konsequent auf Indexpartitionierung ausgerichtete Implementierung des DBMS gefunden. Verbundanfragen, die insbesondere im *Data Warehouse* Umfeld sehr häufig anzutreffen sind, können in einigen Fällen von Indexpartitionierung profitieren, vor allem, wenn es sich um *Hash*-Partitionierung handelt. Das liegt erstens an der gut ausnutzbaren *Intra-Query*-Parallelität und zweitens insbesondere an der Möglichkeit, einen *Hash Join* partitionsweise vorzunehmen, wie es z. B. in [Dad96] beschrieben ist. Allerdings fallen diese beiden Möglichkeiten bei der in dieser Messung verwendeten Verbundanfrage, die nur Teile einer Tabelle mit einer anderen verknüpft, bei *Range*-Partitionierung aus, weil diese Partitionierunsart lokalitätserhaltend ist und sich die Verbundanfrage demnach nur auf eine bzw. wenige Partitionen beschränkt. Davon abgesehen ergibt die Art der Partitionierung, also *Range*- oder *Hash*-Partitionierung, keinen großen Unterschied in der Meßergebnissen. Bei der Frage der möglichen Kombinationen ist der Unterschied zwischen *Hash*- und *Range*-Partitionierung hingegen groß.

Fast allen Messungen gemeinsam ist, daß ernste Probleme des Optimierers in Oracle8i beim Umgang mit Partitionierung aufgefallen sind. In einigen Fällen erscheint das verwendete Kostenmodell zu einfach; in anderen Fällen wird, falls kein Partitionsausschluß möglich ist, auf einen *Table Scan* zurückgegriffen, was eine Verschlechterung der Antwortzeit um mindestens eine Größenordnung gegenüber der nichtpartitionierten Referenzanfrage, bei der ein direkter Indexzugriff oder *Index Scan* benutzt wird, nach sich zieht. Da der durch die Verwaltung von sieben Partitionen verursachte Mehraufwand keinesfalls die Antwortzeiten verzehnfacht, ist daraus zu schließen, daß der Optimierer in solchen Fällen keine optimale Ausführungsstrategie wählt. In bestimmten Einzelfällen, vor allem bei Bereichsanfragen, ergibt sich ein überhaupt nicht nachvollziehbares Verhalten. Möglicherweise liegen hier Fehler bei der Anfragebearbeitung vor. Besondere Vorsicht ist anzuraten, wenn der Indexschlüssel kein Primärschlüssel ist: die Verwendung nichteindeutiger Indexe im Zusammenhang mit Indexpartitionierung ist bei der untersuchten Version völlig ausgeschlossen. Darüber hinaus sind auch Schwächen bei der Verwendung von nichtpartitionierten, nichteindeutigen Indexen zu beobachten.

Beim Einsatz der Indexpartitionierung *ohne* Tabellenpartitionierung tritt bei B*-Baum Indexen eine leichte Leistungsverbesserung von ca. 20% ein. Diese Fragestellung ist zwar in der Praxis irrelevant, da Indexpartitionierung ohne Tabellenpartitionierung keinen Sinn ergibt, ist aber ein Versuch, den Einfluß von Indexpartitionierung isoliert betrachten zu können. Sowohl der im Vergleich zu ausschließlicher Tabellenpartitionierung (z. T. über 50%) geringe Wert als auch die kleinen Unterschiede zwischen den Ergebnissen bei Tabellenpartitionierung mit und ohne Indexpartitionierung legen die Vermutung nahe, daß der Leistungsgewinn der Indexpartitionierung bei B*-Baum Indexen minimal ist. Dafür handelt man sich einige Probleme mit dem Optimierer ein.

Bei Bitmap Indexen ist eine solche Gegenüberstellung nicht möglich. Die Meßwerte legen den Schluß nahe, daß die Indexpartitionierung hier sehr wohl Leistungsverbes-

serungen bringt. Diese Vermutung kann aber aufgrund des fehlenden Vergleichs mit nichtpartitionierten Indexen und Tabellenpartitionierung nicht überprüft werden. Auf jeden Fall ergibt der Einsatz von partitionierten Tabellen und Bitmap Indexen einen deutlichen Leistungsgewinn gegenüber nichtpartitionierten Tabellen und Bitmap Indexen.

Bei den Meßergebnissen kann kein grundlegender Leistungsunterschied zwischen den lokalen und globalen Partitionierungsvarianten festgestellt werden. In beiden Fällen benutzt der Optimierer den Index häufig gar nicht zur Anfragebearbeitung. Falls der Optimierer den Index nutzt, wirken sich die angeblichen Vorteile von lokal partitionierten Indexen (vgl. 4.1.2) bei der Anfragebearbeitung nicht aus. Dennoch kann der Einsatz von lokal partitionierten Indexen aus administrativen Gründen vorteilhaft sein. In diesem Fall ist darauf zu achten, daß die Indexpartitionierung *prefixed* ist, da dort Leistungsgewinne am häufigsten auftreten und Leistungsverluste durch die Partitionierung am unwahrscheinlichsten sind.

Insgesamt ergibt sich also der Schluß, daß Indexpartitionierung im untersuchten System beschränkt einsetzbar ist. In vielen Fällen konnte kein Leistungsgewinn gegenüber einen reinen Tabellenpartitionierung erzielt werden; bei Bitmap Indexen waren Leistungssteigerungen möglich. Allerdings ist das Leistungsverhalten gegenüber verschiedener Anfragetypen als sehr instabil zu bezeichnen. In vielen Fällen waren Leistungsverschlechterungen bis zu mehreren Größenordnungen zu beobachten. Indexpartitionierung kann sinnvoll eingesetzt werden, insbesondere wenn klare Vorteile bei der Administration und Wartung auf der Hand liegen. Allerdings gilt es, die eingesetzten Anfragen und das Verhalten des Optimierers kritisch zu beobachten. Eine Entscheidung unter Leistungsgesichtspunkten bleibt dem Einzelfall vorbehalten und kann nicht pauschal beantwortet werden. Insgesamt ist ein eher abwartendes und zurückhaltendes Vorgehen sinnvoll.

7.2 Ausblick

Im Rahmen dieser Arbeit sind einige Fragestellungen aufgefallen, die unter Umständen Gegenstand weiterführender Arbeiten sein könnten. Zum einen fällt auf, das selbst die in dieser Arbeit verwendete recht grobe Partitionierung als eine Art Indexersatz verwendet werden konnte, da durch Partitionsausschluß und *Table Partition Scan* eine Anfrage ähnlich schnell wie durch normalen Indexzugriff beantwortet werden kann. Diese Eigenschaft der Tabellenpartitionierung könnte bei feingliedriger Unterteilung in Partitionen noch stärker zum Tragen kommen und wäre ein interessanter Untersuchungsgegenstand.

Zum anderen stellt sich die Frage, wie weit wenigstens den gröbsten Fehlentscheidungen des Optimierers in der Praxis durch den gezielten Einsatz von *Hints* vorgebeugt werden könnte. Das Dilemma mit *Hints* ist, daß sie statisch in die Anfragen eingebaut werden und somit die Datenunabhängigkeit der Anfragen beeinträchtigen oder sogar zerstören. Auf der anderen Seite fällt jedoch auf, daß der Optimierer in einigen Fällen auf Indexnutzung ganz verzichtet und sehr teure *Table Scan* Operationen benutzt. Somit stellt sich die Frage, nach welchen Gesichtspunkten und Kriterien

der Einsatz von *Hints* vorgenommen werden sollte, um ein Mindestmaß an Datenunabhängigkeit zu erhalten.

Viele der aufgezeigten Probleme sind Schwächen der konkreten Implementierung des verwendeten DBMS. Ansatzweise zeigten sich Leistungspotentiale bei der Verwendung von Indexpartitionierung, insbesondere bei Bitmap Indexen. Die gerade in Entwicklung befindliche Version Oracle9 soll gerade auf dem Gebiet der Partitionierung starke Überarbeitungen und Verbesserungen erfahren haben. Daher ist anzunehmen, daß einige der aufgezählten Probleme in Zukunft keine Auswirkung mehr zeigen und sich Indexpartitionierung auch im praktischen Einsatz bewährt.

Anhang

A Tabellen

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	2.84/2.49 *?*	—	—	—	—	—	—	—	—
$P_T = P_I$	2.11/0.86 *T1*	3.81/4151 *T7*	2.29/1.28 *I1*	2.44/1.66 *I1*	2.03/1.25 *I1*	3.44/3571 *T7*	3.54/4115 *T7*	3.1/4582 *T7*	2.17/5061 *T7*
$P_T \subset P_I$ (Präfix)	3.59/1.71 *T1*	—	—	—	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	1.89/0.97 *T1*	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	2.71/4002 *T7*	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	2.31/1.1 *?*	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	2.36/1.04 *?*	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	2.28/1.21 *?*	—	—	—	—	—	—	—	—

Tabelle A.1: B*-Baum Index, 2 Attribute, Partitionierungsart HH, Punktanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	178.3/934.5 $T7$	—	—	—	—	—	—	—	—
$P_T = P_I$	189.1/923.4 $T7$	225.1/952.5 $T7$	193.7/520.8 $I7$	226.7/453.7 $I7$	83.81/389.1 $I7$	171.5/920 $T7$	212.7/955.5 $T7$	185/949 $T7$	204/1005 $T7$
$P_T \subset P_I$ (Präfix)	55.96/1036 $T7$	—	—	—	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	97.35/873.8 $T7$	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	159.7/876.3 $T7$	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	218.7/910.6 $T7$	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	224.7/914.6 $T7$	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	194.1/893 $T7$	—	—	—	—	—	—	—	—

Tabelle A.2: B*-Baum Index, 2 Attribute, Partitionierungsart HH, Bereichsanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	206.3/995.1 *T7*	—	—	—	—	—	—	—	—
$P_T = P_I$	211.9/1138 *T7*	190.5/1022 *T7*	213.7/326.4 *I7*	196.6/328.7 *I7*	12.01/33.81 *I7*	184.1/912.5 *T7*	179.6/1021 *T7*	182.3/1020 *T7*	196.3/1166 *T7*
$P_T \subset P_I$ (Präfix)	179.7/719.9 *T7*	—	—	—	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	27.66/669.5 *T7*	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	173.2/954.9 *T7*	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	167/849.5 *T7*	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	169.6/964.5 *T7*	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	167.6/957.6 *T7*	—	—	—	—	—	—	—	—

Tabelle A.3: B*-Baum Index, 2 Attribute, Partitionierungsart HH, Verbundanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$
$P_T \cap P_I = \emptyset$	2.23/3.58 *?*	—	—
$P_T = P_I$	1.86/1.87 *T1*	4.02/4965 *T7*	1.96/473.2 *T1*

Tabelle A.4: B*-Baum Index, 1 Attribut, Partitionierungsart HH, Punktanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$
$P_T \cap P_I = \emptyset$	62.53/1096 *T7*	—	—
$P_T = P_I$	46.4/1029 *T7*	40.6/974.3 *T7*	53.91/977.5 *T7*

Tabelle A.5: B*-Baum Index, 1 Attribut, Partitionierungsart HH, Bereichsanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$
$P_T \cap P_I = \emptyset$	32.75/923.4 *T7*	—	—
$P_T = P_I$	25.21/243.3 *?*	31.75/1015 *T7*	18.78/511.4 *T7*

Tabelle A.6: B*-Baum Index, 1 Attribut, Partitionierungsart HH, Verbundanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	—	—	—	—	—	—	—	—	—
$P_T = P_I$	—	11.01/16.6 *I7*	9.91/3.85 *I1*	11.36/6 *I1*	12.41/4.34 *I1*	12.63/16.66 *I7*	15.25/17.84 *I7*	10.78/15.19 *I7*	15.15/20.61 *I7*
$P_T \subset P_I$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	—	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	—	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	—	—	—	—	—	—	—	—	—

Tabelle A.7: Bitmap Index, 2 Attribute, Partitionierungsart HH, Punktanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	—	—	—	—	—	—	—	—	—
$P_T = P_I$	—	14.67/15.69 *I7*	13.87/27.87 *?*	12.12/310.1 *?*	13.82/10.44 *?*	19.39/21.07 *I7*	13.87/16.22 *I7*	18.8/19.44 *I7*	15.31/16.33 *I7*
$P_T \subset P_I$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	—	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	—	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	—	—	—	—	—	—	—	—	—

Tabelle A.8: Bitmap Index, 2 Attribute, Partitionierungsart HH, Bereichsanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	—	—	—	—	—	—	—	—	—
$P_T = P_I$	—	$1.1 \cdot 10^4$/ 6377 *I7*	$1.3 \cdot 10^4$/ 6367 *I7*	$1.2 \cdot 10^4$/ $2.2 \cdot 10^4$ *I7*	942.6/712.2 *T1*	1928/1219 *I7*	$1.4 \cdot 10^4$/ 8933 *I7*	$1.1 \cdot 10^4$/ 7534 *I7*	$1.3 \cdot 10^4$/ 7242 *I7*
$P_T \subset P_I$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	—	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	—	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	—	—	—	—	—	—	—	—	—

Tabelle A.9: Bitmap Index, 2 Attribute, Partitionierungsart HH, Verbundanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	2.1/0.99 *?*	—	2.24/0.85 *I1*	2.38/0.99 *I1*	—	—	—	—	—
$P_T = P_I$	1.78/0.7 *T1*	—	1.9/0.84 *I1*	1.78/0.93 *I1*	—	—	—	—	—
$P_T \subset P_I$ (Präfix)	2.33/0.58 *T1*	—	1.86/0.72 *I1*	2.04/0.67 *I1*	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	2.01/1.07 *T1*	—	2.13/2.73 *I1*	1.81/0.63 *I1*	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	2.12/0.9 *?*	—	1.96/1.08 *I1*	2.48/0.6 *I1*	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	1.91/0.51 *?*	—	1.92/0.64 *I1*	1.88/0.75 *I1*	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	2.01/0.86 *?*	—	2/0.69 *I1*	2/0.7 *I1*	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	1.87/0.54 *?*	—	1.93/0.65 *I1*	2.08/0.95 *I1*	—	—	—	—	—

Tabelle A.10: B*-Baum Index, 2 Attribute, Partitionierungsart HR, Punktanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	61.15/999.8 *T7*	—	51.63/981.8 *T7*	79.5/1018 *T7*	—	—	—	—	—
$P_T = P_I$	73.56/831 *T7*	—	66.19/849.7 *T7*	65.29/860 *T7*	—	—	—	—	—
$P_T \subset P_I$ (Präfix)	63.05/827.3 *T7*	—	65.69/833.4 *T7*	69.22/53.06 *I1*	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	77.79/895 *T7*	—	58.28/841.2 *T7*	65.79/53.83 *I1*	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	70.48/805.5 *T7*	—	61.26/799.7 *T7*	64.85/810.9 *T7*	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	71.51/853.4 *T7*	—	69.97/865.2 *T7*	63.21/849.7 *T7*	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	67.44/826.9 *T7*	—	70.99/914.9 *T7*	72.68/736.3 *T7*	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	67.18/715 *T7*	—	64.17/725.2 *T7*	67.55/734.2 *T7*	—	—	—	—	—

Tabelle A.11: B*-Baum Index, 2 Attribute, Partitionierungsart HR, Bereichsanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	24.78/873.2 *T7*	—	30.18/888.4 *T7*	38.51/912.7 *T7*	—	—	—	—	—
$P_T = P_I$	72.43/889.6 *T7*	—	93.74/982.2 *T7*	82.14/686.2 *T7*	—	—	—	—	—
$P_T \subset P_I$ (Präfix)	82.01/488.4 *T7*	—	71.69/489.9 *T7*	63.08/100.1 *I2*	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	21.59/800.5 *T7*	—	24.16/611.4 *T7*	64.33/88.01 *I1*	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	75.34/870.9 *T7*	—	64.5/874.9 *T7*	75.3/810.3 *T7*	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	72.54/961.2 *T7*	—	87.07/972.8 *T7*	80.36/931.1 *T7*	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	66.72/935.8 *T7*	—	87.86/898.9 *T7*	69.79/833.7 *T7*	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	63.17/710 *T7*	—	76.86/837.5 *T7*	79.08/773.3 *T7*	—	—	—	—	—

Tabelle A.12: B*-Baum Index, 2 Attribute, Partitionierungsart HR, Verbundanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$
$P_T \cap P_I = \emptyset$	1.62/3.62 *?*	—	3.49/5831 *T7*
$P_T = P_I$	5.08/3.44 *T1*	—	1.77/545.9 *T1*

Tabelle A.13: B*-Baum Index, 1 Attribut, Partitionierungsart HR, Punktanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$
$P_T \cap P_I = \emptyset$	162.9/1349 *T7*	—	79.04/1277 *T7*
$P_T = P_I$	196.8/1176 *T7*	—	56.5/1032 *T7*

Tabelle A.14: B*-Baum Index, 1 Attribut, Partitionierungsart HR, Bereichsanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$
$P_T \cap P_I = \emptyset$	28.25/1034 *T7*	—	25.44/1274 *T7*
$P_T = P_I$	31.06/274.4 *?*	—	34.67/521 *T7*

Tabelle A.15: B*-Baum Index, 1 Attribut, Partitionierungsart HR, Verbundanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	2.23/1.06 *?*	—	1.96/0.93 *I1*	1.75/0.75 *I1*	—	—	—	—	—
$P_T = P_I$	1.81/0.6 *T1*	2.11/4812 *T7*	1.88/1.02 *I1*	1.94/1.01 *I1*	1.84/0.66 *I1*	2.31/508.7 *T1*	1.96/4932 *T7*	1.89/286 *T1*	2.01/4614 *T7*
$P_T \subset P_I$ (Präfix)	2.01/0.92 *T1*	—	2.25/0.85 *I1*	1.84/2.39 *I1*	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	1.76/0.99 *T1*	—	1.98/295.3 *T1*	1.68/0.74 *I1*	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	1.73/404.9 *T1*	—	1.9/0.73 *I1*	1.88/0.62 *I1*	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	1.88/4959 *T7*	—	1.9/0.62 *I1*	1.8/0.68 *I1*	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	1.8/0.59 *T1*	—	2.02/0.78 *I1*	1.78/0.78 *I1*	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	1.56/0.75 *?*	—	2.48/2.9 *I1*	1.81/0.61 *I1*	—	—	—	—	—

Tabelle A.16: B*-Baum Index, 2 Attribute, Partitionierungsart RR, Punktanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	77.56/998.5 *T7*	—	68.59/965.1 *T7*	72.11/962.9 *T7*	—	—	—	—	—
$P_T = P_I$	68.64/141.7 *T1*	65.62/1025 *T7*	67.37/152.7 *T1*	75.9/143.9 *T2*	56.61/234.8 *I4*	68/124.5 *T1*	66.93/971.8 *T7*	67.49/115.4 *T1*	66.13/923.3 *T7*
$P_T \subset P_I$ (Präfix)	70.94/105.5 *T1*	—	65.62/123.5 *T1*	74.46/73.62 *I1*	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	55.24/264.1 *T4*	—	62.45/180.9 *T4*	72.22/68.99 *I2*	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	56.77/129.5 *T2*	—	70.94/117.7 *T2*	67.81/104.1 *T1*	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	67.38/1031 *T7*	—	62.98/1014 *T7*	70.4/992.2 *T7*	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	70.88/114.8 *T1*	—	66.13/113.2 *T1*	65.61/850.9 *T7*	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	59.11/924.8 *T7*	—	68.32/985.4 *T7*	75.58/930.2 *T7*	—	—	—	—	—

Tabelle A.17: B*-Baum Index, 2 Attribute, Partitionierungsart RR, Bereichsanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	85.96/1053 *T7*	—	79.13/1067 *T7*	71.87/1010 *T7*	—	—	—	—	—
$P_T = P_I$	73.68/197.8 *T1*	70.54/1099 *T7*	67.99/198.8 *T1*	82.85/197.7 *T1*	8.03/25.9 *I7*	69.33/186.4 *T1*	76.57/1034 *T7*	68.75/186.2 *T2*	65.25/1006 *T7*
$P_T \subset P_I$ (Präfix)	69.82/149.6 *T1*	—	80.25/171.8 *T1*	74.5/82.13 *I1*	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	7.62/530.4 *T7*	—	69.09/1134 *T7*	94.93/84.58 *I1*	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	71.76/177.5 *T2*	—	68.3/189.7 *T1*	67.21/187.1 *T1*	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	71.52/1064 *T7*	—	67.67/1029 *T7*	72.56/986.9 *T7*	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	76.48/212.8 *T1*	—	67.58/209.4 *T2*	66.89/937.2 *T7*	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	70.64/1093 *T7*	—	76.56/1035 *T7*	64.98/1016 *T7*	—	—	—	—	—

Tabelle A.18: B*-Baum Index, 2 Attribute, Partitionierungsart RR, Verbundanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$
$P_T \cap P_I = \emptyset$	2.12/3.2 *?*	—	2.03/4782 *T7*
$P_T = P_I$	2.99/1.65 *T1*	5.42/4954 *T7*	1.98/400.9 *T1*

Tabelle A.19: B*-Baum Index, 1 Attribut, Partitionierungsart RR, Punktanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$
$P_T \cap P_I = \emptyset$	48.83/986.6 *T7*	—	49.28/1001 *T7*
$P_T = P_I$	44.46/130.2 *T1*	40.32/1147 *T7*	39.67/133 *T1*

Tabelle A.20: B*-Baum Index, 1 Attribut, Partitionierungsart RR, Bereichsanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$
$P_T \cap P_I = \emptyset$	27.45/929.9 *T7*	—	29.03/1053 *T7*
$P_T = P_I$	8.66/108 *T1*	21.37/1016 *T7*	7.87/112.3 *T1*

Tabelle A.21: B*-Baum Index, 1 Attribut, Partitionierungsart RR, Verbundanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	—	—	—	—	—	—	—	—	—
$P_T = P_I$	—	13.77/22.51 *I7*	15.93/6.69 *I1*	16.25/7.02 *I1*	12.59/3.02 *I1*	15.16/5.07 *I1*	11.22/19.1 *I7*	12.71/6.03 *I1*	12.84/21.24 *I7*
$P_T \subset P_I$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	—	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	—	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	—	—	—	—	—	—	—	—	—

Tabelle A.22: Bitmap Index, 2 Attribute, Partitionierungsart RR, Punktanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	—	—	—	—	—	—	—	—	—
$P_T = P_I$	—	14.3/16.27 *I7*	19.23/606.3 *?*	15.12/1698 *?*	13.7/8.31 *?*	13.3/265.9 *?*	12.82/14.14 *I7*	13.66/218.9 *?*	13.43/15.31 *I7*
$P_T \subset P_I$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	—	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	—	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	—	—	—	—	—	—	—	—	—

Tabelle A.23: Bitmap Index, 2 Attribute, Partitionierungsart RR, Bereichsanfrage

	$P_I = \emptyset$ (IS statt P_I)	$P_I \cap IS = \emptyset$	$P_I = IS$	$P_I \subset IS$ (Präfix)	$P_I \subset IS$ (¬Präfix)	$P_I \supset IS$ (Präfix)	$P_I \supset IS$ (¬Präfix)	$P_I \cap IS \neq \emptyset$ (Präfix)	$P_I \cap IS \neq \emptyset$ (¬Präfix)
$P_T \cap P_I = \emptyset$	—	—	—	—	—	—	—	—	—
$P_T = P_I$	—	$1.3 \cdot 10^4$/ 7877 *I7*	$2.4 \cdot 10^4$/ $1.2 \cdot 10^4$ *I1*	$1.0 \cdot 10^4$/ $1.0 \cdot 10^4$ *I1*	917.5/1054 *T1*	$1.3 \cdot 10^4$/ $1.2 \cdot 10^4$ *I1*	$1.3 \cdot 10^4$/ 6636 *I7*	$1.3 \cdot 10^4$/ $1.3 \cdot 10^4$ *I1*	928.9/676.2 *I7*
$P_T \subset P_I$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \subset P_I$ (¬Präfix)	—	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \supset P_I$ (¬Präfix)	—	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (Präfix)	—	—	—	—	—	—	—	—	—
$P_T \cap P_I \neq \emptyset$ (¬Präfix)	—	—	—	—	—	—	—	—	—

Tabelle A.24: Bitmap Index, 2 Attribute, Partitionierungsart RR, Verbundanfrage

Literatur

[Bau99] Mark Bauer. *Oracle8i Tuning, Release 8.1.5.* Oracle Corporation, Redwood, CA, 1999

[Bau00] Robert Baumgarten. *Zugriffspfadstrukturen für partitionierte Tabellen in Datenbanksystemen.* Diplomarbeit, Friedrich-Schiller-Universität Jena, Institut für Informatik, April 2000

[Bay96] Rudolf Bayer. *The Universal B-Tree for Multidimensional Indexing.* Technischer Bericht TUM-I9637, Technische Universität München, November 1996

[CLR92] Thomas H. Cormen, Charles E. Leiserson und Ronald L. Rivest. *Introduction to Algorithms.* Cambridge, Mass.: MIT Press, 1992

[Dad96] Peter Dadam. *Verteilte Datenbanken und Client-/Server-Systeme.* Berlin, Heidelberg: Springer, 1996

[Fee99] Joyce Fee. *Oracle8i Administrator's Guide, Release 8.1.5.* Oracle Corporation, Redwood, CA, 1999

[Heß97] Klaus Heßen. *Oracle8: Tabellen- und Indexpartitionierung, Paralleles DML.* Oracle white paper, Oracle Deutschland GmbH, Juni 1997

[HR99] Theo Härder und Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung.* Berlin, Heidelberg: Springer, 1999

[HS97] Andreas Heuer und Gunter Saake. *Datenbanken: Konzepte und Sprachen.* Bonn: mITP-Verlag, 1997

[ISO92] International Organization for Standardization, Genf. *Information technology, database languages, SQL*, 1992. International Standard ISO/IEC 9075:1992(E)

[LO99] Diana Lorentz und Denise Oertel. *Oracle8i SQL Reference, Release 8.1.5.* Oracle Corporation, Redwood, CA, 1999

[LR99] Lefty Leverenz und Diana Rehfield. *Oracle8i Concepts, Release 8.1.5.* Oracle Corporation, Redwood, CA, 1999

[LS90] David B. Lomet und Betty Salzberg. *The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance. TODS*, **15**(4):625–658, 1990

[Mit95] Bernhard Mitschang. *Anfrageverarbeitung in Datenbanksystemen: Entwurfs- und Implementierungskonzepte.* Braunschweig: Vieweg, 1995

[MN00] Thomas Müller und Jan Nowitzky. *Entwurf von Partitionierungsschemata.* Jenaer Schriften zur Mathematik und Informatik Math/Inf/00/20, Institut für Informatik, Friedrich-Schiller-Universität Jena, Juli 2000

[Mül00] Thomas Müller. *Partitionierung beim strukturellen Entwurf von Datenbankschemata.* Studienarbeit, Friedrich-Schiller-Universität Jena, Juni 2000

[NM00] Jan Nowitzky und Thomas Müller. *Entwurf und Bewertung von Partitionierungsstrategien für Datenbankschemata.* Jenaer Schriften zur Mathematik und Informatik Math/Inf/00/29, Institut für Informatik, Friedrich-Schiller-Universität Jena, Oktober 2000

[Rah94] Erhard Rahm. *Mehrrechner-Datenbanksysteme – Grundlagen der verteilten und parallelen Datenbankverarbeitung.* München: Addison-Wesley, 1994

[Rob81] John T. Robinson. *The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes.* In: Y. Edmund Lien (Hg.), *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, April 29 - May 1, 1981*, Seiten 10–18. ACM Press, 1981

[Sed92] Robert Sedgewick. *Algorithmen.* Bonn, München: Addison-Wesley, 1992

[SH99] Gunter Saake und Andreas Heuer. *Datenbanken: Implementierungstechniken.* Bonn: mITP-Verlag, 1999

[TPC99] Transaction Processing Performance Council. *TPC Benchmark H*, 1999. Revision 1.2.1

[WB98] Ming-Chuan Wu und Alejandro P. Buchmann. *Encoded Bitmap Indexing for Data Warehouses.* In: *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, Seiten 220–230. IEEE Computer Society, 1998

[Wed74] Hartmut Wedekind. *On the Selection of Access Paths in a Data Base System.* In: J. W. Klimbie und K. L. Koffeman (Hg.), *Proceedings of the IFIP Working Conference Data Base Management, Cargèse, Corsica, France, 1–5 April, 1974*, Seiten 385–398. 1974