

Otto-von-Guericke-Universität Magdeburg



Fakultät für Informatik  
Institut für Technische und Betriebliche Informationssysteme

## Diplomarbeit

### **Verwaltung von Statistiken zu Indexkonfigurationen für das Self-Tuning**

Verfasser:

**Kersten Kühne**

18. Februar 2009

Betreuer:

**Dr.-Ing. Eike Schallehn**

Universität Magdeburg  
Fakultät für Informatik  
Postfach 4120, D-39016 Magdeburg  
Germany

**Kühne, Kersten:**

*Verwaltung von Statistiken zu Indexkonfigurationen für das Self-Tuning*

Diplomarbeit, Otto-von-Guericke-Universität  
Magdeburg, 2008.

---

---

# Inhaltsverzeichnis

<b>Inhaltsverzeichnis</b>	<b>i</b>
<b>Abbildungsverzeichnis</b>	<b>v</b>
<b>Tabellenverzeichnis</b>	<b>vii</b>
<b>Verzeichnis der Abkürzungen</b>	<b>ix</b>
<b>1 Einführung</b>	<b>1</b>
1.1 Einleitung . . . . .	1
1.2 Ziele . . . . .	2
1.3 Gliederung . . . . .	2
<b>2 Grundlagen</b>	<b>3</b>
2.1 Datenbanksysteme . . . . .	3
2.1.1 Relationale Datenbanksysteme . . . . .	4
2.1.2 Datenspeicherung in relationalen Datenbanksystemen . . . . .	6
2.1.3 Dateiorganisationsformen . . . . .	8
2.2 Zugriffsverfahren . . . . .	10
2.2.1 Klassifikation von Zugriffsverfahren . . . . .	11
2.2.2 B-Bäume . . . . .	15
2.2.3 Hash-Verfahren . . . . .	18
2.2.4 Bitmap-Indexe . . . . .	20
2.2.5 Mehrdimensionale Verfahren . . . . .	20
2.2.6 Indexverwendung . . . . .	21
2.3 Anfrageverarbeitung . . . . .	22
2.3.1 Ablauf der Optimierung . . . . .	22
2.3.2 Berechnung von Verbunden . . . . .	24

<b>3</b>	<b>Automatische Indexauswahl</b>	<b>27</b>
3.1	Index Selection Problem . . . . .	27
3.2	DB2 Design Advisor . . . . .	28
3.2.1	Architektur . . . . .	29
3.2.2	Optimierung einzelner Queries . . . . .	31
3.2.3	Optimierung von Workloads . . . . .	32
3.3	Weitere Betrachtungen . . . . .	34
<b>4</b>	<b>Analyse und Konzept</b>	<b>39</b>
4.1	Analyse . . . . .	39
4.1.1	Rucksackprobleme - Ein Beispiel . . . . .	39
4.1.2	Indexabhängigkeiten . . . . .	41
4.1.3	Profitberechnung einzelner Indexe . . . . .	44
4.1.4	Zusammenfassung . . . . .	50
4.2	Konzept . . . . .	50
4.2.1	Anforderungen und Ziele . . . . .	50
4.2.2	LISA - Einleitung . . . . .	51
4.2.3	LISA - Statistiken lokaler Indexkonfigurationen . . . . .	52
4.2.4	LISA - Indexauswahlverfahren . . . . .	55
4.2.5	LISA - Vor- und Nachteile . . . . .	60
4.2.6	LISA - Online . . . . .	62
4.3	Zusammenfassung . . . . .	63
<b>5</b>	<b>Implementierung und Evaluierung</b>	<b>65</b>
5.1	LISA - Implementierung . . . . .	65
5.1.1	JLISA - Aufbau . . . . .	65
5.1.2	JLISA - Programmablauf . . . . .	65
5.2	Evaluierung . . . . .	68
5.2.1	Vorbetrachtungen . . . . .	69
5.2.2	Erster Testlauf 300 Megabyte . . . . .	70
5.2.3	Lösungen 150 Megabyte . . . . .	71
5.2.4	Lösungen 300 Megabyte . . . . .	72
5.2.5	Lösungen 700 Megabyte . . . . .	72

---

---

5.3 Zusammenfassung . . . . .	73
<b>6 Zusammenfassung und Ausblick</b>	<b>75</b>
<b>A Anhang</b>	<b>77</b>
A.1 TPC-H . . . . .	77
A.2 Explain Tables . . . . .	79
A.3 Testsystem . . . . .	79
A.4 Indexabhängigkeiten I . . . . .	80
A.5 Indexabhängigkeiten II . . . . .	81
A.6 Indexempfehlungen für den verwendeten Workload . . . . .	81
<b>Literaturverzeichnis</b>	<b>85</b>



---

---

# Abbildungsverzeichnis

2.1	Fünf-Schichten-Architektur nach [H87]	4
2.2	Tabelle Person im Relationenmodell	5
2.3	Einfache SQL-Anfrage	6
2.4	Speicherung von Datensätzen	8
2.5	Index-sequentielle Speicherung	9
2.6	Abfragedauer mit und ohne Index (Eigene Messung)	11
2.7	Dünn- und Dichtbesetzter Index ([SB03] S. 91)	12
2.8	Index Clustering ([SB03] S. 93)	13
2.9	Prinzipieller Aufbau eines B-Baumes (vgl. [SH99] S. 149)	16
2.10	Statisches Hashen	19
2.11	Einfacher Bitmap Index	20
2.12	Phasen der Anfragebearbeitung ([SH99] S. 352)	22
2.13	Umformung von Anfrageplänen	23
3.1	Architektur des DB2 Design Advisors	29
3.2	Ergebnisauszug db2advis	30
3.3	Kosten von Indexkonfigurationen für den TPC-H Workload	34
3.4	Monitor-Diagnose-Zyklus des Design Alerter ([BC06])	35
3.5	Architektur des QUIET-Demos ([SGS03])	36
4.1	Indexabhängigkeiten - Merge-Sort-Join	43
4.2	Query I	44
4.3	Query II	45
4.4	Profitzuweisung DB2	46
4.5	Anfragepläne Query I	47
4.6	Anfragepläne Query II	47

---

---

4.7	Anfragepläne Query II . . . . .	48
4.8	Durchschnittliche Bearbeitungszeiten $W = \{Q_I, Q_{II}\}$ . . . . .	49
4.9	Lokal-optimale Indexkonfigurationen . . . . .	52
4.10	Überlappung lokaler Indexkonfigurationen . . . . .	52
4.11	Behandlung von Überlappungen . . . . .	54
4.12	LISA - Übersicht . . . . .	55
4.13	Hinzufügen neuer lokal-optimaler Indexkonfigurationen . . . . .	57
4.14	ER-Modell der Datenspeicherung in LISA . . . . .	57
4.15	Berechnung $\mathcal{O}(\mathcal{I})$ . . . . .	59
4.16	Neuberechnungen nach Auswahl einer Indexkonfiguration . . . . .	60
4.17	LISA - Nachteile . . . . .	61
5.1	Lisa - DB2 Design Advisor . . . . .	73
A.1	Indexabhängigkeiten . . . . .	80
A.2	Indexabhängigkeiten II . . . . .	81

# Tabellenverzeichnis

2.1	Abbildung der konzeptuellen Ebene auf das Dateisystem ([SH99] S. 89) . . . . .	7
2.2	Einordnung von Zugriffsverfahren ([SH99] S. 130) . . . . .	15
2.3	Zugriffsstrukturen einiger DBS ([SB03] S. 108) . . . . .	21
4.1	Rucksackproblem . . . . .	40
4.2	Optimale Lösung für $Q_I$ und $Q_{II}$ . . . . .	45
4.3	Indexe in $\mathcal{I}_{I_{opt}}$ und $\mathcal{I}_{II_{opt}}$ . . . . .	46
4.4	Indexkonfigurationen für $W$ . . . . .	49
4.5	Statistiken für Indexkonfigurationen und Indexe . . . . .	54
5.1	Antwortzeiten unoptimierter Workload . . . . .	69
5.2	Indexkonfigurationen Workload . . . . .	70
5.3	Indexauswahl 150 MB . . . . .	71
5.4	Indexauswahl 300 MB . . . . .	72
5.5	Indexauswahl 700 MB . . . . .	72
A.1	Indexe Workload . . . . .	84



# Verzeichnis der Abkürzungen

<b>CLOB</b>	Character Large Object
<b>DBMS</b>	Datenbank-Management-System
<b>DBS</b>	Datenbanksystem
<b>GB</b>	Gigabyte
<b>ISP</b>	Index Selection Problem
<b>KB</b>	Kilobyte
<b>LISA</b>	Local Indexset Assortment
<b>LOB</b>	Large Object
<b>MB</b>	Megabyte
<b>TPC</b>	Transaction Processing Performance Council
<b>SAEFIS</b>	Smart Column Enumeration for Index Scans
<b>SQL</b>	Structured Query Language
<b>TID</b>	Tuple Identifier



# Kapitel 1

## Einführung

### 1.1 Einleitung

Das Zeitalter der Informationsgesellschaft ist geprägt durch die Gewinnung, Speicherung und Verarbeitung von Informationen. Für die Speicherung von Informationen ist heutzutage eine Technologie von zentraler Bedeutung: Datenbanken.

Wie in Computersystemen aktuell üblich, speichern auch Datenbanken Informationen in der Regel auf Magnetfestplatten. Magnetfestplatten, als eines der letzten verbliebenen mechanischen Bestandteile moderner Computer, bilden aber den Engpass beim Speichern und Lesen von Informationen aus Datenbanken. Man ist deshalb seither bestrebt, die Auswirkungen dieses Engpasses zu minimieren. In Datenbanken spielen Indexe dabei eine wichtige Rolle.

Die Auswahl von Indexen ist für die Leistungsfähigkeit (hier vor allem Antwortzeiten) einer Datenbank von großer Bedeutung. Eine vorteilhafte Indexauswahl kann das Lesen und Schreiben von Informationen um ein Vielfaches beschleunigen. Eine schlechte Indexauswahl kann die Leistungsfähigkeit einer Datenbank aber auch verringern. Die Identifikation nützlicher Indexe spielt beim Datenbank-Tuning, also der Leistungsverbesserung, eine entscheidende Rolle und wird als Index-Selection-Problem (kurz ISP) bezeichnet.

Seit einigen Jahren existieren automatisierte Lösungen des ISP [ZRL<sup>+</sup>04], [Ora03], [CN98] im Bereich des Datenbank-Selbsttuning. Diese Lösungen modellieren das ISP als Rucksackproblem auf Basis von (erwartetem) Indexnutzen und (erwarteter) Indexgröße. Diese Betrachtungen vernachlässigen jedoch mögliche Abhängigkeiten zwischen Indexen. Abhängigkeiten können dazu führen, dass ein Index A den erwarteten Nutzen nur beim Vorhandensein eines Indexes B erreichen kann, oder ein Index A bei der Existenz eines Indexes B einen Großteil seines Nutzens einbüßt und führen somit eventuell zu einer suboptimalen Indexauswahl.

Um eine optimale Indexauswahl zu treffen, müssten alle möglichen Kombinationen von Indexen gebildet und in Hinsicht auf ihren Nutzen evaluiert werden. Eine solche Vorgehensweise würde jedoch zu einer kombinatorischen Explosion mit nicht vertretbarem Aufwand führen, so dass an der Betrachtung als Rucksackproblem festgehalten wird.

## 1.2 Ziele

In der Arbeit soll untersucht werden, ob die Betrachtung des ISP als Rucksackproblem auf Basis einzelner Indexe optimale Lösungen liefern kann. In diesem Zusammenhang sollen Abhängigkeiten zwischen Indexen und deren Auswirkungen auf das ISP betrachtet werden.

Aufbauend auf den gewonnen Ergebnissen der oben genannten Betrachtungen, soll ein Verfahren entwickelt werden, welches Indexabhängigkeiten bei der automatischen Indexauswahl besser berücksichtigt. Das Verfahren soll mit möglichst geringem Aufwand und Datenbedarf arbeiten. Das Problem der kombinatorischen Explosion muss dabei vermieden werden.

## 1.3 Gliederung

Die vorliegende Arbeit ist in sechs Kapitel unterteilt, deren Inhalt im Folgenden kurz vorgestellt wird. In diesem ersten Kapitel werden nach einer kurzen Einleitung Ziele und Gliederung der Arbeit aufgezeigt.

Das zweite Kapitel behandelt Arten und Funktionsweise von Indexen und gibt einen kleinen Einblick in die Arbeitsweise relationaler Datenbanken.

Kapitel drei stellt Ergebnisse von Forschungen im Bereich der automatischen Indexauswahl vor. Einige dieser Forschungen bilden die Grundlage des in Kapitel vier beschriebenen Konzeptes.

In Kapitel fünf wird die Implementierung des zuvor erarbeiteten Konzeptes beschrieben. Bei der anschließenden Evaluierung soll festgestellt werden, ob eine Verbesserung der Indexauswahl erreicht werden konnte.

Die Arbeit schließt mit einer Zusammenfassung des Geleisteten und einem Ausblick auf mögliche Erweiterung in Kapitel sechs.

# + Kapitel 2

## Grundlagen

In diesem Kapitel werden die benötigten Grundlagen der Arbeit behandelt. Im ersten Unterabschnitt werden relationale Datenbanksysteme kurz beschrieben und einige Begriffe erläutert. Um zu erklären wie Indexe die Geschwindigkeit der Anfrageverarbeitung erhöhen können, befaßt sich ein Teil dieses Unterabschnittes mit der Speicherung von Daten in relationalen Datenbanksystemen.

Der zweite Abschnitt befaßt sich mit dem effizienten Zugriff auf die gespeicherten Daten. Es werden verschiedene Dateiorganisationsformen, Zugriffsstrukturen und Indexarten vorgestellt. Darüber hinaus werden einige Indexverfahren, wie zum Beispiel B-Bäume, präsentiert und untersucht. Dieser Abschnitt bildet den Hauptteil dieses Kapitels.

Die Verarbeitung und Optimierung von Datenbankabfragen ist Inhalt des Abschnittes 2.3. Die bei der Optimierung erzeugten Zugriffspläne geben Aufschluss über die Verwendung und den Nutzen von Indexen.

### 2.1 Datenbanksysteme

Datenbanksysteme haben die Aufgabe große Datenbestände zu verwalten und mehreren Nutzern gleichzeitig zur Verfügung zu stellen. Ein Datenbanksystem besteht aus einer, oder mehreren, *Datenbank(en)* und einem *Datenbank-Management-System* (kurz DBMS). Unter einer Datenbank versteht man dabei einen strukturierten, von einem DBMS verwalteten Datenbestand (vgl. [AH01] S. 9). Als Datenbank-Management-System bezeichnet man die Software, die zur Verwaltung von Datenbanken eingesetzt wird. Die von einem DBMS erwarteten Basis-Funktionalitäten sind in den neun Codd'schen Regeln, s. Anhang, definiert.

Datenbank-Management-Systeme sind nach dem Fünf-Schichtenmodell von Härder [H87] aufgebaut. Die einzelnen Ebenen abstrahieren von den entsprechenden untergeordneten Ebenen und stellen Funktionen und Objekte für die nächsthöhere Ebene bereit. Abbildung 2.1 zeigt die fünf Schichten eines DBMS und den „Weg“ der Anfrageabarbeitung beziehungsweise der Ergebniserstellung.

Die Aufgaben und Funktionen der gezeigten Ebenen werden im Verlauf der Arbeit, sofern benötigt, näher beschrieben und erklärt.

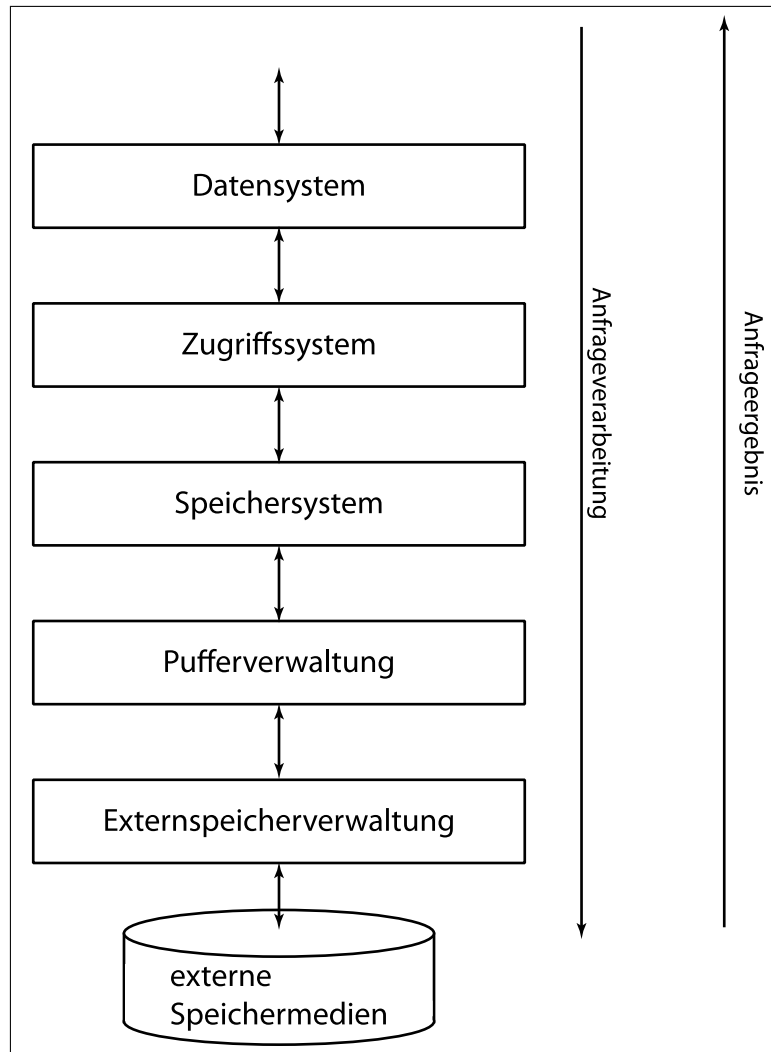


Abbildung 2.1: Fünf-Schichten-Architektur nach [H87]

### 2.1.1 Relationale Datenbanksysteme

Relationale Datenbanksysteme setzen das Relationenmodell von E.F. Codd um. Die Speicherung von Daten kann man sich dabei in Form von Tabellen vorstellen. Abbildung 2.2 auf Seite 5 veranschaulicht die Datenspeicherung in Tabellenform und soll uns als Begriffsklärung dienen.

In der Tabelle Person können die Merkmale ID, Name und Wohnort von Personen gespeichert werden. Für jede Person wird dafür eine Tabellenzeile angelegt. Einzelne Zeilen werden als *Tupel* bezeichnet. Die Menge aller Tupel, also aller Tabellenzeilen beziehungsweise Personen, bilden die *Relation*. Informationen über die Struktur einer Relation und ihrer zugehörigen Tupel finden sich im Relationenschema. Die einzelnen Spaltenüberschriften werden fortan als *Attribute* bezeichnet. Die zugehörigen Werte einer Spalte, zum Beispiel Magdeburg in der Spalte Wohnort, heißen *Attributwert*.

Während Name und Wohnort typische Informationen zu Personen sind, ist das Attribut *ID* von anderer Bedeutung. Es dient der eindeutigen Identifikation einer Person in der Datenbank. Ein solches identifizierendes Attribut wird als *Schlüssel* bezeichnet. Das Attribut Name wäre dann ein Schlüssel, wenn es keine Personen mit gleichen Namen ge-

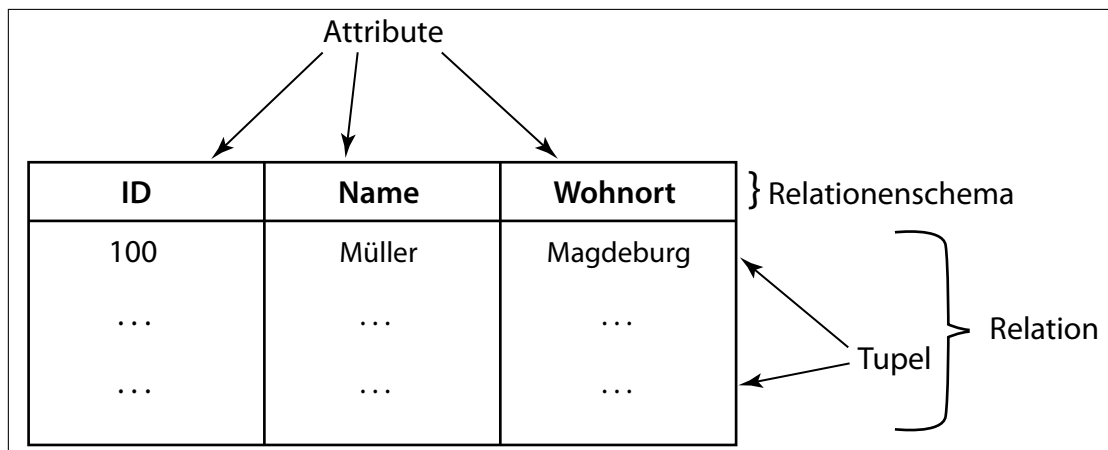


Abbildung 2.2: Tabelle Person im Relationenmodell

ben würde. Auch Kombinationen von Attributen können Schlüssel sein. Die im weiteren Verlauf der Arbeit eingeführten *Suchschlüssel* erfüllen die Forderung nach Eindeutigkeit nicht.

Enthält eine Tabelle ein Attribut, das Schlüsselwerte einer anderen Tabelle enthält, spricht man von *Fremdschlüsseln*.

Die grundlegenden Operationen die auf Relationen ausgeführt werden können, sind in der Relationenalgebra zusammengefasst. Die wichtigsten Operationen werden im Folgenden kurz vorgestellt.

## Selektion

Die Selektion ermöglicht die Auswahl von Tabellenzeilen. Die Auswahl erfolgt über den Vergleich ( $=, \neq, >, \geq, \dots$ ) von Attributwerten einer Zeile und dem Selektionskriterium. Die Selektion wird mit dem griechischen Buchstaben  $\sigma$  (Sigma) gekennzeichnet. Die folgende Anfrage wählt aus der Tabelle Person alle Personen mit Namen Müller aus:

$$\sigma_{Name='Müller'}(Person)$$

## Projektion

Bei der Projektion werden Tabellenspalten als Ergebnis zurückgegeben. Die Tabelle wird also vertikal zerlegt. In der Relationenalgebra werden bei der Projektion doppelte Attributwerte entfernt. Die Notation der Projektion ist wie folgt:

$$\pi_{Name}(Person)$$

Die oben aufgeführte Anfrage zeigt alle Einträge aus der Tabellenspalte Name. Wie bereits erwähnt werden Duplikate eliminiert.

## Natürlicher Verbund

Stellen wir uns neben der Tabelle *Person* eine weitere Tabelle *Telefon* mit Telefonnummern von Personen vor. Neben dem Attribut *Telefon* enthält die Tabelle *Telefon* auch das Attribut *ID* als Fremdschlüssel. *ID* bezieht sich auf das Attribut *ID* der *Personen*-Tabelle und stellt damit eine eindeutige Verbindung von Telefonnummern (über gleiche Schlüsselwerte) zu Personen her. Die Herstellung eines Verbundes (engl. *join*) der Tabellen *Person* und *Telefon* wird wie folgt notiert:

$$(Person) \bowtie (Telefon)$$

An dieser Stelle soll nicht weiter auf das Relationenmodell eingegangen werden. Es sei stattdessen auf [Vos99] S. 415 ff. für weitere Informationen verwiesen.

Die Operationen der Relationenalgebra sind in der Anfragesprache *SQL* (Structured Query Language) implementiert. *SQL* ermöglicht u.a. die Abfrage, Manipulation und Definition von Daten. Die *SQL*-Anweisung in Abbildung 2.3 gibt als Ergebnis *ID* und *Name* (Projektion) aller Personen mit dem Wohnort Magdeburg (Selektion) zurück. Das Anfrageergebnis kann ebenfalls als Tabelle beziehungsweise Relation verstanden werden. Da das Attribut *ID* als eindeutig definiert wurde, kann auf die explizite Eliminierung von Duplikaten, wie sie in der Relationenalgebra stattfindet, durch die Angabe des *SQL* Schlüsselwortes *distinct* verzichtet werden.

```
select ID, Name
from   Person
where  Wohnort = 'Magdeburg'
```

Abbildung 2.3: Einfache *SQL*-Anfrage

### 2.1.2 Datenspeicherung in relationalen Datenbanksystemen

[H87] unterscheidet in einer zweistufigen Speicherhierarchie zwischen *Hauptspeicher* (auch Primärspeicher) und *externem Speicher* (auch Sekundärspeicher). Die Datenverarbeitung findet im schnellen Hauptspeicher statt. Auf Grund seiner *Flüchtigkeit* und seiner, im Vergleich zum externen Speicher, verhältnismäßig kleinen Kapazität (bei 32-Bit-Adressierung nur  $2^{32}$  Bytes direkt adressierbar) ist der Hauptspeicher nicht für die Datenspeicherung geeignet. Die (langfristige) Datenspeicherung erfolgt stattdessen auf *nicht-flüchtigen* externen Speichermedien. In der Regel sind dies Magnetfestplatten mit Speicherkapazitäten von mehreren hundert Gigabyte (Stand Juli 2008) pro Medium. Der Geschwindigkeitsunterschied zwischen Haupt- und Externspeicher wird in der Literatur [SH99], [H87] und [GMUW00] übereinstimmend auf den Faktor  $10^5$  beziffert und als *Zugriffslücke* bezeichnet. In Datenbanksystemen wird daher versucht die Anzahl der externen Speicherzugriffe durch Pufferverfahren (siehe [H87] S. 193 ff) und die Verwendung geeigneter Zugriffsverfahren, letztere werden nachstehend vorgestellt, zu minimieren. In dieser Arbeit wird von Magnetfestplatten (kurz Festplatte oder Platte) als Externspeicher ausgegangen und die Begriffe synonym verwendet.

Bevor im Folgenden kurz die Speicherung und das Lesen von Daten betrachtet wird, sollen hier die Begriffe Block und Seite geklärt werden. Ein *Block* ist die kleinste Datenzugriffseinheit auf der Platte und hat eine fest vorgegebene Größe. Vereinfacht kann man sich den externen Speicher als Aneinanderreihung von Blöcken vorstellen. Die physischen Blöcke werden in den höheren Schichten von Datenbanksystemen den *Seiten* zugeordnet und über Seitennummer und einem Positionsoffset adressiert. Seiten sind die Einheiten des Datentransfers zwischen externem und Hauptspeicher. Mehrere zusammenhängende Seiten bilden eine *Datei*. Die Seiten sind dabei als verkettete Listen, mit Informationen über Vorgänger- und Nachfolgeseite, organisiert.

Tabelle 2.1 zeigt wie die Objekte aus dem Relationenmodell, hier die *Konzeptuelle Ebene*, auf externen Datenträgern abgebildet werden. Die in der Tabelle angegebene Zwischenstufe, die *Interne Ebene*, entspricht dem Daten- oder Zugriffssystem der Fünf-Schichten-Architektur in Abbildung 2.1 auf Seite 4 und abstrahiert von den physischen Objekten wie Blöcken und Dateien, wie sie noch in der Externspeicherverwaltung zu finden sind. Die gezeigte Abbildung ist in beide Richtungen zu interpretieren. Damit ist gemeint, dass eine als Datei gespeicherte Relation auch wieder aus der Datei herstellbar sein muß. Diese Forderung gilt entsprechend für Tupel und Attributwerte.

Konzeptuelle Ebene		Interne Ebene		Dateisystem/Platte
Relationen	→	Logische Dateien	→	Physische Dateien
Tupel	→	Datensätze (Records)	→	Seiten/Blöcke
Attributwerte	→	Felder	→	Bytes

Tabelle 2.1: Abbildung der konzeptuellen Ebene auf das Dateisystem ([SH99] S. 89)

Einhergehend mit Tabelle 2.1 kann die Speicherung von Relationen wie folgt beschrieben werden (vgl. [GMUW00] S. 83):

- Attribute werden als Folge von Bytes gespeichert und intern als Felder bezeichnet.
- Felder werden zu Datensätzen zusammengefügt und repräsentieren Tupel.
- Datensätze werden in Blöcken gespeichert.
- Alle Blöcke einer Relation werden in Dateien zusammengefaßt gespeichert.

In Anlehnung an [GMUW00] S. 92 veranschaulicht Abbildung 2.4 das sogenannte *Blocken* von Datensätzen. Ausgehend von der Relation Person ist zu sehen, wie die Attribute ID (4 Byte), Name (200 Byte) und Wohnort (100Byte) als Datensätze gespeichert werden. Zu sehen ist ein vereinfachter Datensatz mit fester Länge und ein physischer Block, der mehrere Datensätze aufnehmen kann. Sätze mit fester Länge haben den jeweils gleichen Speicherplatzbedarf. Auf Sätze mit variabler Länge soll hier nicht weiter eingegangen werden. Im Header eines Blockes können u.a. die Position der Datensätze und der Zeitpunkt der letzten Änderungen im Block gespeichert werden.

### TID-Konzept

Im Vorfeld wurde bereits erwähnt, dass Datensätze über die Angabe von Seitennummer und einem Offset adressiert werden können. Ein *Tupel-Identifikator* (TID) ist eine solche

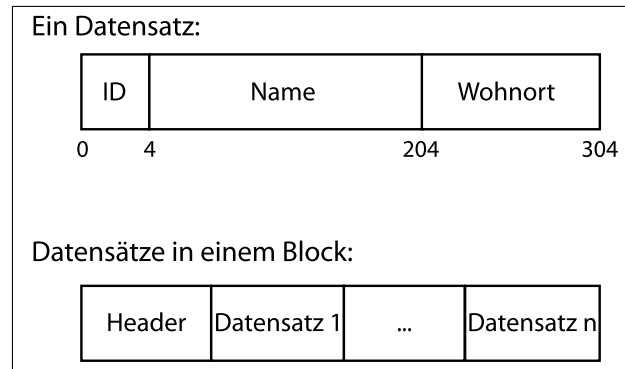


Abbildung 2.4: Speicherung von Datensätzen

Datensatzadresse bestehend aus Seitennummer und Offset. Anstatt direkt auf die Position eines Datensatzes innerhalb einer Seite zu verweisen, zeigt der Offsetwert  $i$  eines TID auf den  $i$ -ten Eintrag in einer Datensatzliste am Anfang der entsprechenden Seite. In dieser Datensatzliste finden sich nun ein *Zeiger* auf die Position des gewünschten Datensatzes. Diese Verfahrensweise ermöglicht es Datensätze ohne großen Aufwand (nur das Zeigerfeld auf der Seite selbst muß angepaßt werden) zu verschieben.

Zum Abschluß dieses Abschnitts werden hier noch kurz die zur Verfügung stehenden Dateioperationen nach [SH99] S. 50 ff, bezogen auf das Zugriffssystem (interne Ebene), vorgestellt. Das Einfügen von Datensätzen wird als *insert* bezeichnet. Die Operationen *remove* und *delete* löschen Datensätze. Für die Modifikation von Datensätzen verwendet man *modify*. Das Suchen von Datensätzen erfolgt über die *lookup*, beziehungsweise *fetch*, Operationen.

Beim *lookup* unterscheidet man folgende Arten:

- **single-match query:** Gegeben ist *ein* Wert eines bestimmten Attributes. Der „lookup“ liefert als Ergebnis alle Tupel, die diesen Attributwert besitzen.
- **exact-match query:** Für die Suche sind Werte für *alle* Attribute einer Relation angegeben. Im späteren Verlauf der Arbeit beziehen sich die Attribute auf Schlüsselwerte eines Indexes.
- **partial-match query:** Es sind *einige* Attributswerte einer Relation werden für die Suche angegeben.
- **range query/Bereichsanfrage:** *Ein oder mehrere* Attribute werden nach Werten eines bestimmten Bereichs gesucht und geben die passenden Tupel zurück.

In der SQL-Anfrage aus Abbildung 2.3 (siehe Seite 6) entspricht der Selektionsteil „where Wohnort = 'Magdeburg'“ einem *single-match query* mit dem Suchwert „Magdeburg“ über dem Attribut „Wohnort“.

### 2.1.3 Dateiorganisationsformen

An dieser Stelle sollen noch kurz die wesentlichen Arten Datensätze in Dateien zu speichern beschrieben werden. Unter *Dateiorganisationsform* versteht man die Form der Speicherung interner Relationen.

## Heap-Organisation

Bei der Heap-Organisation werden Datensätze auf „einem Haufen gestapelt“. Die physische Reihenfolge der Datensätze entspricht dem Zeitpunkt der Aufnahme selbiger. Das Einfügen (insert) von neuen Datensätzen geschieht mit sehr geringem Aufwand, weil neue Datensätze schlicht an das Ende der Datei angefügt werden. Die Suche (lookup) erfordert das Durchsuchen aller Datensätze, was dem Maximalaufwand entspricht. Diese Art der Suche nennt man *Full-Table-Scan*. Da für das Löschen und das Modifizieren von Datensätzen jeweils ein lookup notwendig ist, sind auch diese Operationen sehr teuer.

## Sequentielle Speicherung

In sequentiell gespeicherten Relationen sind die Datensätze sortiert gespeichert. Die Relation Person könnte zum Beispiel nach Wohnorten sortiert gespeichert werden. Die Suche nach Datensätzen mit dem Wohnort Magdeburg kann beim sequentiellen Lesen vom Dateianfang beendet werden, wenn ein Attributwert alphabetisch betrachtet „größer“ als Magdeburg (zum Beispiel München) ist. Durch die Forderung einer sortierten Datei ist das Einfügen von Datensätzen komplizierter als noch bei der Heap-Organisation und erfordert unter Umständen das Verschieben bereits vorhandener Datensätze auf den Seiten. In der Praxis werden die Seiten deshalb nur zu einem bestimmten Grad (zum Beispiel 66%) gefüllt, damit beim Einfügen möglichst keine neuen Seiten erstellt werden müssen.

Bei der sortierten Speicherung ergibt sich beim Suchen das Problem, wo die Suche beginnen soll. Wünschenswert ist ein Verzeichnis (Index), welches Auskunft über die Position bestimmter Datensätze liefert. Eine solche Organisationsform bestehend aus Index- und Hauptdatei bezeichnet man als *index-sequentielle Dateiorganisation*. Auch in Indexdateien entsteht das Problem, wo die Suche beginnen soll. Um die Anzahl der Plattenzugriffe zu verringern, wird die Indexdatei selbst auch index-sequentiell organisiert. Man spricht von *mehrstufigen* Indizes. Abbildung 2.5 zeigt einen mehrstufigen Index für die Person. Im Idealfall besteht der Index höchster Stufe nur noch aus einer Seite.

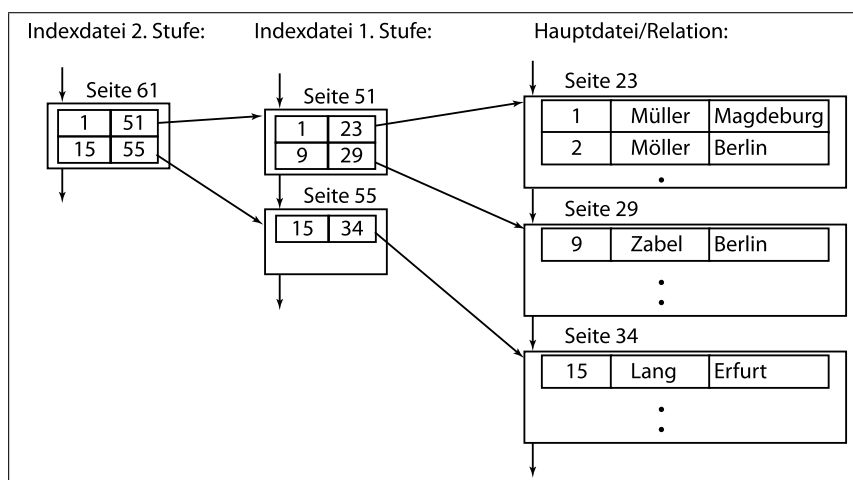


Abbildung 2.5: Index-sequentielle Speicherung

## Hash-Organisation

Datensätze werden bei der Hash-Organisation gestreut gespeichert. Die Speicheradresse (Seitennummer) eines Datensatzes wird durch eine *Hash-Funktion* über eine bestimmte Attributkombination berechnet. Da Hash-Funktionen im Allgemeinen nicht injektiv sind, kann für verschiedene Attributwerte der selbe Speicherort errechnet werden. Diese *Kollisionen* können dazu führen, dass die errechnete Seite zu klein für die zu speichernden Datensätze wird. In einem solchen Fall spricht man von (*Seiten*)*Überläufen*. Tritt ein solcher Überlauf ein, wird eine neue Seite in die entsprechende Datei aufgenommen und mit der Ursprungsseite verkettet. Einzelne Datensätze können bei der Hash-Organisation sehr schnell gefunden werden. Durch die Berechnung der Speicheradresse muß nur eine Seite (ggf. aber auch die Überlaufseiten) vom externen Speicher gelesen werden. Das Einfügen neuer Datensätze erfolgt ebenfalls mit geringem Aufwand. Im nächsten Kapitel wird näher auf Hash-Verfahren eingegangen.

## 2.2 Zugriffsverfahren

Nachdem die Datenspeicherung und die grundlegenden Dateiorganisationsformen vorgestellt wurden, sollen jetzt zunächst die Begriffe Zugriffspfad und Index definiert werden. Von nun an gilt die Definition von [SH99] S. 121:

„Mit *Zugriffspfad* bezeichnen wir jede über die grundlegende Dateiorganisationsform hinausgehende Zugriffsstruktur, also etwa jede Indexdatei über einer internen Relation.“

*Indexdateien/Indexe* sind Verzeichnisse mit Einträgen der Form  $(K, K\uparrow)$ .  $K$  entspricht dabei den Werten von Primär- oder Sekundärschlüsseln und wird im Folgenden als Suchschlüssel bezeichnet. Während *Primärschlüssel* innerhalb einer Relation eindeutig - also duplikatenfrei - sein müssen, gilt diese Forderung nicht für *Sekundärschlüssel*. Sekundärschlüssel weichen damit von der Schlüsseldefinition aus Abschnitt 2.1.1 ab.  $K\uparrow$  ist ein Adressverweis (zum Beispiel ein TID) auf einen Datensatz bei dem das Schlüsselattribut den Wert  $K$  hat. Sind Sekundärschlüssel nicht eindeutig, werden entweder mehrere  $(K, K\uparrow)$ - Paare  $((K, K\uparrow_1, \dots, (K, K\uparrow_n))$  gespeichert, oder  $K\uparrow$  entspricht einer Liste von Datensatzadressen. Neben dem Verweis auf die Datensatzadresse kann  $K\uparrow$  auch der Datensatz selbst sein. Ist dies der Fall, so ist der Zugriffspfad zu einer Dateiorganisationsform entartet.

Dass Indexe die Anfragebearbeitung deutlich beschleunigen können, wird in Abbildung 2.6 deutlich.

Die einfache SQL-Abfrage<sup>1</sup>:

```
select * from partsupp where ps_availqty > 3120 order by ps_suppkey
```

wurde ohne und mit einem Index über die Attribute „PS\_SUPPKEY“ und „PS\_PARTKEY“ (die Attribute bilden einen Primärschlüssel) gestellt. Die Anfragedauer, im Mittel von zehn Testläufen, wurde durch Verwendung des Indexes nahezu halbiert.

<sup>1</sup>Die Tabelle „PARTSUPP“ stammt aus dem TPC-H Benchmark, der im weiteren Verlauf der Arbeit noch näher beschrieben wird. Genutzt wurde DB2 Express-C.

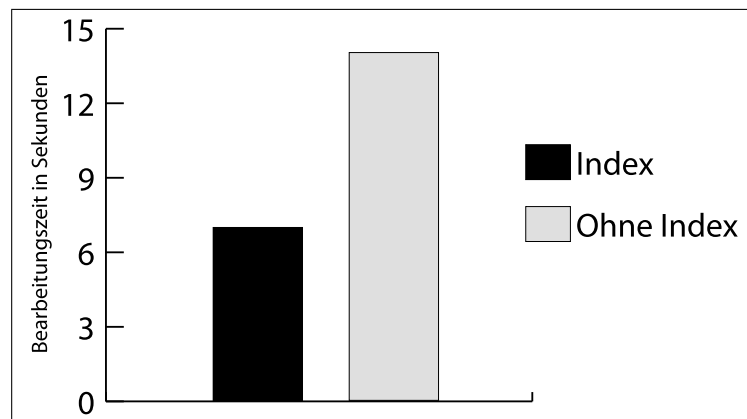


Abbildung 2.6: Abfragedauer mit und ohne Index (Eigene Messung)

Der Geschwindigkeitsvorteil ergibt sich aus der Sortierung des Indexes, die bei der Anfragebearbeitung ausgenutzt wird. Da Indexe beim Einfügen, Löschen und Modifizieren von Datensätzen in der zugrunde liegenden Relation aktualisiert werden müssen, ist der Aufwand der genannten Operationen bei der Existenz eines Indexes entsprechend höher. Im Fall des hier verwendeten Indexes ist die Dauer einer Einfügeoperation um etwa 50% gestiegen. Die Größe der erstellten Indexdatei beträgt mit rund 30 Megabyte etwa ein Fünftel der PARTSUPP-Tabelle.

Im weiteren Verlauf der Arbeit wird in Anlehnung an [SH99] der Begriff *Zugriffsstruktur* verwendet, wenn zwischen Dateiorganisationsform und zusätzlichen Zugriffspfaden nicht weiter unterschieden werden soll.

Im nächsten Gliederungspunkt („Klassifikation von Zugriffsverfahren“) werden verschiedene Indexarten und Indexvarianten beschrieben. Anschließend daran werden gängige Zugriffsverfahren wie zum Beispiel B-Bäume und Hash-Verfahren präsentiert.

### 2.2.1 Klassifikation von Zugriffsverfahren

In diesem Abschnitt sollen einige der in [SH99] Seite 118 ff. und [SB03] Seite 77 ff. vorgestellten Merkmale von Zugriffsstrukturen behandelt werden.

#### Schlüsselzugriff und Schlüsseltransformation

Die Ermittlung der Adresse von Datensätzen mit bestimmten Attributs- beziehungsweise Schlüsselwerten kann auf zwei verschiedene Arten erfolgen. Beim *Schlüsselzugriff* werden Schlüsselwerte und Datensatzadresse einander zugeordnet und in Hilfsstrukturen wie zum Beispiel einer Indexdatei gespeichert. Die Größe der Datei hängt dabei vor allem von Schlüsselgröße und Größe des Adressverweises ab.

Bei der *Schlüsseltransformation* werden Datensatzadressen anhand der Schlüsselwerte berechnet. Anstatt der Schlüsselwert-Adressverweis-Paare wird nur eine Berechnungsvorschrift gespeichert. Bei der bereits behandelten Hash-Organisation von Dateien findet eine solche Schlüsseltransformation statt. Mit den *Hash-Verfahren* werden in Kapitel 2.2.3 typische Vertreter der Schlüsseltransformation behandelt.

## Primär- und Sekundärindexe

*Primärindexe* sind Zugriffspfade die die Dateiorganisationsform, wie zum Beispiel die Sortierung nach einem Attribut, der zugrunde liegenden Relation ausnutzen können (index-sequentielle Speicherung). In der Regel werden Primärindexe über dem Primärschlüssel der Relation definiert. Der Suchschlüssel  $K$  des Indexes nimmt in diesem Fall die Attributwerte des Primärschlüssels als (Such-)Schlüsselwerte an und muß nicht mit doppelten Schlüsselwerten rechnen. Wie später zu sehen ist, können Primärindexe *geclustert* und *dünnbesetzt* sein. Pro Relation kann es maximal einen Primärindex geben.

*Sekundärindexe* sind alle weiteren (indexiert-nichtsequentiellen) Zugriffspfade, die für eine Relation definiert sind. Prinzipiell können Sekundärindexe für jede mögliche Kombination von Attributen einer Relation angelegt werden. Durch den bereits erwähnten Speicherplatzbedarf und die benötigte Aktualisierung der Indexe beim Einfügen, Löschen und Modifizieren von Datensätzen ist dies aber nicht sinnvoll.

## Dünnbesetzte und Dichtbesetzte Indexe

Um die Anzahl der zu speichernden Schlüsselwert-Adress-Paare  $(K, K \uparrow)$  zu verringern, kann man Indexe *dünnbesetzt* speichern. Anstatt Adressverweise für alle Datensätze einer Relation zu speichern, gibt es in dünnbesetzten Indexen nur einen Eintrag pro Seite, der sich auf den jeweiligen *Seitenanführer* bezieht. Verweist das Paar  $(K_1, K_1 \uparrow)$  auf den ersten Datensatz einer Seite, so verweist der Folgeeintrag  $(K_2, K_2 \uparrow)$  auf den ersten Datensatz der nachfolgenden Seite. Dünnbesetzte Indexe sind nur möglich, falls die zugrundeliegende Relation *sortiert* beziehungsweise *sequentiell* (siehe S. 9) nach dem Schlüsselwert  $K$  gespeichert ist. Damit sind dünnbesetzte Indexe gemäß der obigen Definition nur als Primärindexe realisierbar. Ein Datensatz mit dem Schlüsselwert  $K_?$ , für den  $K_1 \leq K_? < K_2$  gilt, ist auf der Seite von  $K_1 \uparrow$  zu finden. Abbildung 2.7 veranschaulicht das Prinzip von dünn- und dichtbesetzten Indexen.

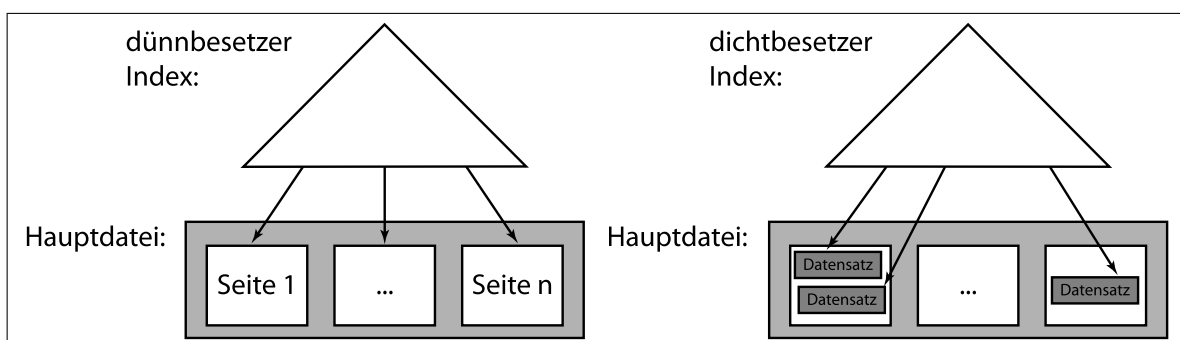


Abbildung 2.7: Dünn- und Dichtbesetzter Index ([SB03] S. 91)

Ein *dichtbesetzter Index* speichert für jeden Datensatz in einer Relation  $(K, K \uparrow)$ -Einträge. Da im Normalfall mehrere Datensätze auf einer Seite gespeichert sind, gilt folgende Formel:

*Anzahl von Einträgen in dichtbesetzten Indexen =*

*Anzahl von Einträgen in dünnbesetzten Indexen  $\times$  Anzahl von Datensätzen pro Seite*

Ein Index muß dichtbesetzt sein, wenn die entsprechende Relation nicht nach dem Schlüsselattribut des Indexes sortiert gespeichert ist. Durch die fehlende Sortierung der

Datensätze auf den Seiten, kann der Vergleich von Seitenanführern benachbarter Seiten, anders als noch bei dünnbestzten Indexen, keine Aussagen über die Schlüsselwerte „zwischen“ ihnen treffen. Ist eine Relation unsortiert gespeichert, zum Beispiel als Heap-Datei, so muß demnach auch ein Primärindex dichtbesetzt sein. Sekundärindexe sind hingegen immer dichtbesetzt. Um den Platzbedarf eines dichtbesetzten Indexes zu verringern, kann man mehrfach vorhandene Schlüsselwerte in einem Indexeintrag speichern. Der Adressverweis  $K^\uparrow$  ist dann eine Liste von Datensatzadressen.

Das Speichern aller Schlüsselwerte ermöglicht die Abarbeitung bestimmter Anfragen ohne Zugriff auf die Relation selbst. Zum Beispiel kann die Ermittlung der Anzahl von Datensätzen mit dem Schlüsselwert  $n$  allein durch die Suche in der Indexdatei geschehen.

### Geclusterte und nicht-geclusterte Indexe

Ein weiteres Klassifikationsmerkmal ist die Form der Sortierung eines Indexes beziehungsweise der Indexdatei. Es sollen an dieser Stelle die zwei verschiedenen Arten vorgestellt werden. Ein *geclustertes Index* liegt vor, wenn der Index die gleiche Sortierung aufweist wie die Relation, auf die er verweist. Ist die Relation Person nach dem Attribut ID sortiert gespeichert, ist ein Index über dem Attribut ID geclustert, wenn die Einträge der Indexdatei ebenfalls nach dem Attribut ID sortiert gespeichert werden. *Nicht-geclusterte* Indexe stellen keine Anforderungen an die Sortierung der zugrundeliegenden Relation. Ein Index über dem Attribut Name der Person-Datei ist nicht-geclustert, denn die Relation ist in der Datei nach dem Attribut ID sortiert gespeichert.

Wegen der gleichen Sortierung unterstützen geclusterte Indexe Bereichsanfragen über das Schlüsselattribut sehr gut. Abbildung 2.8 zeigt die in [SB03] S. 92 ff festgestellten Unterschiede der Anfragebearbeitungsgeschwindigkeit für *exact-match queries* am Beispiel von DB2. Dargestellt wird das Durchsatzverhältnis untereinander. Der geclusterte Index entspricht dabei der 1.

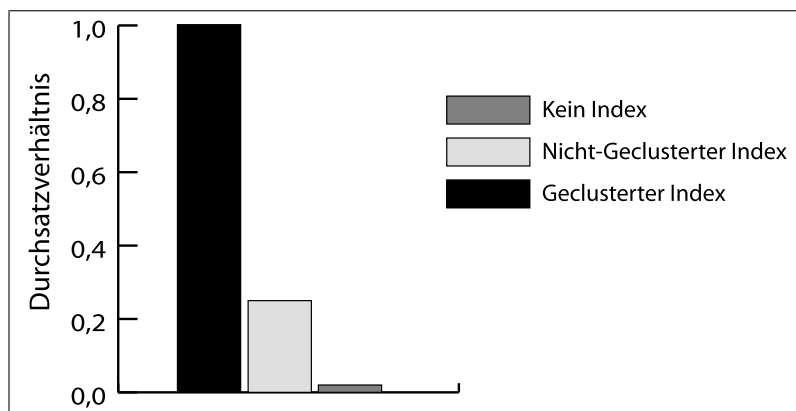


Abbildung 2.8: Index Clustering ([SB03] S. 93)

### Ein-Attribut- und Mehr-Attribut-Indexe

Der Großteil der bisher angeführten Beispielindexe bezog sich auf ein einziges Attribut der entsprechenden Relation. Ein solcher Index unterstützt den Datensatzzugriff nur über dieses einzelne Attribut und wird als *Ein-Attribut-Index* bezeichnet.

Im Zusammenhang mit Abbildung 2.6 auf Seite 11 wurde ein Index über zwei Attribute vorgestellt. Man nennt diese Art von Indexen, die über mindestens zwei Attributen definiert sind, *Mehr-Attribut-Indexe*. Sind bestimmte Attributkombinationen in Relationen eindeutig (duplikatenfrei), kann diese Eindeutigkeit in Mehr-Attribut-Indexen ausgenutzt werden. Wie gut einzelne Anfragen durch Zwei-Attribut-Indexe unterstützt werden, hängt von deren *Dimensionalität* ab, die im Folgenden behandelt wird.

### Eindimensionale und mehrdimensionale Zugriffsstrukturen

In *eindimensionalen* Zugriffsstrukturen definieren die Werte eines *einzigsten* Zugriffs- oder Schlüsselattributes eine lineare Ordnung in einem eindimensionalen Raum (s. [SH99] S. 125 ff). Dieser Raum wird dann bei Anfragen nach passenden Werten durchsucht. Da Ein-Attribut-Indexe über nur ein Attribut definiert sind, sind sie immer eindimensional.

Mehr-Attribut-Indexe können ebenfalls eindimensional gespeichert werden. Die einzelnen Attributwerte werden in diesem Fall zusammengesetzt und als Zugriffsattribut betrachtet, welches die lineare Ordnung in einem eindimensionalen Raum definiert. Bei der Zusammensetzung einzelner Attributwerte spielt die Reihenfolge eine wichtige Rolle in Bezug auf die Anfragebearbeitung, da sie die Ordnung/Sortierung vorgibt. Ein Telefonbuch ermöglicht beispielsweise die schnelle Suche von Telefonnummern von Personen in bestimmten Orten, was mit einem eindimensionalen Mehr-Attribut-Index über die Attribute Ort und Name vergleichbar ist. Eine *exact-match* Suche (Suche nach Ort *und* Name) führt schnell zum Erfolg. Erfolgt die Suche nach nur einem Attribut (*partial-match*), also entweder Ort oder Name, kann im Falle des Namens keine Sortierung ausgenutzt werden und die ganze Indexdatei muß eingelesen<sup>2</sup>, beziehungsweise das ganze Telefonbuch durchblättert werden.

In *mehrdimensionalen* Zugriffsstrukturen spannen die Zugriffsattribute einen mehrdimensionalen Raum auf. Bei *exact-match queries* bilden einzelne Punkte im Raum die Anfrageergebnisse beziehungsweise die Adressverweise. Mehrdimensionale Zugriffsstrukturen unterstützen *partial-match queries* für alle Attributkombinationen.

### Statische und Dynamische Zugriffsstrukturen

Das letzte hier behandelte Unterscheidungsmerkmal von Zugriffsstrukturen bezieht sich auf das Verhalten von Zugriffsstrukturen bei Wachstum und Schrumpfung der Anzahl zu verwaltender Datensätze. Unterschieden werden *statische* Zugriffsstrukturen und *dynamische* Zugriffsstrukturen.

Statische Zugriffsstrukturen bieten nur bei einer gleichbleibenden Anzahl von Datensätzen optimale Zugriffsunterstützung. Bei einer stark wachsenden, oder aber auch stark schrumpfenden, Anzahl von Datensätzen verringert sich der Beitrag zur Anfragebeschleunigung. Bei Schlüsseltransformationsverfahren mit unveränderter Berechnungsvorschrift führt die Erhöhung der Datensatzanzahl zu ungewünschten Seitenüberläufen. Im Gegenzug dazu kommt es bei schrumpfenden Datensatzmengen zu schlecht ausgelasteten Seiten. In beiden Fällen wird die Anzahl der benötigten Plattenzugriffe unnötig erhöht. In der Regel sind statische Zugriffsstrukturen im Datenbankbereich ungeeignet.

<sup>2</sup>s. Heap- und sequentielle Organisation: Kapitel 2.1.3 Dateiorganisationsformen

*Dynamische* Zugriffsstrukturen verwalten Datensätze, unabhängig von deren Anzahl, optimal. Bei Transformationsverfahren wird der Bildbereich der Berechnungsvorschrift an die Anzahl der Datensätze angepasst. Dynamische Schlüsselzugriffsverfahren passen sich, wie später zu sehen, ebenfalls an die Datensatzanzahl an.

### Einordnung konkreter Zugriffsverfahren

Bevor in den nächsten Abschnitten B-Bäume und Hash-Verfahren beschrieben werden, sollen in Tabelle 2.2 gängige Zugriffsverfahren in Bezug auf einige der behandelten Merkmale eingeordnet werden.

Verfahrensart	Dynamik	Dimensionalität	Verfahren
Schlüsseltransformation	statisch	eindimensional mehrdimensional	klassisches Hashen
	dynamisch	eindimensional mehrdimensional	lineares Hashen ... mehrdimensionales Hashen
Schlüsselzugriff	statisch	eindimensional  mehrdimensional	indexsequentiell indexiert-nichtsequentiell mehrstufig indexsequentiell mehrst. indexiert-nichtseq.
			dynamisch
Grid-Files	dynamisch	mehrdimensional	Grid-Files

Tabelle 2.2: Einordnung von Zugriffsverfahren ([SH99] S. 130)

### 2.2.2 B-Bäume

Nachdem verschiedene Arten und Merkmale von Zugriffsverfahren vorgestellt wurden und auch schon eine Einordnung der gängigen Techniken stattfand, soll jetzt das wichtigste Zugriffsverfahren aktueller Datenbanksysteme, der *B-Baum*, in einigen seiner Varianten vorgestellt werden. Grundlegende Kenntnisse über Bäume in der Informatik werden an dieser Stelle vorausgesetzt.

#### Definition

Die von Bayer [BM72] eingeführten B-Bäume dienen in Datenbanken als dynamische Schlüsselzugriffsverfahren für Primär- und Sekundärindexe. Wegen ihrer Baumstruktur entsprechen sie den mehrstufigen Indexverfahren. B-Bäume passen dabei die Stufenzahl, die sich in der *Höhe* eines B-Baumes widerspiegelt, an die Anzahl der gespeicherten Datensätze an. Speichert man in den Knoten eines B-Baumes anstatt der  $(K, K\uparrow)$ -Paare ganze Datensätze, so wird er zur Dateiorganisationsform.

Um eine effiziente Suche zu ermöglichen, werden die Schlüsselwerte *sortiert* gespeichert (*Suchbaum*). Die Indexeinträge werden bei B-Bäumen in den Knoten gespeichert.

Die Größe eines Knotens wird dabei an die Seitengröße des Datenbanksystems angepasst. Neben den Indexeinträgen müssen in den Knoten (ausgenommen den Blattknoten) auch Zeiger auf die Kinderknoten gespeichert werden. Knoten mit  $i$  Einträgen haben  $i+1$  Folgeknoten. B-Bäume sind *balanciert*. Das bedeutet, dass in einem B-Baum alle Pfade von der Wurzel zu den Blättern gleich lang sind. Auch nach dem Einfügen oder Löschen von Datensätzen bleibt diese Balancierung erhalten. Das dafür von Bayern eingeführte Balancierungskriterium besagt, dass jeder Baumknoten, mit Ausnahme der Wurzel, mindestens  $m$  und maximal  $2m$  Elemente (Datensätze<sup>3</sup>) enthalten muss. Aus dieser Forderung ergibt sich eine Knotenauslastung von mindestens 50%, die durch das Einpassen der Knoten auf Seiten auch für diese gilt. Die Konstante  $m$  bezeichnet man als *Ordnung* des B-Baumes und ergibt sich aus Seitengröße, Datensatzgröße der Indexeinträge und der Größe der Zeiger auf die Folgeknoten. Abbildung 2.9 zeigt schematisch den Aufbau eines 3-stufigen B-Baumes der Ordnung 1.

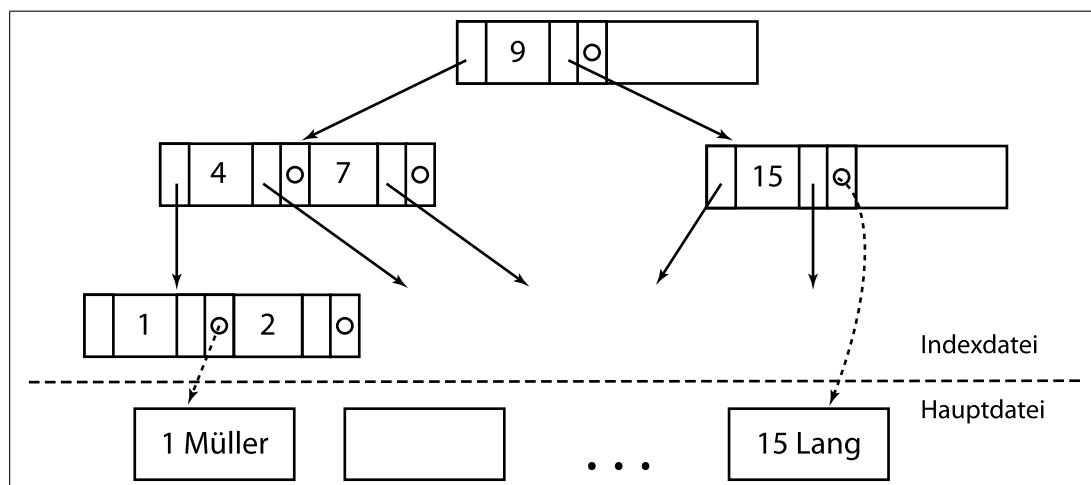


Abbildung 2.9: Prinzipieller Aufbau eines B-Baumes (vgl. [SH99] S. 149)

An der gezeigten Abbildung lassen sich die Eigenschaften ablesen, die ein B-Baum erfüllen muss und hier zusammengefasst gemäß [SH99] S. 142 ff noch einmal aufgelistet werden:

1. Jede Seite enthält höchstens  $2m$  Elemente.
2. Jede Seite, ausser der Wurzelseite, enthält mindestens  $m$  Elemente.
3. Jede Seite ist entweder eine Blattseite ohne Nachfolger oder hat  $i+1$  Nachfolger, falls  $i$  die Anzahl ihrer Elemente ist.
4. Alle Blattseiten liegen auf der gleichen Stufe.

### Operationen in B-Bäumen

Die *Suche* von Schlüsselwerten in B-Bäumen erfolgt von der Wurzel an und endet, spätestens auf der Blattebene. Der gesuchte Schlüsselwert  $k$  wird mit den Schlüsselwerten  $K_1$ ,

<sup>3</sup>Es sei noch einmal erwähnt, dass Indexeinträge ebenfalls als Datensätze gespeichert werden

$K_2, K_3, \dots, K_n$  der Wurzel vergleichen. Wird der Wert dabei gefunden, ist die Suche beendet. Wird der Wert nicht gefunden wird die Suche auf einem der nachfolgenden Knoten fortgesetzt. Die Auswahl des neuen Knotens geschieht über die gespeicherten Zeiger und erfolgt dabei wie folgt:

- Falls  $k < K_1$ , folge dem ersten Zeiger.
- Für alle  $i < n$ : Falls  $K_i < k < K_{i+1}$ , folge dem  $i+1$ -ten Zeiger.
- Falls  $K_n < k$ , folge dem letzten Zeiger.

Wird  $k$  auch auf der neuen Seite nicht gefunden, wiederholt sich der Suchvorgang bis hin zu den Blattknoten. Wird der Schlüsselwert auf der Blattebene nicht gefunden, ist er nicht im durchsuchten Baum vorhanden und die Suche wird beendet. Wendet man dieses Verfahren für  $k=1$  auf den Baum in Abbildung 2.9 an, wird ersichtlich, dass für die Suche eines einzelnen Wertes maximal drei Seiten gelesen werden müssen. Dies entspricht der Höhe des Baumes und gilt für alle B-Bäume. Allgemeiner formuliert ist bei einem B-Baum der Ordnung  $m$  und einer Hauptdatei mit  $n$  gespeicherten Datensätzen der Zugriff auf die richtige Seite mit maximal  $\log_m(n)$  Seitenzugriffen möglich.

Dem *Einfügen* eines neuen Eintrages  $k$  in den Baum geht zunächst die Suche des richtigen Blattknotens voraus. Kommt es durch das Einfügen zum *Seitenüberlauf*, also  $2m + 1$  Einträge auf der entsprechenden Seite, wird die Knotenstruktur möglichst lokal geändert. Die letzten  $m$  Einträge des Ursprungsknoten werden auf einen neu angelegten Knoten verschoben, während die ersten  $m$  Einträge unverändert bleiben. Der mittlere Eintrag  $m+1$  des Ursprungsknoten wird nach oben weitergereicht. Führt die Aufnahme des mittleren Eintrages erneut zu einem Überlauf, wird das Verfahren rekursiv fortgesetzt. Führt das Einfügen zu keinem Überlauf wird  $k$  in den entsprechenden Knoten einsortiert und das Einfügen ist beendet.

Auch dem *Löschen* eines Eintrages  $k$  geht eine Suche voraus. Ist  $k$  auf einer Blattseite gespeichert, wird der entsprechende Datensatz gelöscht. Nach dem Entfernen von  $k$  muss eine *Unterlaufbehandlung* durchgeführt werden, falls die Anzahl der Einträge  $n$  kleiner ist als die Ordnung  $m$  des Baumes. Befindet sich  $k$  auf einem inneren Knoten  $N$ , tritt der, der Sortierung entsprechende, nächstkleinere Eintrag eines der Kindknoten  $N$ 's an die Position des gelöschten  $k$ -Eintrages. Kommt es auf dem abgebenden Kindknoten zum Unterlauf, wird wie folgt verfahren.

Die *Unterlaufbehandlung* erfolgt durch einen *Ausgleich* mit einem benachbarten Knoten oder dem *Zusammenlegen* zweier Knoten zu einem. Hat einer der benachbarte Knoten mehr als  $m$  Einträge, kommt es zum Ausgleich. Der größte beziehungsweise kleinste Eintrag des Nachbarknotens steigt dabei eine Ebene auf und ersetzt dort den nächstgrößeren beziehungsweise nächstkleineren. Der verdrängte Satz wird dem unterlaufenen Knoten hinzugefügt. Kann keine der Nachbarseiten Einträge abgeben, wird die Unterlaufseite mit einer Nachbarseite zusammengelegt. Zusätzlich wird der „mittlere“ Eintrag des Vaterknotens auf den neuen Knoten, der dann  $2m$  Einträge besitzt, verschoben. Führt dieses Verschieben zu einem Unterlauf, wird das Verfahren für den unterlaufenen Knoten durchgeführt.

Der maximale Aufwand von Suche, Einfügen und Löschen beträgt in B-Bäumen immer  $O(\log_m(n))$  Operationen und entspricht damit der Höhe des Baumes (vgl. [SH99] S.

147). Anschauliche Beispiele für die hier gezeigten Operationen finden sich zum Beispiel in [GMUW00] S. 159 ff.

### B<sup>+</sup>-Bäume

Der *B<sup>+</sup>-Baum* ist eine Variante des B-Baumes, bei der die Datensätze einer Datei in den Blättern gespeichert werden. Anders als noch in B-Bäumen werden in den inneren Knoten nur noch Schlüsselwerte und keine Datensatzadressen gespeichert. Schlüsselwerte der inneren Knoten finden sich auch in den Blättern wieder, wo sie zusammen mit dem zugehörigen Datensatz gespeichert vorliegen. Zeiger auf Kindknoten sind natürlich auch in den inneren Knoten abgelegt. Durch diese Art der Speicherung unterscheidet sich die Größe von inneren Knoten und Blättern, weshalb die Ordnung in B<sup>+</sup>-Bäumen als Wertepaar  $(x, y)$  definiert ist.  $x$  entspricht der Mindestbelegung der inneren Knoten und  $y$  die der Blätter. Blätter haben zusätzliche Zeiger auf Vorgänger- und Nachfolgeblatt. Da ein B<sup>+</sup>-Baum Datensätze sortiert speichert, entspricht die Hauptdatei der in Kapitel 2.1.3 vorgestellten sequentiellen Organisationsform. Weil ein B<sup>+</sup>-Baum gleichzeitig einen Index zur Seitensuche verkörpert, wird klar, dass B<sup>+</sup>-Bäume mehrstufige indexsequentielle Dateien darstellen und als Primärindexe geeignet sind. Baumebenen entsprechen dabei den Indexstufen und die Knoten den Indexseiten (s. Abbildung 2.5. Die oben stehende Betrachtung gilt nur im Falle der Verwendung als Primärindex. Ein als Sekundärindex eingesetzter B<sup>+</sup>-Baum speichert in der Blattebene die  $(K, K\uparrow)$ -Einträge der Indexdatei.

Such- und Einfügeoperationen weichen kaum von denen des B-Baumes ab. Die Suche erfolgt in B<sup>+</sup>-Bäumen allerdings immer bis zur Blattebene. Das Löschen von Datensätzen geschieht wesentlich effizienter als noch in B-Bäumen, weil es sich normalerweise nur auf die Blattebene auswirkt. Schlüsselwerte verbleiben nämlich in den inneren Knoten, auch wenn es keine Datensätze mehr mit diesem Schlüsselwert gibt. Es kommt dadurch in B<sup>+</sup>-Bäumen zu weniger Seitenunterläufen.

Der B<sup>+</sup>-Baum ist die in der Praxis gebräuchlichste B-Baum-Variante (siehe [SH99] S. 149).

Weitere Informationen zum Verhalten von B-Bäumen in der Praxis und weitere, zum Teil von B-Bäumen abweichende, Baumverfahren wie beispielsweise *B<sup>\*</sup>-Bäume*, *B#-Bäume* und *Tries* finden sich in [SH99] S. 150 ff.

### 2.2.3 Hash-Verfahren

Die den Hash-Verfahren zugrunde liegende Idee der Adressberechnung durch Schlüsselwerte (Schlüsseltransformation) über eine *Hash-Funktion* wurde bei der Vorstellung der Hash-Organisation auf Seite 10 im Abschnitt 2.1.3 (Dateiorganisationsformen) bereits kurz erläutert. Ein *Hash-Index* speichert die  $(K, K\uparrow)$ -Einträge eines Indexes in einer hash-organisierten Struktur (*Hash-Table*) beziehungsweise Indexdatei. Streng genommen sind Hash-Indexe immer Sekundärindexe, da hash-organisierte Dateien keine separaten Primärindexstrukturen benötigen. Hash-organisierte Dateien haben durch die Hash-Funktion sozusagen automatisch einen Primärindex, der den direkten Datenzugriff erlaubt (vgl. [SKS97] S. 362). Werden Hash-Indexe über mehrere Attribute definiert, kann der Zugriff nur mit allen Schlüsselteilen erfolgen.

Allgemein werden beim Hashen Datensätze anhand des gewählten Schlüsselattributs

den sogenannten *Buckets* (deutsch Eimer) zugeordnet, die aus einer oder mehreren Seite(n) bestehen und einzeln adressierbar sind. Die Zuordnung der Schlüsselwerte  $K$  zu den Buckets  $B$  erfolgt über eine *Hash-Funktion*  $h(K_i)$ . Der Wertebereich der Hash-Funktion impliziert die möglichen Bucketadressen. Da in der Regel nicht für jeden möglichen Wert  $K_i$  ein einzelner Bucket angelegt wird, kann für die Schlüsselwerte  $K_1 \neq K_2$  in Hash-Verfahren  $h(K_1) = h(K_2)$  gelten. In einem solchen Fall spricht man von einer *Kollision*. Im Folgenden sollen wieder die Operationen für Einfügen, Suche und Löschen beschrieben werden. Ausgegangen wird an dieser Stelle von einem *statischen* Hashverfahren. Statisch bedeutet hierbei, dass sich weder die Anzahl der Buckets  $B$  noch die Hash-Funktion  $h(K)$  selbst verändert. Eine echte Anpassung an sich verändernde Datenmengen findet also nicht statt.

Beim Einfügen eines Datensatzes mit dem Schlüsselwert  $k$ , wird durch die Hash-Funktion zunächst der Bucket  $B_i = h(k)$  bestimmt. Der Datensatz wird anschließend in  $B_i$  gespeichert. Bietet  $B_i$  nicht mehr ausreichend Platz für die Speicherung des Datensatzes, wird eine *Überlaufseite* angelegt und mit  $B_i$  verkettet. Die Überlaufseite ist für die Hash-Funktion „unsichtbar“ und kann damit nicht direkt adressiert werden.

Die *Suche* nach einem Schlüsselwert  $k$  ist denkbar einfach. Anhand der Hash-Funktion wird der Bucket  $B_i = h(k)$  bestimmt, auf dem der Schlüsselwert zu finden ist. Die  $B_i$  entsprechenden Seiten werden eingeladen und durchsucht. Ist  $B_i$  überlaufen, müssen die verketteten Überlaufseiten ebenfalls durchsucht werden.

Dem *Löschen* von Datensätzen geht (wie immer) eine Suche, bei der wieder  $B_i$  ermittelt wird, voraus. Ist nach dem Löschen des Datensatzes eine Überlaufseite leer, so kann diese aus der Verkettung entfernt werden.

Die folgende Abbildung veranschaulicht noch einmal das Prinzip des (statischen) Hashens mit der Hash-Funktion  $h(k) = k \bmod 3$  und 3 aus  $h(k)$  resultierenden Hash-Buckets. Der gesuchte Schlüsselwert  $k = 9$  ist im gezeigten Beispiel bereits auf einer Überlaufseite des Buckets  $B_0$  gespeichert.

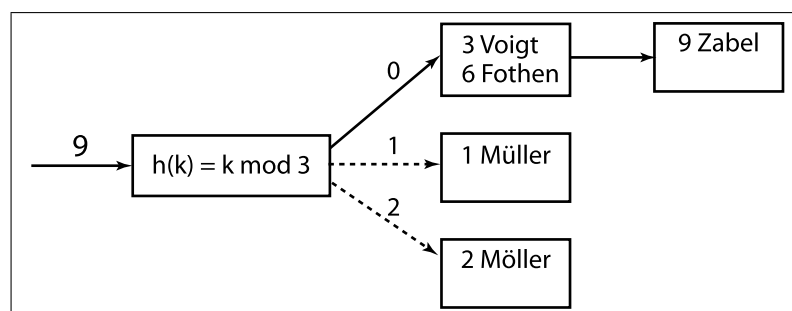


Abbildung 2.10: Statisches Hashen

Um die Anzahl der Kollisionen zu verringern, ist es möglich, die Hash-Funktion so zu verändern, dass mehr Buckets adressiert werden können. Mit  $h(k) = k \bmod 4$  käme es im Beispiel von Abbildung 2.10 zu keinem Überlauf. Da jedoch beim Ändern der Hash-Funktion alle Datensätze neu verteilt werden müssen, ist dieses Vorgehen wenig nützlich. Wünschenswert ist eine „feste“ Hash-Funktion, deren Bildbereich sich dynamisch verändert. Einige dieser *dynamischen* Verfahren wie zum Beispiel *lineares* und *erweiterbares* Hashen werden in [GMUW00] S. 177 ff und [SKS97] S. 362 ff beschrieben. Weitere Hash-Techniken werden in [Knu03] S. 513 vorgestellt und analysiert.

### 2.2.4 Bitmap-Indexe

B-Bäume und Hash-Verfahren bieten im Fall vieler gleicher Schlüsselwerte nur suboptimale Zugriffsunterstützung. Die vor allem in Data-Warehouse-Systemen [AB04] eingesetzten *Bitmap-Indexe* nehmen sich dieses Problems an. Für die verschiedenen Attributwert wird je ein *Bitvektor*, daher auch die Bezeichnung *Bit Map*, erstellt. Mit 0 und 1 wird dann markiert, welchen Wert ein bestimmter Datensatz hat. Abbildung 2.11 zeigt einen Bitmap-Index für das Attribut Geschlecht. Angezeigt werden zwei männliche und eine weibliche Person.

TID	männlich	weiblich
1	1	0
2	0	1
3	1	0
...	...	...

Abbildung 2.11: Einfacher Bitmap Index

Bitmap-Indexe haben verhältnismäßig wenig Speicherplatzbedarf, da nur TID's und Bitwerte gespeichert werden. Besonders effizient arbeiten Bitmap-Indexe bei Verbundoperationen (*joins*). Hier werden die Bitvektoren beider Indexe jeweils durch einfache boolesche Operationen verglichen, um übereinstimmende Schlüsselwerte zu finden. Für Attribute mit vielen verschiedenen Werten sind Bitmap-Indexe weniger geeignet, da für jeden verschiedenen Wert ein Bitvektor angelegt werden muss. Kommen neue Schlüsselwerte hinzu, muss ein neuer Bitvektor erzeugt und für alle vorhanden Datensätze gepflegt werden.

### 2.2.5 Mehrdimensionale Verfahren

[GMUW00] S. 188 ff beschreibt die Notwendigkeit mehrdimensionaler Zugriffsverfahren am Beispiel *geographischer Informationssysteme* anhand der folgenden Abfragen. Wie schon in Kapitel 2.2.1 wird in mehrdimensionalen Strukturen im Folgenden von Punkten anstatt Datensätzen gesprochen.

- *Partial-match queries*: Für Suchwerte einer oder mehrerer Dimension(en) werden alle Punkte gesucht, die den Suchwert(en) dieser Dimensionen entsprechen.
- *Range queries*: Für Bereiche einer oder mehrerer Dimension(en) werden die Punkte innerhalb des Suchraumes gesucht.
- *Nearest-neighbor queries*: Gesucht werden Punkte, die sich im Umfeld (der Nachbarschaft) eines bestimmten Punktes befinden.
- *Where-am-I (deutsch Wo bin ich) queries*: Für einen gegebenen Punkt wird gefragt, in welcher übergeordneten Struktur er sich befindet. Ein solche Anfrage könnte beispielsweise lauten: „Befindet sich Magdeburg in den Grenzen Sachsen-Anhalts“.

Zu den in [GMUW00] beschriebenen mehrdimensionalen Verfahren zählen *Grid-Files*, *kd-Bäume*, *mehrdimensionale Hash-Verfahren* und *R-Bäume*. An dieser Stelle soll es bei der Aufzählung der genannten mehrdimensionaler Strukturen bleiben und auf die angegebene Literatur verwiesen werden.

## 2.2.6 Indexverwendung

Zum Abschluss des Abschnittes Zugriffsverfahren soll der Einsatz von Indexen zur Anfragebeschleunigung in Datenbanksystemen zusammenfassend beschrieben werden. Ausgangspunkt wird dabei von *B-Bäumen*.

Der Index als Verfahren zur Unterstützung des effizienten Zugriffs auf Datensätze wurde bereits beschrieben. Für die Suche nach Datensätzen mit bestimmten Attributwerten, wird die Datensatzposition anhand des Indexes ermittelt. Dieses Verfahren kann jedoch ineffizient werden, wenn die Menge der angefragten Datensätze einem Großteil der Relation entsprechen. In einem solchen Fall kann die Relation komplett gelesen werden (Table-Scan) und auf kostenverursachende Indexzugriffe verzichtet werden. Die Entscheidung über die Verwendung von Indexen wird bei der Anfrageoptimierung (siehe Abschnitt 2.3.1 getroffen).

Die Sortierung von Indexen kann Anfragen ebenfalls beschleunigen, da das explizite Sortieren des Anfrageergebnisses entfallen kann. Besonders bei großen Relationen kann das Sortieren nicht allein im Hauptspeicher erfolgen und Zwischenergebnisse müssen auf Platte zwischengespeichert werden (externes Sortieren). Bei Verbundoperationen wirken sich Indexe ebenfalls positiv auf die Anfragebearbeitung aus (siehe Abschnitt 2.3.2). Durch sortierte Indexe kann hier der besonders effiziente *Merge-Join* ohne Sortierphase durchgeführt werden.

Einen negativen Effekt haben Indexe bei Einfüge-, Lösch- und Änderungsoperationen, weil sie, den Änderungen entsprechend, aktualisiert werden müssen. Da bei Lösch- und Änderungsoperationen zunächst eine Suche nach den betroffenen Datensätzen erfolgt, kann der Index den durch die Aktualisierung entstehenden Aufwand ausgleichen. Beim Einfügen neuer Datensätze kann der Aktualisierungsaufwand hingegen weniger gut ausgeglichen werden (siehe [SB03] S. 108) und die Bearbeitungsgeschwindigkeit sinkt.

Tabelle 2.3 zeigt die unterstützten Verfahren einiger Datenbanksysteme. Als grundlegende Dateiorganisationsform wird von allen genannten Systemen die sequentielle und die Heap-Speicherung unterstützt.

System	Dateiorganisation	Indexverfahren
IBM DB2 UDB V7.1	B-Baum (dichtbesetzt)	B-Baum
Oracle 9i EE	B-Baum (dicht), Hash-Org.	B-Baum, Bitmap, Funktionen
SQL Server 7	B-Baum (dünnbesetzt)	B-Baum
Sybase	B-Baum (dünnbesetzt)	B-Baum

Tabelle 2.3: Zugriffsstrukturen einiger DBS ([SB03] S. 108)

## 2.3 Anfrageverarbeitung

Im bisherigen Verlauf dieser Arbeit wurden bereits einige Beispiele von konkreten, in *SQL* verfassten, Datenbankabfragen aufgeführt. Dieser Abschnitt soll nun beschreiben, wie *SQL*-Abfragen in relationalen Systemen verarbeitet werden. Die dabei entstehenden *Zugriffspläne* sind in den folgenden Kapiteln von besonderer Bedeutung, da aus ihnen Informationen über den Bearbeitungsaufwand von Abfragen gewonnen werden können.

Abbildung 2.12 zeigt den prinzipiellen Ablauf der Abfragebearbeitung. In den ersten beiden Schritten *Übersetzung* und *Sichtauflösung* werden *SQL*-Abfragen in die relationale Algebra übersetzt und Sichten<sup>4</sup> werden durch ihre entsprechenden Abfragen ersetzt. Bereits an dieser Stelle entsteht ein möglicher (unoptimierter) Zugriffsplan für die Abfrage. Bei der *Optimierung* versucht man, einen bessern Plan, mit dem gleichen Ergebnis, zu finden. Die dabei entstehenden Alternativen werden bewertet und der günstigste Plan wird ausgewählt. Der gewählte Plan wird anschließend bei der *Code-Erzeugung* in ausführbaren Code umgewandelt und ausgeführt.

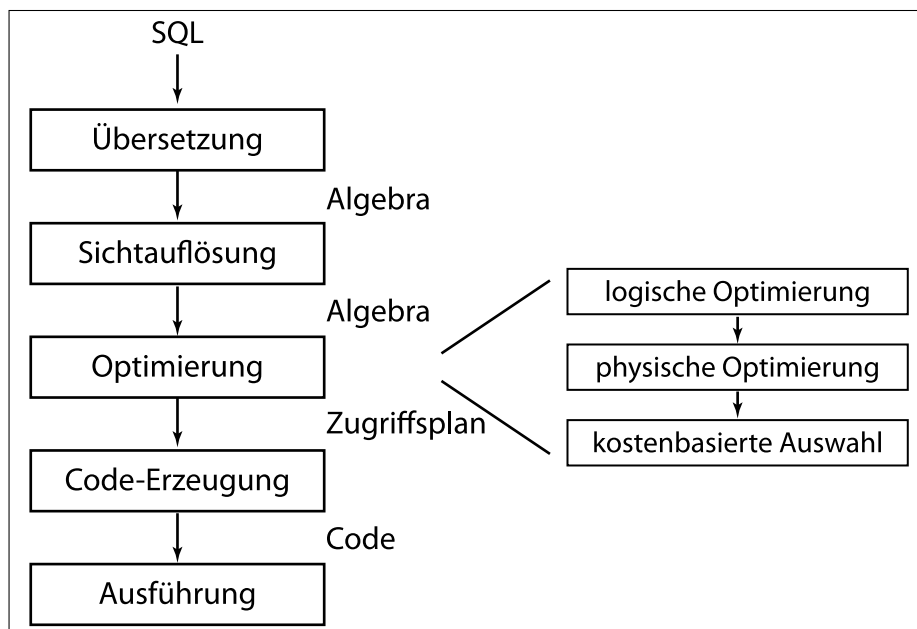


Abbildung 2.12: Phasen der Abfragebearbeitung ([SH99] S. 352)

### 2.3.1 Ablauf der Optimierung

Die oben gezeigte Abbildung lässt erkennen, dass die Optimierung in drei weitere Phasen aufgeteilt ist. Die Softwarekomponente, die für die Optimierung zuständig ist, wird als *Optimizer* bezeichnet und im weiteren Verlauf der Arbeit, insbesondere in den Kapitel 3 und 4, synonym für die Abfrageverarbeitung und -optimierung verwendet. Im Folgenden sollen nun die drei Phasen der Optimierung kurz vorgestellt werden.

<sup>4</sup>Sichten sind virtuelle Relationen, die durch gespeicherte *SQL*-Abfragen definiert sind und bei Verwendung neu berechnet werden.

## Logische Optimierung

Bei der *logischen Optimierung* werden physische Daten wie zum Beispiel die Größe von Relationen oder das Vorhandensein von Indexen nicht berücksichtigt. Die Optimierung erfolgt in dieser Phase durch algebraische Umformungs- und Ersetzungsregeln. Beispiele dieser Regeln finden sich zum Beispiel in [SKS97] S. 420 ff. Eine grundlegende Umformungsmöglichkeit ist das in Abbildung 2.13 zu sehende Verschieben von Selektionen.

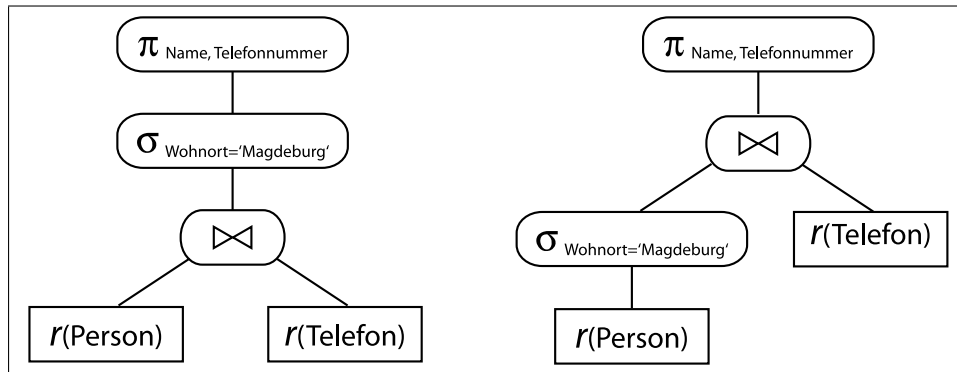


Abbildung 2.13: Umformung von Anfrageplänen

Beide Anfragepläne beschreiben die Anfrage:

*select Name, Telefonnummer from Person, Telefon where Person.ID = Telefon.ID*

Im linken Plan wird ein Verbund aller Datensätze über dem Attribut ID beider Relationen berechnet. Aus diesem Zwischenergebnis werden dann die Datensätze mit dem Wohnort „Magdeburg“ gefiltert und zurückgegeben. Im rechten Anfrageplan wird stattdessen die Relation Person erst nach Datensätzen, die dem Selektionskriterium „Wohnort = Magdeburg“ entsprechen, gefiltert und dann ein Verbund berechnet. Selbst wenn man davon ausgeht, dass beide Relationen vollständig eingelesen werden, ist der rechte Plan auf Grund der verringerten Anzahl von Vergleichen beim Join und dem kleineren Zwischenergebnis, das unter Umständen temporär auf die Platte geschrieben werden muss, günstiger.

## Physische Optimierung

Die *physische Optimierung* knüpft an die Ergebnisse der logischen Optimierung an. Die Anfragepläne werden in ausführbare Pläne transformiert. Die Operationen der Relationenalgebra werden durch konkrete interne Operationen für zum Beispiel Selektion, Projektion und Joins ersetzt. Die Verfahrensweise einiger dieser Operationen sind in [SH99] Kapitel 6 beschrieben. Für die Auswahl der entsprechenden Operationen werden im Gegensatz zur logischen Optimierung Informationen über vorhandene Indexe, etc. ausgenutzt. Darüber hinaus werden neue Verfahren wie zum Beispiel das *Pipelining* eingeführt. Das Ergebnis der physischen Optimierung sind mehrere (interne) Zugriffspläne.

## Kostenbasierte Auswahl

Aufgabe der *kostenbasierten Auswahl* ist die Bestimmung des Besten der zuvor erstellten Zugriffspläne. Erneut vorrangiges Ziel ist es, die Auswirkungen der Zugriffslücke durch

möglichst wenige Zugriffe auf den externen Speicher zu verringern. Die Auswahl erfolgt unter Berücksichtigung von Informationen wie zum Beispiel dem Vorhandensein von Indexen, der Größe von Seiten und Datensätzen, der Anzahl von Tupeln einer bestimmten Relation und der *Selektivität* von Attributen. Die Selektivität ist ein prozentuales Maß für die Größe einer Selektion im Vergleich zur entsprechenden Relation  $R$  und lässt sich beispielsweise für Gleichheit eines Attributes  $A$  mit dem Wert  $c$ , unter Annahme einer Gleichverteilung, wie folgt darstellen:

$$sel(A = c, R) = \frac{1}{W_{A,R}}$$

Die Variable  $W_{A,R}$  in der Gleichung entspricht der Anzahl verschiedener Werte des Attributes  $A$  in der Relation  $R$ .

Es ist anzumerken, dass es sich bei den ermittelten Kosten jeweils um Abschätzungen handelt. Die genauen Kosten eines Zugriffsplanes sind erst bei beziehungsweise nach dessen Ausführung ermittelbar. So gilt auch für die Optimierung allgemein, dass der ausgegebene Zugriffsplan nicht der *Beste*, sondern nicht der *Schlechteste* ist.

### 2.3.2 Berechnung von Verbunden

Die Abhängigkeit von Indexen lässt sich am Beispiel der Verbundberechnung gut erklären. Wie bereits in den vorangegangenen Kapiteln erwähnt, werden beim Verbund verschiedene Tabellen verknüpft. In der Regel werden dabei Tabellen über die Gleichheit von Attributen zusammengeführt, wie auch im Beispiel des Zugriffsplanes in Abbildung 2.13 über die Attribute  $ID$  beider Tabellen zu sehen ist. Von einem solchen *Equi-Verbund* zweier Tabellen  $R_1$  und  $R_2$  wird auch in den folgenden Betrachtungen dreier möglicher Verbundberechnungsverfahren ausgegangen.

Der *Nested-Loop-Join* erfolgt über zwei verschachtelte Schleifen. Der erste Datensatz von  $R_1$  wird mit allen Datensätzen  $R_2$ 's verglichen. Anschließend wird der zweite Datensatz von  $R_1$  erneut mit allen Datensätzen aus Relation  $R_2$  verglichen und so weiter. Jeder Datensatz von  $R_1$  wird also mit jedem Datensatz von  $R_2$  über das Verbundattribut verglichen.  $R_1$  wird bei diesem Verfahren einmal komplett von der Platte gelesen. Zum Vergleich mit dem aktuellen Datensatz aus  $R_1$  müssen von  $R_2$  immer alle Datensätze betrachtet werden. Reicht der Puffer eines Datenbanksystemes nicht aus, um  $R_2$  vollständig aufzunehmen, so muss  $R_2$  während der Verbundberechnung mehrfach vom externen Speicher gelesen werden, was wiederum zu hohen Kosten führt. Die Anzahl der Vergleiche entspricht dem Produkt der Anzahl von Datensätzen beider Relationen. Der *Nested-Loop-Join* ist als einziges hier genanntes Verfahren für alle Vergleichsarten ( $=, \neq, <, \dots$ ) geeignet.

Ein weiteres Berechnungsverfahren ist der *Hash-Join*, welcher in zwei Schritten erfolgt. Zuerst werden die Datensätze beider Relationen über eine *Hash-Funktion* in *Buckets* aufgeteilt. Im zweiten Schritt werden die Datensätze in den Buckets dann miteinander verglichen, um das Ergebnis zu berechnen. Neben dem Einlesen beider Relationen entstehen Plattenzugriffe durch die Verwendung der Buckets, welche abhängig von Größe und Anzahl extern zwischengespeichert werden. Die Anzahl der durchgeführten Vergleiche ist durch die Aufteilung in die Buckets wesentlich geringer als noch beim *Nested-Loop-Join*. Das in dieser Arbeit hauptsächlich betrachtete Datenbanksystem *DB2*

verwendet in der Regel Hash-Joins, wenn keine Indexe vorhanden sind (vgl. [SB03] S. 103).

*Merge-Joins* (deutsch: Misch-Verbund), auch *Merge-Sort-Joins* genannt, arbeiten mit nach dem Verbundattribut sortierten Relationen, oder sortieren die Relationen vor dem Vergleich, der in einer Mischphase stattfindet. Für das nachstehend beschriebene Mischen der bereits sortierten Relationen  $R_1$  und  $R_2$  seien  $k_1$  und  $k_2$  die Attribute, über die der Verbund wie folgt berechnet wird:

1. Falls  $k_1 < k_2$ , lies den nächstgrößeren Wert von  $k_1$ .
2. Falls  $k_1 > k_2$ , lies den nächstgrößeren Wert von  $k_2$ .
3. Falls  $k_1 = k_2$ , erfolgt der Verbund zwischen  $k_1$  und allen nachfolgenden  $k_2$  mit  $k_2 = k_1$ . Beim Auftreten des ersten  $k_2 \neq k_1$  wird der Nachfolger von  $k_1$  und das ursprüngliche  $k_2$  gewählt. Punkt 3 wird solange wiederholt, bis  $k_1$  einen neuen Wert annimmt. Nachdem in diesem Falle auch für  $k_2$  der nächstgrößere Wert gewählt wurde, wird wieder im Punkt 1 begonnen.

Das oben beschriebene Verfahren bricht ab, wenn für  $k_1$  oder  $k_2$  in Schritt 1 beziehungsweise Schritt 2 kein höherer Wert mehr vorhanden ist. Der Aufwand des Merge-Joins hängt zum einen davon ab, ob vor der Verbundberechnung eine Sortierung der Relationen erfolgen muss und von der Verteilung der Verbundattribute in den Relationen. Ist der Wert des Verbundattributes beider Relationen in allen Datensätzen gleich, werden alle Datensätze beider Relationen wie beim Nested-Loop-Join miteinander verglichen und verbunden. Sind  $k_1$  und  $k_2$  Schlüssel in ihren Relationen, entspricht die maximale Anzahl von Vergleichen der Summe aus Anzahl  $k_1$  und  $k_2$ .

Indexe über Verbundattribute können in allen drei genannten Verfahren die Anzahl der Plattenzugriffe verringern. Für die Vergleiche werden in diesem Fall die Schlüsselwerte aus dem Index gelesen und nur dann der Datensatz aus der Hauptdatei, wenn er zum Verbundergebnis gehört. Die Verringerung der Plattenzugriffe ergibt sich dadurch, dass auf einer Seite mehr Schlüsselwerte als Datensätze gespeichert werden können. Ein weiterer Vorteil ist die Sortierung von Indexen, welche Merge-Joins ohne Sortierphase ermöglichen. Der gemeinsame Nutzen zweier Indexe ist dann größer, als die Summe des Nutzen bei einzelnen Indexen. Dieses Verhalten ist in den Betrachtungen des 4. Kapitels nachzulesen.



## Kapitel 3

# Automatische Indexauswahl

Im vorigen Abschnitt wurde gezeigt, dass es für die Anfragebearbeitung in der Regel vorteilhaft ist, Indexe über Relationen zu bilden. So konnte beispielsweise durch Verwendung eines Indexes die Bearbeitungszeit der Anfrage zur Abbildung 2.6 (siehe Seite 11) in etwa halbiert werden. Die Auswahl passender Indexe ist für die Gewährleistung einer performanten Anfragebearbeitung also von besonderer Bedeutung.

Im Folgenden wird dieses *Index Selection Problem* (kurz ISP) mitsamt der dabei auftretenden Probleme allgemein erläutert. Nach dieser Erläuterung wird mit dem *DB2 Design Advisor* ([VZZ<sup>+</sup>00], [ZRL<sup>+</sup>04]) eine Softwarelösung vorgestellt, welche Datenbankadministratoren unterstützt, eine geeignete Indexauswahl zu treffen.

Abschließend erfolgen in einem letzten Unterabschnitt weitergehende Betrachtungen zur automatischen Indexauswahl und dem autonomen Index-Selbsttuning.

### 3.1 Index Selection Problem

Das ISP wird in [LSS07] anlehnend an [CFM95] wie folgt beschrieben: Gegeben sind  $m$  Anfragen  $Q_1, Q_2, \dots, Q_m$  und  $n$  Indexkandidaten  $I_1, I_2, \dots, I_n$ . Jeder Index  $I_i$  hat Verwaltungskosten (für die Aktualisierung bei Inserts, Updates, ...) von  $mcost(I_i)$  und benötigt den Speicherplatz  $size(I_i)$ . Für eine Anfrage  $Q_k$  ergibt sich der Gewinn *profit* eines Indexes  $I_i$  aus der Differenz der Ausführungskosten ohne  $I_i$  ( $cost(Q_k)$ ) und mit  $I_i$  ( $cost(Q_k, I_i)$ ). Demnach ist der Index  $I_i$  mit der maximalen Differenz für die Ausführung von  $Q_k$  optimal. Ferner gilt folgende Formel:

$$profit(Q_k, I_i) = \max\{0, cost(Q_k) - cost(Q_k, I_i)\}$$

Das ISP besteht nun aus der Auswahl einer Untermenge von  $C \subseteq I_1, \dots, I_n$  Indexen, die für die Anfragen  $Q_i$  verwendet werden können.  $C$  wird als *Indexkonfiguration* bezeichnet und muss

$$\sum_{i=1}^m \max\{profit(Q_i, I_j) : I_j \in C\} - \sum_{I_j \in C} mcost(I_j)$$

maximieren. Des Weiteren ist  $C$  nur dann eine gültige Lösung, wenn eine gegebene Maximalgröße  $S$  nicht überschritten wird:

$$\sum_{I_j \in C} size(I_j) \leq S.$$

Das ISP wird als *NP-Problem* ([Com78]) als Variante des *Rucksackproblems* beziehungsweise der ganzzahligen linearen Optimierung ([KPP04]) verstanden. In Rucksackproblemen wird für eine gegebene Menge von Objekten mit bestimmtem Nutzen und einem bestimmten Gewicht (Kosten) die Teilmenge gesucht, die eine definierte Gewichtschränke nicht überschreitet. Der summierte Nutzen der ausgewählten Objekte soll maximiert werden. Die betrachteten Objekte des Rucksacks sind im Falle des ISP die Indexe, welche mit den jeweiligen *profits* als Nutzen und ihrer Größe ( $size(I)$ ) so ausgewählt, dass sie die Maximalgröße  $S$  nicht überschreiten.

In der Praxis gestaltet sich die Auswahl der Indexe vor allem durch die große Anzahl möglicher Kandidaten als aufwendig. In einer Relation mit  $n$  Attributen sind

$$\sum_{k=1}^n \frac{n!}{(n-k)!}$$

verschiedene Indexe über  $k \leq n$  Attribute ohne Berücksichtigung der Sortierung einzelner Attribute möglich (vgl. [VZZ<sup>+</sup>00]). Im Fall der Relation *Person* ergeben sich durch die drei enthaltenen Attribute also 15 mögliche Indexe. Für Relationen mit vier Attributen sind es bereits 64 mögliche Indexe. Für Datenbankanwendungen wie zum Beispiel SAP R/3 [SAP08] mit mehreren hundert Tabellen wird die Anzahl möglicher Indexe schnell unüberschaubar groß.

Ein weiteres Problem ergibt sich durch die Queries und deren Verarbeitung selbst. Da die Arbeit der Anfrageoptimierung (siehe Abschnitt 2.3.1) für Administratoren nicht transparent ist, ist selbst für vielversprechende Indexkandidaten nicht sichergestellt, dass sie bei der Anfrageverarbeitung auch wirklich verwendet werden. Für komplexe oder generierte Queries lassen sich womöglich erst nach langer Analyse geeignete Indexe finden und auch die Verwendung von Sichten führt unter Umständen zu Fehleinschätzungen. Und nicht zuletzt macht es die Menge von Queries, die an ein Datenbanksystem gestellt werden, schwierig, eine gute Indexkonfiguration zu finden, die für die Gesamtheit aller Queries möglichst profitabel ist und die gegebenen Größenschranken nicht überschreitet.

Neben diesen Mengenproblemen herrschen zwischen Indexen Abhängigkeiten in Hinblick auf deren *profit*. Der *profit* eines Indexes hängt in der Regel von der Existenz anderer Indexe ab. Auf diese Problematik wird im Abschnitt 4 näher eingegangen.

Im Folgenden soll nun mit dem *DB2 Design Advisor* eine Software vorgestellt werden, die sich des Index-Selection-Problems annimmt und Anwender bei der Auswahl von Indexen unterstützt.

## 3.2 DB2 Design Advisor

Die folgenden Ausführungen zum *DB2 Design Advisor* (auch *Design Advisor* oder kurz *Advisor*) basieren hauptsächlich auf den Artikeln [VZZ<sup>+</sup>00] und [ZRL<sup>+</sup>04], die einen guten Einblick in die Arbeitsweise des Advisors geben. Neben Indexen werden in der aktuellen<sup>1</sup> Advisorversion auf Wunsch auch *Materialisierte Sichten* vorgeschlagen und Empfehlungen zur Erstellung *mehrdimensional-geclusterter Tabellen* ausgegeben. Die beiden letztgenannten Funktionen seien hier aber nur zur Vollständigkeit erwähnt und finden in dieser Arbeit keine weitere Berücksichtigung.

<sup>1</sup>Zum Zeitpunkt dieser Arbeit DB2 Universal Database 9.5

Die Arbeit des Design Advisors entspricht im Wesentlichen den zum ISP gemachten Ausführungen im vorhergehenden Abschnitt. Für eine gegebene Menge von Queries  $Q_k$ , im Folgenden *Workload* genannt, werden unter Zuhilfenahme von Statistiken (zum Beispiel Anzahl, Größe und Selektivität von Attributen) der betroffenen Relationen, Indexe ermittelt, die eine möglichst kostengünstige Abarbeitung des Workloads ermöglichen und eine vorgegebene Größenschranke nicht überschreiten. Nachfolgend sollen nun Architektur und Arbeitsweise des Design Advisors beschrieben werden.

### 3.2.1 Architektur

Der DB2 Design Advisor folgt dem Vorschlag von [FST88], den Optimizer eines DBMS zur Bewertung von Indexkandidaten zu nutzen. Anhand der sich ergebenden Kosten für die Ausführung einer Anfrage mit und ohne Indexkandidaten lässt sich feststellen, ob bestimmte Indexe bei der Anfragebearbeitung genutzt werden und welchen *profit* damit verbunden ist.

Abbildung 3.1 zeigt die Architektur des DB2 Design Advisors in Anlehnung an [VZZ<sup>+</sup>00]. Der Advisor besteht dabei aus vier Komponenten:

- Der **Graphischen Benutzeroberfläche**,
- dem **db2advis Kommandozeilenprogramm**, welches den Prozess der Indexfindung durchführt,
- dem **Optimizer**, dem zusätzliche Erweiterungen zur Suche und Auswertung geeigneter Indexe hinzugefügt wurden und
- den **Advise Tables** (deutsch etwa: Vorschlagstabellen; siehe Anhang), die als Kommunikationskonstrukt zwischen db2advis und dem Optimizer dienen und für den Indexauswahlprozess wichtige Objekte speichern.

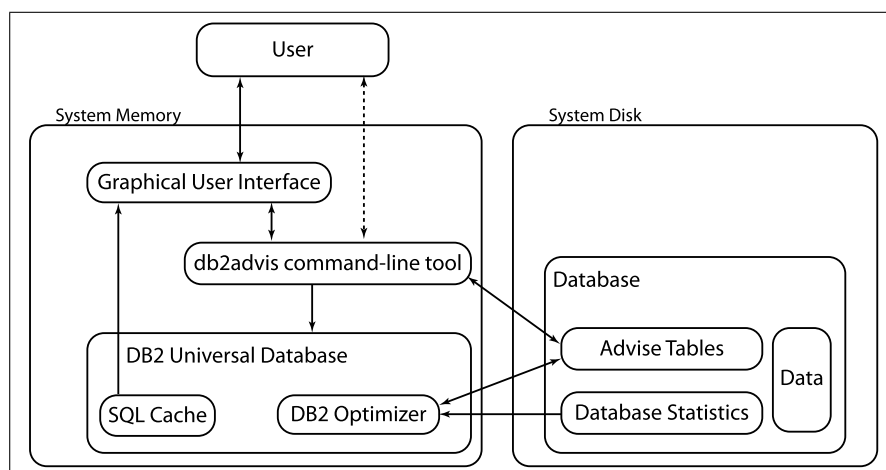


Abbildung 3.1: Architektur des DB2 Design Advisors

Im Normalfall wird der Design Advisor durch die graphische Benutzeroberfläche aufgerufen. Zunächst erfolgt dann die Auswahl des zu bearbeitenden Workloads. Dazu können Queries zum Beispiel aus dem *SQL Cache*, der eine Art Anfragehistorie des

DBS speichert und in fast allen Datenbanksystemen anzutreffen ist, importiert werden. Der SQL Cache ermöglicht die Erstellung von Workloads, die das Anfrageverhalten im „Alltag“ eines DBS gut wiedergeben und deren Optimierung damit von besonderem Interesse ist. Der Workload wird in der *ADVISE\_WORKLOAD*-Tabelle für die Abarbeitung gespeichert. Des Weiteren kann man die maximale Größe der Indexempfehlungen, sowie die maximale Laufzeit der Suche nach geeigneten Indexen angeben.

Nach dem Start der eigentlichen Indexauswahl durch den Aufruf von *db2advis* werden alle Queries des Workloads nacheinander vom Optimizer in den Modi *RECOMMEND\_INDEXES* oder *EVALUATE\_INDEXES* ausgeführt. Dabei handelt es sich jeweils um sogenannte *EXPLAIN* (deutsch: erklären, aufzeigen) Modi, bei denen nur Kostenabschätzungen (gemessen in *Timerons*) für Anfragen erstellt werden und keine Ausführung stattfindet. Das Ergebnis ist also nur der Zugriffsplan, der die geschätzten Anfragekosten beinhaltet. Nach der Abarbeitung des Workloads werden die Ergebnisse, also die empfohlenen Indexe, ausgegeben.

```

$ ./db2advis.exe -d diplom -s "select * from partsupp
where ps_availqty > 3120 order by ps_suppkey"
...
Optimierung abgeschlossen.
  1 Indizes in aktueller Lösung
[221877,0000] Timeron (ohne Empfehlungen)
[58255,0000] Timeron (mit aktueller Lösung)
[73,74%] Verbesserung
...
LISTE DER EMPFOHLENEN INDIZES
=====
index[1], 13,165MB
CREATE UNIQUE INDEX "KKUEHNE"."IDX808261311460000"
ON "KKUEHNE"."PARTSUPP" ("PS_SUPPKEY" ASC, "PS_PARTKEY"
DESC) ALLOW REVERSE SCANS ;
...
<index>
<identifier>
<name>IDX808261311460000</name>
<schema>KKUEHNE </schema>
</identifier>
<table><identifier>
<name>PARTSUPP</name>
<schema>KKUEHNE </schema>
</identifier></table>
<statementlist>1</statementlist>
<benefit>163622,000000</benefit>
<overhead>0,000000</overhead>
<diskspace>13,165062</diskspace>
</index>
...

```

Abbildung 3.2: Ergebnisauszug db2advis

Abbildung 3.2 zeigt eine gekürzte Ergebnisausgabe von *db2advis*. Der Workload besteht an dieser Stelle nur aus einer Anfrage, die manuell definiert wurde. Es ist zu sehen, wie groß der *profit* der vorgeschlagenen Indexkonfiguration *C* vom Optimizer

eingeschätzt wird und welche Größe sie beansprucht. Diese Informationen werden auch für die einzelnen Indexe in Form des *Benefits* (deutsch: Nutzen, entspricht dem *profit*) und der Größe (Discspace) angezeigt.

### 3.2.2 Optimierung einzelner Queries

Nachdem bereits in kurzes Beispiel zur Arbeit des DB2 Design Advisors gegeben wurde, soll nun geklärt werden, wie der ebenfalls bereits kurz vorgestellte RECOMMEND\_INDEXES-Modus, sprich die Suche nach geeigneten Indexen, abläuft.

Die dem RECOMMEND\_INDEX-Modus zugrundeliegende Idee ist die Einführung *virtueller Indexe*. Diese Indexe sind nicht materialisiert, also nicht in der Datenbank vorhanden, können aber bei der Suche nach dem besten Zugriffsplan wie „normale“ Indexe verwendet werden. Enthält der vom Optimizer als bester erachtete Zugriffsplan virtuelle Indexe, so werden diese Indexe für die Erstellung vorgeschlagen (englisch: recommend).

#### SAEFIS

In den Betrachtungen zum Index-Selection-Problem zeigte sich die bereits hohe Anzahl möglicher Indexe nur einzelner Relationen. Die Einführung aller möglichen Attributkombinationen als virtuelle Indexe würde den Optimizer mit einem zu großen Suchraum konfrontieren und dessen Arbeit ineffizient machen. Zur Auswahl geeigneter virtueller Indexe verwendet der RECOMMEND\_INDEXES-Modus daher den sogenannten *Smart column Enumeration for Index Scans* Algorithmus (SAEFIS), der gegebene Anfragen nach vielversprechenden Indexkandidatsspalten durchsucht. Es wird dabei in fünf Kategorien unterschieden:

- **EQ**: Spalten/Attribute, die in Gleichheitsprädikaten erscheinen (Wohnort = 'Magdeburg').
- **O**: Spalten, nach denen sortiert, oder gruppiert wird (order by Wohnort).
- **RANGE**: Spalten die in Bereichsprädikaten angefragt werden ( $ID > 10$  and  $ID < 200$ ).
- **SARG**: Spalten, die in jedem Prädikat außer in geschachtelten Unterabfragen vorkommen und keine große Objekte (LOB) beinhalten
- **REF**: Spalten, die keiner der anderen Kategorien entsprechen, und in der Anfrage vorkommen.

Der SAEFIS-Algorithmus betrachtet also nur die Attribute, die auch in den betrachteten Anfragen vorkommen. Um auch Mehr-Spalten-Indexe empfehlen zu können, werden die betrachteten Attribute in verschiedenen Kombinationen wie zum Beispiel:

- EQ + O
- EQ + O + RANGE
- EQ + O + RANGE + SARG

- ...

ebenfalls berücksichtigt. Doppelte Indexe und Indexspalten werden dabei eliminiert. Um möglichst keine gewinnbringenden Attributkombinationen zu verpassen, werden einige weitere, zufällige, Kombinationen der gewählten Attribute erzeugt. In [VZZ<sup>+</sup>00] wird dieses Verfahren als *Brute Force and Ignorance*, kurz *BFI*, bezeichnet.

### Index Statistiken

Bevor der Optimizer die Kosten für Indexoperationen (lookup, insert, update, ...) berechnen kann, müssen für die virtuellen Indexkandidaten noch entsprechende Statistiken erstellt werden. Für bereits materialisierte Indexe entfällt dieser Schritt.

Da Indexe in DB2 durch  $B^+$ -Bäume realisiert werden, sind Statistiken über den Aufbau des Indexbaumes von Interesse. Dazu zählen unter anderem Höhe und Ordnung des Baumes, die sich aus der Anzahl der Datensätze und der Größe des Indexsuchschlüssels und Seitengröße errechnen lassen. Für Werte wie Clustering und Attributeindeutigkeit werden dagegen pessimistische Werte vergeben, um mögliche Fehlberechnungen, bei denen Indexoperationen zu geringe Kosten zugerechnet werden, zu vermeiden.

Für weitere Informationen über die Berechnung von Indexstatistiken sei an dieser Stelle auf die in Abschnitt 3.2 eingangs erwähnte Literatur verwiesen.

Abschließend soll der Algorithmus für die Indexempfehlung einer Anfrage  $Q$  noch einmal im Pseudocode beschrieben werden. Es ist anzumerken, dass die Indexempfehlungen mit nur einem einzigen Optimizeraufruf generiert werden können.

1. Aktiviere den RECOMMEND\_INDEXES-Modus.
2. Rufe den Optimizer auf.
3. Wähle virtuelle Indexe mit SAEFIS und generiere deren Statistiken.
4. Wähle virtuelle Indexe mit BFI und generiere deren Statistiken.
5. Stelle dem Optimizer die virtuellen Indexe zur Verfügung.
6. Erzeuge den besten Zugriffsplan für  $Q$  mit dem Optimizer.
7. Durchsuche den Zugriffsplan nach virtuellen Indexen.
8. Gib die gefundenen virtuellen Indexe als Empfehlungen aus.

### 3.2.3 Optimierung von Workloads

Nachdem die grundlegende Funktionsweise der Indexauswahl im DB2 Design Advisor vorgestellt wurde und Indexkonfigurationen lokal (für einzelne Queries) optimiert wurden, soll nun betrachtet werden, wie für mehrere Queries umfassende Workloads Indexempfehlungen unter Angabe einer Größenschränke erstellt werden. Db2advis wurde dafür um Algorithmen erweitert, die aus den Ergebnissen der Einzel-Query-Optimierung eine für den gesamten Workload profitable Auswahl erstellen. Der Algorithmus der Workloadoptimierung arbeitet für einen Workload  $W$  wie folgt:

1. Beziehe den Workload  $W$ .
2. Setze die Menge aller empfohlener Indexe  $R = \emptyset$ .
3. Für jedes Query  $Q_k$  in  $W$ ,
  - (a) Berechne die Kosten  $cost_{vorhanden}(Q_k)$  mit den existierenden Indexten.
4. Für jedes Query  $Q_k$  in  $W$ ,
  - (a) Berechne die Kosten  $cost_{virtuell}(Q_k)$  (mit virtuellen Indexten) im RECOMMEND\_INDEX Modus.
  - (b) Füge die empfohlenen virtuellen Indexte der Menge  $R$  hinzu.
5. Für jeden Index  $I_j$  in  $R$ ,
  - (a) Berechne  $profit(I_j) = cost_{vorhanden}(Q_k) - cost_{virtuell}(Q_k)$  und
  - (b) Berechne die Größe  $size(I_j)$
6. Sortiere die Indexte in  $R$  nach dem Verhältnis  $\frac{profit(I_j)}{size(I_j)}$
7. Suche Präfixindexte (siehe Abschnitt 4.1.2)
8. Füge Indexte aus  $R$  so lange zu Lösung hinzu, bis die Größenschranke  $S$  erreicht ist.
9. So lange die Zeit  $T$  nicht überschritten ist, wiederhole
  - (a) TRY\_VARIATION

Zunächst werden also die Kosten für alle Queries  $Q_k$  ohne virtuelle Indexte als Ausgangswert der Betrachtungen berechnet (3a). Anschließend werden im RECOMMEND\_INDEXES Modus Indexkandidaten vorgeschlagen und deren jeweiliger *profit* samt ihrer Größe (*size*) berechnet (4 und 5). Sind Indexte an der Ausführung mehrerer Queries beteiligt, so werden die jeweiligen *profits* aufaddiert. Ferner gilt, dass alle Indexte die für ein bestimmtes Query empfohlen werden, den gleichen *profit* für dieses Query erhalten. Das Auswahlverfahren der zur Gesamtlösung vorgeschlagenen virtuellen Indexte ist als Rucksackproblem modelliert und wird mit einer Greedy-Variante gelöst (6 und 8).

[VZZ<sup>+</sup>00] merkt an, dass es problematisch ist, allen Indexten eines Queries den gleichen Profit zuzuordnen. In Kapitel 4 wird dieses Problem genauer beschrieben. Db2advis wurde deshalb mit *TRY\_VARIATION* (9) ein Verfahren hinzugefügt, welches der Verbesserung der durch das Greedy-Verfahren entstandenen Lösung dient. *TRY\_VARIATION* tauscht eine kleine Menge von Indexten der Lösungsmenge mit kleinen Mengen von Indexten, die nicht in der Lösungsmenge enthalten sind, aus und berechnet die Kosten des Workloads neu. Dazu wird der EXPLAIN Modus *EVALUTE\_INDEXES* verwendet. Im Gegensatz zum RECOMMEND\_INDEXES Modus werden dabei keine Indexte empfohlen, sondern nur die Kosten von Queries unter Berücksichtigung ausgewählter virtueller Indexte berechnet. Ist eine solche, zufällig erzeugte Lösung in Bezug auf die Workloadbearbeitung kostengünstiger als die aktuelle Lösung, so wird diese mit der von

TRY\_VARIATION gefundenen ersetzt. Die Laufzeit von TRY\_VARIATION wird als Parameter an db2adviz übergeben.

Weiterhin argumentiert [VZZ<sup>+</sup>00], dass die durch Indexe entstehenden Kosten, also negative Profite durch Updates, Inserts und Deletes nicht berücksichtigt werden, da man in der RECOMMEND\_INDEXES Phase nicht weiß, welche Indexe später tatsächlich materialisiert werden.

Als Abschluss der Betrachtungen zum DB2 Design Advisor wurden Indexempfehlungen zum TPC-H Workload (1 Gigabyte) auf DB2 UDB 9.5 Express erzeugt. Abbildung 3.3 zeigt die Ergebnisse, die dabei auf dem Testsystem (siehe Anhang) für eine Lösung mit 300 Megabyte, bestehend aus 29 Indexen, und eine Lösung mit 150 Megabyte, bestehend aus 15 Indexen, erzielt wurden. Die vorgegebene maximale Laufzeit von zehn Minuten wurde in keinem der Fälle erreicht. Die Bearbeitungszeit lag in beiden Fällen unter einer Minute.

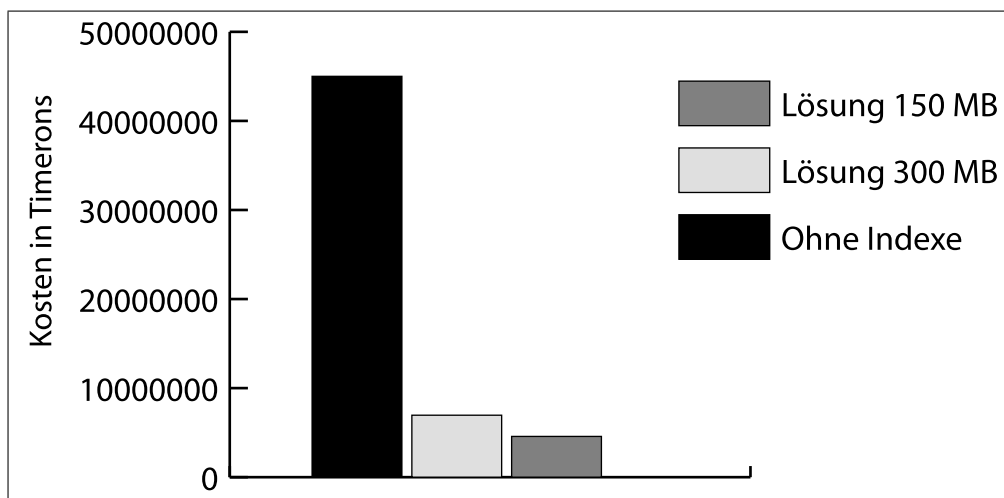


Abbildung 3.3: Kosten von Indexkonfigurationen für den TPC-H Workload

### 3.3 Weitere Betrachtungen

Durch seine Architektur und die Art seines Aufrufes wird der DB2 Design Advisor auch als *Index-Wizard Werkzeug* bezeichnet. Der Zeitpunkt eines Optimierungslaufes muss dabei vom Anwender selbst bestimmt werden. Auch die Lösungen anderer Hersteller [CN98], [Ora03] entsprechen diesem Prinzip. Augenscheinlich ergibt sich dadurch die Frage, wann ein Optimierungslauf durchzuführen ist. Ein zu seltenes Anpassen der Indexkonfiguration kann dazu führen, dass eine früher gewählte Indexkonfiguration für die aktuelle Situation in Bezug auf Anfragen und Daten zunehmend ungeeignet wird. Ebenso suboptimal ist ein zu häufiges Starten der Wizards, welches zu unnötiger Arbeitslast für das Datenbanksystem führt.

#### Design Alerter des MS SQL-Servers

[BC06] stellt mir dem *Design Alerter* ein Werkzeug vor, welches den Anwender bei der Entscheidung zur Durchführung einer Indexoptimierung durch einen Wizard unterstützt.

Der Design-Alterter arbeitet nach dem Prinzip des Regelkreislafs des Indexselbsttunings [WHMZ94]. Der Regelkreislauf besteht aus den drei Phasen:

- **Überwachung:** Für alle Anfragen werden Kosten und Nutzen möglicher Indexe überprüft.
- **Diagnose/Vorhersage:** Sind bessere Indexkonfigurationen als die aktuell materialisierte möglich?
- **Reaktion:** Erzeugen der neuen Konfiguration zu geeigneten Zeitpunkten.

Da es beim genannten Regelkreislauf zu einer kontinuierlichen Überwachung des Datenbanksystems kommt, ist es wichtig, dass bei Verfahren, die diesen Regelkreislauf implementieren, ein möglichst geringer Mehraufwand (Overhead) für das Datenbanksystem entsteht. Beim Design Alerter wird dies durch sogenannte *Tags* in den Zugriffsplänen realisiert. Diese Tags (mit Notizen zu übersetzen) enthalten *Index-Request*, die ähnlich dem RECOMMEND.INDEXES Modus des DB2 Advisors, bei der Optimierung von Anfragen gewonnen werden. Der Profit eines solchen Index-Request wird ermittelt und in den Tags gespeichert. Überschreiten die Profite dieser Tags einen bestimmten Grenzwert, so wird ein „Alarm“ ausgelöst, der den Anwender über die Notwendigkeit einer Indexoptimierung informiert.

Abbildung 3.4 zeigt den, dem Design Alerter zugrunde liegenden Zyklus, der auf dem oben genannten Regelkreislauf basiert.

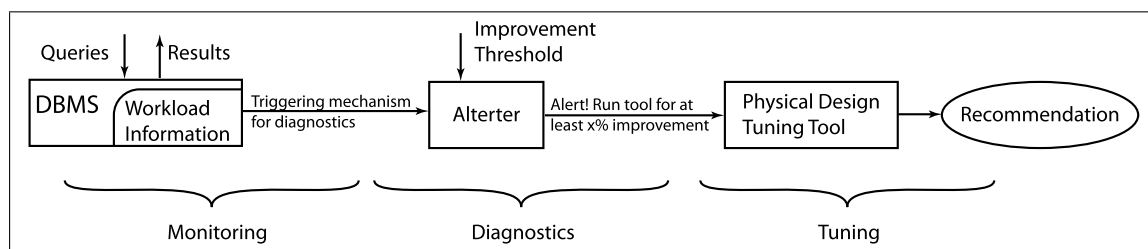


Abbildung 3.4: Monitor-Diagnose-Zyklus des Design Alerter ([BC06])

In [BC07] wird ein vollständig autonomes Indexoptimierungsverfahren vorgestellt, welches den Design Alerter so erweitert, dass er beim Überschreiten des Profitgrenzwertes selbstständig eine neue Indexkonfiguration erzeugt. Dieser Ansatz, der den Anwender komplett vom ISP entbindet und zu jeder Zeit die Indexkonfiguration überwacht, befindet sich allerdings noch im experimentellen Status.

## QUIET

Eine weitere vollkommen selbstständige, stets aktive Lösung (auch als *online*-Verfahren bezeichnet) des ISP stellt das *QUIET* System [SGS03] dar. QUIET setzt auf der DB2 Universal Database auf und arbeitet als sogenannte *Middleware* zwischen Datenbanksystem und Anfragesteller. Dies bedeutet, dass Datenbankanfragen an das QUIET System übergeben werden, welches diese dann an die Datenbank weiterleitet. Abbildung 3.5 zeigt die Architektur in [SGS03] verwendeten QUIET-Demos. Im praktischen Einsatz würde der Query Generator durch eine anfragestellende Institution ersetzt werden.

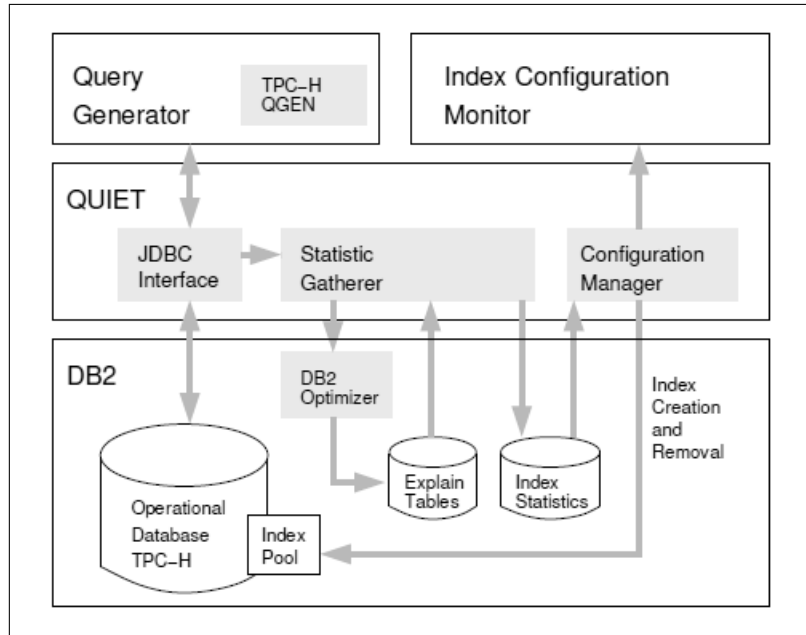


Abbildung 3.5: Architektur des QUIET-Demos ([SGS03])

Jedes an QUIET gestellte Query wird zunächst im RECOMMEND\_INDEXES und im EXPLAIN Modus an die Datenbank weitergeleitet. Die verwendeten Indexe (virtuelle und materialisierte) werden in einem *Index-Katalog*  $\mathcal{D} = \{I_1, \dots, I_k\}$  gespeichert. Für jeden Index  $I_k$  werden die Informationen  $profit(I_k)$ ,  $type(I_k)$  (virtuell oder materialisiert) und  $size(I_k)$  gepflegt. Ziel ist es erneut, eine Indexkonfiguration  $C$  bestimmter Größe mit maximalem Profit zu finden. Die Auswahl dieser Konfiguration geschieht auch in QUIET nach einem Greedy-Verfahren. Der Profit einzelner Indexe wird auf unterschiedliche Weise berechnet. Ausgangspunkt sind zunächst optimale (lokale) Indexkonfigurationen  $\mathcal{I}_q$ , die alle Indexe  $I$  beinhalten, die für ein Query  $Q$  durch den RECOMMEND\_INDEXES-Modus geliefert werden. Während im DB2 Design Advisor noch allen Indexen einer lokalen Indexkonfiguration der Profit der gesamten Konfiguration zurechnet wurde, werden in [SSG04] die folgenden Möglichkeiten vorgestellt:

- Rechne den Indexen einen *konstanten* Profit zu.
- Rechne allen Indexen  $I \in \mathcal{I}_q$  den *vollen* Profit  $profit(Q, \mathcal{I}_q)$  zu (wie DB2 Advisor).
- Rechne den Indexen  $I \in \mathcal{I}$  den *durchschnittlichen* Profit  $\frac{profit(Q, \mathcal{I}_q)}{|\mathcal{I}_q|}$ .
- Rechne einzelnen Indexen  $i$  den *gewichteten* Profit  $profit(Q, \mathcal{I}_q) \cdot \frac{profit(Q, \{i\})}{\sum_{j \in \mathcal{I}_q} profit(Q, \{i\})}$  zu.

Da es sich bei den o.g. Verfahren nur um Näherungslösungen handelt, ist davon auszugehen, dass in jedem Fall gilt:

$$profit(Q, \mathcal{I}_q) \neq \sum_{j \in \mathcal{I}_q} profit(Q, \{j\})$$

Dieses Verhalten soll in Kapitel 4 noch näher betrachtet werden.

Als Online-Verfahren muss QUIET auch auf Änderungen im Workload reagieren und in diesem Zusammenhang die Indexkonfiguration ändern. Da die Kosten zu häufiger Änderungen der materialisierten *globalen* Indexkonfiguration  $mat(\mathcal{D})$  den dadurch gewonnen Profit schmälern, wenn nicht sogar überwiegen, kommt es nur zu einem Konfigurationswechsel, wenn eine bestimmte Profitgrenze überschritten wird. Änderungen der materialisierten Indexkonfiguration  $\mathcal{D}$  werden durch den Austausch von Indexmengen  $\mathcal{I}$  realisiert, wenn folgende Bedingung erfüllt ist:

$$\begin{aligned} & profit(mat(\mathcal{D}) \cup \mathcal{I} \setminus \mathcal{I}_{rep}) - \\ & profit(mat(\mathcal{D})) > MIN\_DIFF \wedge \\ & size((mat(\mathcal{D}) \cup \mathcal{I} \setminus \mathcal{I}_{rep}) < MAX\_SIZE \end{aligned}$$

$\mathcal{I}_{rep} \subseteq mat(\mathcal{D})$  entspricht der ersetzten Konfiguration. Die neue Indexkonfiguration darf die Maximalgröße  $MAX\_SIZE$  nicht überschreiten.

Um eine schnellere Anpassung an den aktuellen Workload zu ermöglichen, verwendet QUIET bei der Berechnung der Indexprofite eine *Alterungs-Strategie* (aging strategy) basierend auf Ideen von [SSV96] und [OOW93]. Die Idee des *Aging* ist, alte Indexstatistiken geringer zu gewichten als weniger alte. Der Aging-Prozess wird ausgelöst, wenn ein Index  $I_{max}$  eine bestimmte Profitgrenze überschreitet:

$$profit(I_{max}) \geq MAX\_PROFIT.$$

Kommt es zu einem solchen Fall, werden die Profite aller Indexe  $I_k \in \mathcal{I}$  angepasst:

$$profit(I_k) := s \cdot profit(I_k), 0 < s < 1$$

Kommt es durch ein Query zur Änderung von  $mat(\mathcal{D})$ , so kann die neue Konfiguration vor der eigentlichen Anfrageverarbeitung materialisiert werden.



# Kapitel 4

## Analyse und Konzept

In Kapitel 3 wurde das Indexauswahlverfahren des DB2 Design Advisors vorgestellt. Wie zu sehen war, erfolgt die Selektion der Indexe nach dem Prinzip des Rucksackverfahrens auf Basis einzelner Indexe. Diese Vorgehensweise soll im ersten Teil dieses Kapitels analysiert werden. In diesem Zusammenhang wird die Abhängigkeit zwischen Indexen in Hinblick auf deren Nutzen betrachtet und hinterfragt, ob mit dem Rucksackverfahren eine optimale Lösung erzielt werden kann.

Aufbauend auf den Ergebnissen des ersten Teils, wird im zweiten Teil des Kapitels ein alternatives Auswahlverfahren vorgestellt, welches auf lokal-optimalen Indexkonfigurationen basiert. Ziel ist dabei die Verbesserung des Bewertungs- und Auswahlverfahrens der zu materialisierenden Indexkonfigurationen. Ferner wird erörtert, inwieweit das Problem der Indexabhängigkeiten durch die Verwendung lokaler Indexkonfigurationen als Basis des Auswahlverfahrens gelöst werden kann.

### 4.1 Analyse

Der Abschnitt der Analyse beschreibt zunächst das Rucksackproblem an einem Beispiel, welches mögliche Abhängigkeiten zwischen den Objekten des Rucksacks beschreibt. Im folgenden Unterabschnitt werden mögliche Abhängigkeiten zwischen Indexen betrachtet und an einem konkreten Beispiel dargestellt. Abschließend wird das Auswahlverfahren des DB2 Design Advisors allgemein und im Hinblick auf Indexabhängigkeiten untersucht.

#### 4.1.1 Rucksackprobleme - Ein Beispiel

Das Rucksackproblem soll hier kurz an einem Beispiel veranschaulicht werden. Für einen Wanderausflug soll ein Rucksack mit der Kapazität von 7 Gewichtseinheiten mit den Gegenständen aus Tabelle 4.1 gepackt werden. Die Gegenstände haben einen bestimmten Nutzen und ein Gewicht, welche bezogen auf das ISP den Profit  $profit(I)$  und die Größe  $size(I)$  eines Indexes  $I$  darstellen. Weiterhin sollen die Gegenstände unteilbar (keine 0,3 Messer in der Lösung) und nur einmal auswählbar sein, was ebenfalls dem Verhalten von Indexen entspricht.

Ein möglicher Lösungsweg über sogenannte *Greedy-Verfahren* wurde bereits in Kapitel 3 erwähnt. Greedy-Verfahren (beziehungsweise Greedy-Algorithmen) arbeiten schritt-

Gegenstand	Nutzen	Gewicht	$\frac{\text{Nutzen}}{\text{Gewicht}}$
Messer	1	1	1
Brotaufstrich	6	2	3
Brot	6	4	1,5
Decke	2	1	2

Tabelle 4.1: Rucksackproblem

weise und wählen immer die zulässige Lösung, im Fall des Rucksackes den Gegenstand, der aktuell den höchsten Gewinnzuwachs verspricht. Im gezeigten Beispiel wird der Gewinn als Quotient aus Nutzen und Gewicht berechnet. Als Lösung liefert ein Greedy-Verfahren die Lösungsmenge  $L = \{\text{Brotaufstrich}, \text{Brot}, \text{Decke}\}$  mit dem Nutzen  $U(L) = 14$ . Obwohl Greedy-Verfahren keine optimalen Lösungen garantieren (ein Messer mit dem Nutzen 2 würde das Brot in der Greedy-Lösung ersetzen, so dass  $U(L) = 10$ ), wird es im DB2 Design Advisor zur Indexauswahl verwendet. Die Komplexität von Greedy-Verfahren ist  $O(n)$  für  $n$

Weitere Lösungsverfahren wie zum Beispiel das *Backtracking*, bei dem alle möglichen Lösungen, also alle Gegenstandskombinationen, betrachtet werden oder Verfahren der *Dynamischen Programmierung* [SD06] garantieren zwar optimale Lösungen, arbeiten aber mit höheren Komplexitäten von  $O(2^n)$  für das Backtracking, beziehungsweise  $O(n \cdot S)$ , mit  $S$  Gewichtseinheiten als Gewichtsschranke, für die Dynamische Programmierung, die außerdem nur für Objekte von ganzzahligen Nutzen und Gewicht anwendbar ist. Im Hinblick auf ein stets aktives Indextuning (online) ist dieser Mehraufwand nicht vertretbar.

### Abhängigkeiten bei der Nutzenbewertung

In Tabelle 4.1 wurde jedem Gegenstand (im Folgenden wird der Begriff Objekt verwendet) ein eindeutiger Nutzen zugewiesen und eine Lösung mit maximalem Nutzen gefunden. Beim Versuch den Brotaufstrich mit der Decke auf das Brot zu streichen, wird diese Lösung jedoch hinterfragt und eine neue Nutzenbetrachtung erscheint sinnvoll. Anstatt den Objekten einen festen Nutzen zuzuweisen, ergibt sich der Nutzen eines Objektes aus einer Funktion, die von allen anderen Objekten abhängig ist. Eine solche Nutzenfunktion könnte für das Messer wie folgt definiert sein:

$$\text{Nutzen Messer} = \begin{cases} 5, & \text{wenn Brot und Brotaufstrich eingepackt werden} \\ 2, & \text{wenn Brot eingepackt} \\ 1, & \text{in allen anderen Fällen} \end{cases}$$

Ein Rucksackproblem, welches mit solchen Nutzenfunktionen modelliert ist, lässt sich nicht durch ein Greedy-Verfahren, wie es zuvor beschrieben wurde, lösen. Die Objektauswahl führt also zu einem hohen Rechenaufwand, der im Hinblick auf ein online-Verfahren zur Indexauswahl nicht vertretbar ist.

## 4.1.2 Indexabhängigkeiten

Motiviert durch das vorangegangene Beispiel, welches Abhängigkeiten bei der Nutzenbetrachtung im Rucksackproblem einführte, sollen nun Indexabhängigkeiten in Hinblick auf die automatische Indexauswahl kurz untersucht werden. Die Untersuchungen wurden erneut auf Basis der DB2 Universal Database Version 9.5 Express durchgeführt.

### Joins

Beim Verbund zweier Tabellen können Indexe über den jeweiligen Verbundattributen während einer Operation gleichzeitig verwendet werden. Die Abhängigkeitsvermutung liegt also nahe und wird durch die Praxis bestätigt. Bezogen auf den Profit zweier Indexe  $I_1$  und  $I_2$  ergeben sich zwei Möglichkeiten, die für das ISP von besonderem Interesse sind:

1.  $I_1$  **reduziert die Notwendigkeit beziehungsweise den Nutzen von  $I_2$** : Dieses Verhalten wird in [VZZ<sup>+</sup>00] am Beispiel eines Nested-Loop-Joins (siehe Abschnitt 2.3.2) beschrieben. Die Relation ohne Index wird wie beim Nested-Loop-Join üblich in einer Schleife komplett durchlaufen. Die Verbundpartner der anderen Relation werden dann über den Index angefragt. Die zusätzliche Verwendung von  $I_2$  würde die Kosten des Verbundes nur geringfügig verbessern, so dass in Bezug auf die Kosten von  $I_2$  eine Materialisierung (Indexauswahl) nicht sinnvoll ist.
2.  $I_1$  **erreicht einen höheren Nutzen<sup>1</sup>, wenn  $I_2$  vorhanden ist**: Dieses Verhalten ist zum Beispiel bei Bitmap-Indexten (siehe Abschnitt 2.2.4) zu beobachten, wenn für den Verbund die Bitvektoren zweier Indexe verglichen werden können. Der Verbund kann in diesem Fall sehr effizient berechnet werden. Über die TID's werden dann die Datensätze des Verbundergebnisses gelesen.

Bevor zu den oben genannten Möglichkeiten zwei Beispiele gegeben werden, sollen die Sachverhalte 1 und 2 noch einmal als Formel formuliert werden:

$$(1) \quad profit(Q, \{I_1, I_2\}) < profit(I_1) + profit(I_2)$$

$$(2) \quad profit(Q, \{I_1, I_2\}) > profit(I_1) + profit(I_2)$$

Als Beispiel für den ersten der genannten Fälle soll die Anfrage:

```
select * from partsupp, part where part.p_partkey = partsupp.ps_partkey
```

auf den Tabellen des TPC-H-Benchmarks betrachtet werden. Die Gesamtkosten der Anfrage belaufen sich ohne die Verwendung von Indexten auf etwa 211.219 Timerons<sup>2</sup>. Der Verbund wurde durch einen Hash-Join berechnet. Die Verwendung des Indexes  $I_{PART}$

<sup>1</sup>Gemessen am halbierten Gesamtprofit, der durch  $I_1$  und  $I_2$  entsteht.

<sup>2</sup>In [IBM08] heißt es: „Die Einheit zur Berechnung des Aufwands ist der sog. Timeron. Ein Timeron entspricht nicht direkt der tatsächlich vergangenen Zeit, sondern stellt eine ungefähre Schätzung der Ressourcen (Aufwände) dar, die vom Datenbankmanager für die Ausführung von zwei Plänen für dieselbe Abfrage benötigt werden.“

über dem Attribut P\_PARTKEY der Relation PART konnte den Gesamtaufwand auf etwa 110.000 Timerons verringern. Verwendet wurde dabei ein Nested-Loop-Join zur Verbundberechnung.  $I_{PART}$  hat also einen Profit von etwa 100.000 Timerons.

Eine weitere Umformung des Anfrageplanes wurde durch den Index  $I_{PARTSUPP}$  über dem Attribut PS\_PARTKEY der Tabelle PARTSUPP erreicht. Der berechnete Aufwand (ohne  $I_{Part}$ ) betrug 42.298 Timerons. Der Profit von  $I_{PARTSUPP}$  entspricht demnach etwa 170.000 Timerons. Einhergehend mit der Anfragebearbeitung unter Verwendung von  $I_{PARTSUPP}$  wurde erneut die Verbundberechnung verändert. Verwendet wurde ein Merge-Sort-Join.

Bei der Betrachtung der Ergebnisse wird klar, dass der Profit von etwa 100.000 Timerons im Falle von  $I_{PART}$  seine Gültigkeit verliert, falls  $I_{PARTSUPP}$  vorhanden ist und umgekehrt. Die Anfragepläne auf die hier Bezug genommen wurde, finden sich im Anhang (A.4) wieder.

Für das ISP bedeutet das eben gezeigte Beispiel, dass der vom Optimizer im RECOMMEND\_INDEXES-Modus empfohlene Index  $I_{PART}$  den Profit von 100.000 Timerons nicht mehr erreichen kann, falls, bedingt durch ein anderes Query, der Index  $I_{PARTSUPP}$  bereits gewählt wurde. Da das Beispielquery nach der eventuellen Auswahl von  $I_{PARTSUPP}$  aber nicht neu berechnet wird, bleibt  $I_{PART}$  der Profit von 100.000 Timerons im Auswahlprozess erhalten und es besteht die Gefahr  $I_{PART}$  zu wählen, obwohl er durch das Vorhandensein von  $I_{PARTSUPP}$  keinen Profit mehr erzielt.

Nachdem für Fall 1 ein Beispiel untersucht wurde, soll nun ein Beispiel für den zweiten der eingangs erwähnten Fälle gezeigt werden. Da das hier verwendete Datenbanksystem DB2 Version 9.5 keine Bitmapindexe unterstützt, wird das beschriebene Verhalten an einem Merge-Sort-Join demonstriert. Um die ebenfalls im Anhang gezeigten Anfragepläne zu vereinfachen, wurde die Tabelle LINEITEM aus dem TPC-H-Benchmark kopiert und LINEITEM2 genannt. Anschließend wurde die folgende Anfrage  $Q$  unter verschiedenen Indexkonfigurationen gestellt:

```
select * from lineitem2, lineitem where lineitem2.l_orderkey = lineitem.l_orderkey.
```

Die verwendeten Indexe  $I_1$  und  $I_2$  wurden jeweils über dem Join-Attribut L\_PARTKEY definiert und Anfragen für alle Indexkombinationen gestellt. Die errechneten Aufwände sind in Abbildung 4.1 dargestellt. Der Anfrageaufwand ohne Indexe betrug etwa 4 Millionen Timerons.

Wie zu sehen ist, wird  $profit(Q, \{I_1, I_2\}) > profit(I_1) + profit(I_2)$  erfüllt. Der Profit von etwa 3 Millionen Timerons bei der Verwendung beider Indexe ist um 600.000 Timerons höher als die Summe der Einzelprofite von etwa 1,2 Millionen Timerons.

Ein Auswahlverfahren im ISP, welches nur einen der beiden Indexe wählt, reduziert den Platzbedarf zwar um 50% (beide Indexe haben in diesem Beispiel die gleiche Größe), der Anfrageaufwand steigt im Gegenzug jedoch um einen Faktor von etwa 2,8.

## Präfixindexe und Index-Merging

Eine weitere Indexabhängigkeit entsteht durch *Präfixindexe*. Von Präfixindexen spricht man, wenn ein Index  $I_1$  einen Präfix (eine Vorsilbe) für einen Index  $I_2$  darstellt.  $I_1$  definiert über die Spalten  $(a, b)$  ist beispielsweise ein Präfixindex für den Index  $I_2$ , der über

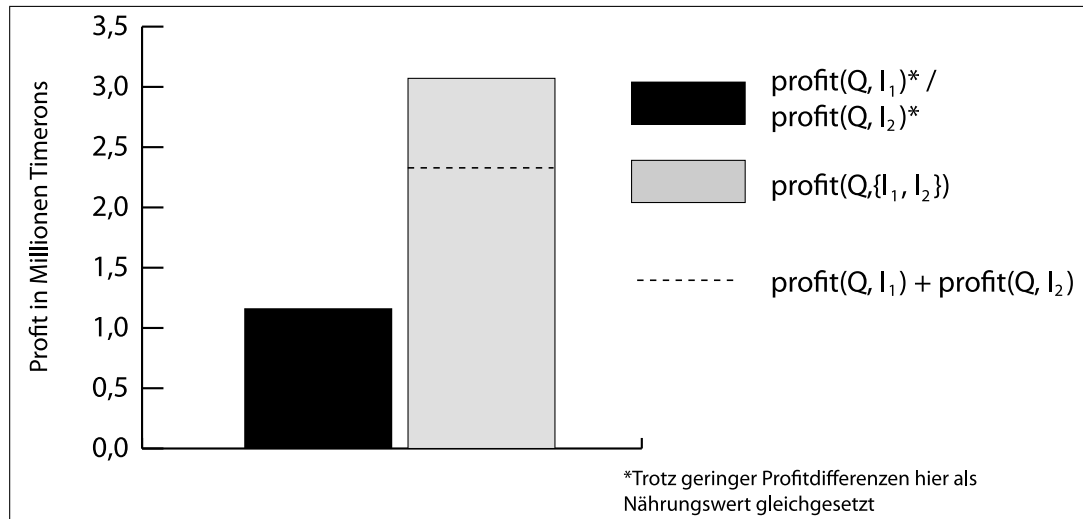


Abbildung 4.1: Indexabhängigkeiten - Merge-Sort-Join

die Spalten  $(a, b, c)$  definiert ist. Anfragen, die durch  $I_1$  beschleunigt werden, erfahren in der Regel durch  $I_2$  eine ähnlich starke Beschleunigung. Die Nutzendifferenz entsteht im Wesentlichen durch den erhöhten Leseaufwand des größeren Indexes  $I_2$ . Anfragen die hingegen alle Spalten von  $I_2$  für die effiziente Bearbeitung erfordern, können durch  $I_1$  nicht ähnlich schnell beantwortet werden.

Die Idee der Präfixindexe wird beim *Index-Merging* [CN99] erweitert. Beim Index-Merging wird durch das Verschmelzen von Indexen versucht eine Platzersparnis zu erzielen und die Anfrageeigenschaften der verschmolzenen Indexe zu erhalten. Werden zwei Indexe  $I_1$  und  $I_2$  verschmolzen, so ist einer der beiden Indexe ein Präfixindex des neu gebildeten Indexes  $I_3$ . Die Indexe  $I_1 = (a, b)$  und  $I_2 = (b, c)$  werden beim Index-Merging beispielsweise zum neuen Index  $I_3 = (b, c, a)$  verschmolzen. Mit  $I_3$  kann zwar eine Indexsuche mehr über  $a$  durchgeführt werden, aber  $a$  und  $b$  können effizient aus dem Index gelesen werden (Index-Scan).

Für das ISP werden Präfixindexe vor allem bei der Optimierung mehrerer Anfragen interessant, da Abhängigkeiten queryübergreifend auftreten. Mit der Auswahl eines Indexes  $I_2 = (a, b, c)$  verringert sich der Nutzen eines Indexes  $I_1 = (a, b)$ . Der DB2 Design Advisor sucht bei dem in Abschnitt 3.2.3 beschriebenen Auswahlverfahren nach Präfixindexen (Schritt 7), so dass diese in der Ergebnismenge in der Regel nicht auftauchen.

### Weitere Betrachtungen

Während die genannten Abhängigkeiten leicht nachvollziehbar sind, sind weitere Situationen vorstellbar, in denen  $\text{profit}(Q, \mathcal{I}) < \sum_{I \in \mathcal{I}} \text{profit}(I)$  beziehungsweise  $\text{profit}(Q, \mathcal{I}) > \sum_{I \in \mathcal{I}} \text{profit}(I_n)$  gilt. So könnten zum Beispiel Sortierungen oder logisch verknüpfte Prädikate Abhängigkeiten induzieren.

Eine korrekte Bewertung des Nutzens von Indexkombinationen beziehungsweise von Indexabhängigkeiten in einer Indexkonfiguration  $\mathcal{I} = \{I_1, I_2, I_3, I_4\}$  ist vorstellbar, wenn der Nutzen aller möglichen Kombinationen durch den Optimizer berechnet wird. Dies führt jedoch zu einer „kombinatorischen Explosion“ in Bezug auf die Anzahl der zu

testenden Kombinationen<sup>3</sup>, so dass diese Überlegung eher theoretischer Natur ist.

Um Indexabhängigkeiten und dem anschließend vorgestellten Problem der Profitberechnung einzelner Indexe  $I$  in Konfigurationen  $\mathcal{I}$  Rechnung zu tragen, wurde im DB2 Design Advisor mit TRY\_VARIATION (siehe Abschnitt 3.2.3) eine Strategie ähnlich der oben genannten Kombinationsbildung einhergehend mit neuen Nutzenberechnungen eingeführt. Im folgenden Abschnitt 4.1.3 wird die Wirksamkeit von TRY\_VARIATION kurz betrachtet.

Trotz TRY\_VARIATION führt die Analyse des Auswahlverfahrens im DB2 Design Advisor zu dem Schluss, dass Indexabhängigkeiten der Form  $profit(Q, \mathcal{I}) > \sum_{I \in \mathcal{I}} profit(I)$  nicht berücksichtigt werden, da die Profitberechnung von Indexten die Existenz anderer Indexe nicht berücksichtigt.

### 4.1.3 Profitberechnung einzelner Indexe

Neben Indexabhängigkeiten besteht beim ISP das Problem der Profitberechnung einzelner Indexe. Im Fall von DB2 werden durch den RECOMMEND\_INDEXES-Modus in der Regel mehrere Indexe  $I$  für eine Anfrage  $Q$  zurückgeliefert. Für jedes Query  $Q_n$  eines Workloads entsteht somit eine lokal-optimale Indexkonfiguration  $\mathcal{I}_n$ , die vom Optimizer berechnet wird. Der Profit einer solchen Indexkonfiguration ist, wie bereits beschrieben, die Kostendifferenz  $profit(Q, \mathcal{I}_n) = cost(Q_n) - cost(Q_n, \mathcal{I}_n)$ . Sind für alle Queries  $Q_n$  die Indexkonfigurationen  $\mathcal{I}_n$  berechnet, wird im DB2 Design Advisor jedem Index  $I$  der gesamte Profit aller Indexkonfigurationen  $\mathcal{I}_n$  zugewiesen für die gilt  $I \in \mathcal{I}_n$ . Im Folgenden soll nun untersucht werden, welche Auswirkungen diese Profitzuteilung auf die Indexauswahl hat.

Gegenstand der Betrachtungen sind zwei Queries (I und II), die dem DB2 Design Advisor als Workload  $W$  übergeben wurden. Die Queries sind dem TPC-H-Workload entnommen wurden und in den Abbildungen 4.2 und 4.3 dargestellt. Es ist anzumerken, dass vor den durchgeführten Tests alle Indexe und Schlüsselbeziehungen der betroffenen Tabellen entfernt wurden. Alle Kosten und Profite sind in Timern angegeben.

```
select sum(l_extendedprice* (1 - l_discount)) as revenue from lineitem, part
where ( p_partkey = l_partkey and p_brand = 'Brand#42' and p_container in
('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG') and l_quantity >= 6 and l_quantity <=
6 + 10 and p_size between 1 and 5 and l_shipmode in ('AIR', 'AIR REG') and
l_shipinstruct = 'DELIVER IN PERSON' ) or ( p_partkey = l_partkey and p_brand
= 'Brand#23' and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
and l_quantity >= 10 and l_quantity <= 10 + 10 and p_size between 1 and
10 and l_shipmode in ('AIR', 'AIR REG') and l_shipinstruct = 'DELIVER IN PERSON' )
or ( p_partkey = l_partkey and p_brand = 'Brand#52' and p_container in ('LG
CASE', 'LG BOX', 'LG PACK', 'LG PKG') and l_quantity >= 26 and l_quantity <= 26 + 10
and p_size between 1 and 15 and l_shipmode in ('AIR', 'AIR REG') and
l_shipinstruct = 'DELIVER IN PERSON' )
```

Abbildung 4.2: Query I

<sup>3</sup>Es gilt in diesem Falle  $\binom{n}{k}$  für die Anzahl der Möglichkeiten  $k$  verschiedene Indexe aus  $n$  möglichen zu wählen.

```

select supp_nation, cust_nation, l_year, sum(volume) as revenue from (
  select n1.n_name as supp_nation, n2.n_name as cust_nation, Year(l_shipdate) as l_year,
    l_extendedprice * (1 - l_discount) as volume from supplier, lineitem, orders,
    customer, nation n1, nation n2
  where s_suppkey = l_suppkey and o_orderkey = l_orderkey
  and c_custkey = o_custkey and s_nationkey = n1.n_nationkey
  and c_nationkey = n2.n_nationkey and ((n1.n_name
  = 'UNITED KINGDOM' and n2.n_name = 'INDONESIA')
  or (n1.n_name = 'INDONESIA' and n2.n_name = 'UNITED
  KINGDOM')) and l_shipdate between date ('1995-01-01')
  and date ('1996-12-31')
) as shipping
group by supp_nation, cust_nation, l_year
order by supp_nation, cust_nation, l_year

```

Abbildung 4.3: Query II

### Optimale Lösung

Beginnen sollen die Betrachtungen mit der vom RECOMMEND\_INDEXES-Modus berechneten Lösung  $C_{opt}$ , die auch vom DB2 Design Advisors (ohne Verwendung einer Größenschranke) vorgeschlagen wird. Diese Lösung wird als optimal betrachtet und beinhaltet die beiden lokal-optimalen Indexkonfigurationen  $\mathcal{I}_{I_{opt}}$  und  $\mathcal{I}_{II_{opt}}$ .

Kosten und Profit der beiden untersuchten Queries sind in Tabelle 4.2 dargestellt, die auch Informationen über die Größe der jeweils lokal-optimalen Indexkonfiguration  $\mathcal{I}$  enthält. Die Größe der in den Queries referenzierten Relationen beläuft sich auf etwa einen Gigabyte.

Query	$cost(Q_x)$	$cost(Q_x, \mathcal{I}_{x_{opt}})$	$profit(Q_x, \mathcal{I}_{x_{opt}})$	$size(\mathcal{I}_x)$
$Q_I$	217.784	7.935	209.849	53,7
$Q_{II}$	290.827	71.385	219.442	127,23
Gesamt	508.611	79.320	429.291	180,93

Tabelle 4.2: Optimale Lösung für  $Q_I$  und  $Q_{II}$ 

Die optimale Lösung enthält insgesamt sechs verschiedene Indexe ohne Präfixabhängigkeiten. In Tabelle 4.3 sind unter anderem Informationen über Relation, Größe und relativen Profit der Indexe angegeben. Der relative Profit beschreibt das Indexprofit/Indexgröße Verhältnis, welches dem DB2 Design Advisor als Auswahlkriterium dient. Wie bereits mehrfach beschrieben wurde, wird beim DB2 Design Advisor allen Indexen  $I$  einer lokal-optimalen Indexkonfiguration  $\mathcal{I}$  der gesamte Profit zugeordnet, der  $\mathcal{I}$  verbunden ist. Abbildung 4.4 zeigt eine gekürzte Ausgabe des db2advis-Tools für Query I. Die Profitzuweisung (Benefit) entspricht den bisher gemachten Aussagen.

Beim Betrachten der relativen Indexprofite stellt man fest, dass kleine Indexe offensichtlich überbewertet werden. Ursache dafür ist die vom DB2 Design Advisor vorgenommene Profitberechnung der einzelnen Indexe. Die in QUIET vorgeschlagenen Berechnungsmethoden (siehe Abschnitt 3.3) liefern hier vermutlich ein besseres Ergebnis. Eine optimale Profitzurechnung ist erneut nur durch die Kostenberechnung aller möglichen

```

...
<index>
<identifier><name>IDX809141734310000</name></identifier>
<table><identifier><name>PART</name></identifier></table>
<benefit>209849,000000</benefit>
<diskspace>1,247094</diskspace>
</index>
...
<index>
<identifier><name>IDX809141734290000</name></identifier>
<table><identifier><name>LINEITEM</name></identifier></table>
<benefit>209849,000000</benefit>
<diskspace>52,555687</diskspace>
</index>
...

```

Abbildung 4.4: Profitzuweisung DB2

<i>Index</i>	<i>Indexkonfig.</i>	<i>Relation</i>	<i>size(I) (MB)</i>	<i>rel.Profit</i>
$I_{I_1}$	$\mathcal{I}_{I_{opt}}$	LINEITEM	52,5	3.997
$I_{I_2}$	$\mathcal{I}_{I_{opt}}$	PART	1,2	174.874
$I_{II_1}$	$\mathcal{I}_{II_{opt}}$	CUSTOMER	2,47	88.842
$I_{II_2}$	$\mathcal{I}_{II_{opt}}$	ORDERS	24,6	8.920
$I_{II_3}$	$\mathcal{I}_{II_{opt}}$	NATION	0,16	1.371.512
$I_{II_4}$	$\mathcal{I}_{II_{opt}}$	LINEITEM	100	2.194

Tabelle 4.3: Indexe in  $\mathcal{I}_{I_{opt}}$  und  $\mathcal{I}_{II_{opt}}$ 

Indexkombinationen möglich, die aber außer Frage steht.

## Lösungen unter verschiedenen Größenschranken

Es soll nun überprüft werden, welche Indexe vom DB2 Design Advisor ausgewählt werden, wenn es zu Änderungen der Größenschranke kommt. Dazu werden für  $Q_I$  und  $Q_{II}$  nun Indexempfehlungen für Größenschranken von 160 und 60 Megabyte erzeugt. Um eventuell bessere Lösungen zu finden, wurden die beiden Queries zusätzlich unter verschiedenen Indexkonfigurationen im EVALUATE\_INDEXES-Modus ausgeführt.

Bevor auf die Ergebnisse unter Berücksichtigung der einzelnen Größenschranken eingegangen werden soll, werden in den Abbildungen 4.5 und 4.6 zunächst die (vereinfachten) Zugriffspläne mit und ohne Verwendung der optimalen Indexe angezeigt.

Jeweils in Klammern steht der Aufwand, der vom Optimizer geschätzt wird. Der Aufwand besteht aus dem kumulierten Aufwand des Anfrageplanes bis zu einem bestimmten Knoten, einschließlich des Aufwandes der mit dem Knoten verbunden Operation. Nähere Informationen über die Zugriffspläne und Operationen können dem DB2 Infocenter [IBM08] entnommen werden. Bei *GENROW*, zu sehen in Abbildung 4.5, handelt es sich um virtuelle Tabellen, die im Fall von  $Q_I$  Werte für *in*-Prädikate enthalten (zum Beispiel: `Lshipmode` in ('AIR', 'AIR REG')).

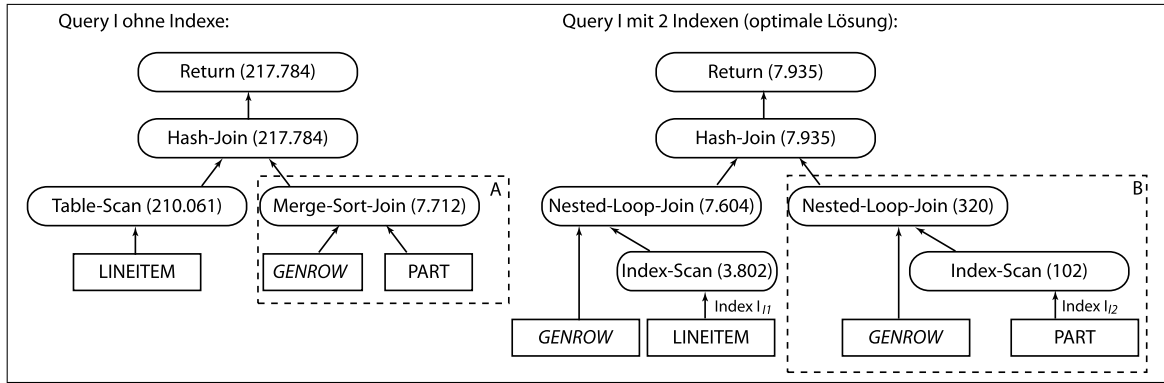


Abbildung 4.5: Anfragepläne Query I

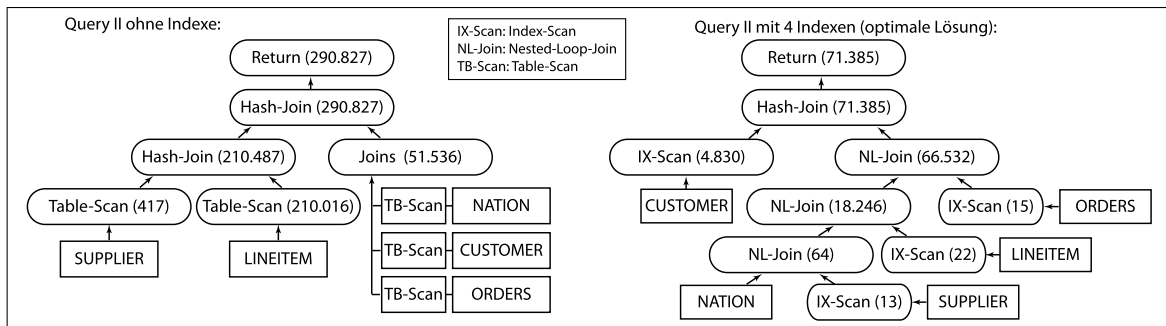


Abbildung 4.6: Anfragepläne Query II

### Einschränkung auf 160 Megabyte

Die Indexauswahl des DB2 Design Advisors entfernt bei der Einschränkung auf 160 Megabyte den Index mit dem schlechtesten relativen Profit (siehe Tabelle 4.3)  $I_{II_4}$ . Während die Kosten von  $Q_I$  unverändert bleiben ( $\mathcal{I}_{I_{opt}} = \mathcal{I}_{I_{160}}$ ), liefert die lokale Indexkonfiguration  $\mathcal{I}_{II_{160}}$  für  $Q_{II}$  nur noch einen Profit von 27.243 Timerons.

Ist dieses Ergebnis optimal? Bei der Betrachtung des Anfrageplanes in Abbildung 4.6 wird deutlich, dass der Table-Scan auf der Relation LINEITEM die teuerste Operation darstellt. Durch den Index  $I_{II_4}$  und der damit einhergehenden Planumformung, geschieht das Lesen von LINEITEM jedoch ohne nennenswerten Aufwand. Es ist zu vermuten, dass  $I_{II_4}$  den Großteil des Profites der lokal-optimalen Lösung  $I_{II_{opt}}$  ausmacht. Um dieser Vermutung nachzugehen, wurde  $Q_{II}$  mit dem EVALUATE\_INDEXES-Modus unter der alternativen lokalen Indexkonfiguration  $\mathcal{I}_{II_{160alt}} = \{I_{II_4}\}$  ausgeführt, die  $\mathcal{I}_{II_{160}}$  in der Lösung ersetzen könnte, ohne die Größenschränke zu überschreiten. Die Anfragepläne unter beiden lokalen Indexkonfigurationen sind in Abbildung 4.7 verkürzt dargestellt.

Die grau unterlegten Knoten im Anfragebaum zeigen die Auswirkungen des Indexes  $I_{II_4}$  auf die Kosten beim Bearbeiten von LINEITEM. Die Kosten  $cost(Q_{II}, \mathcal{I}_{II_{160alt}})$  sind um etwa 160.000 Timerons günstiger, als die der vom DB2 Design Advisor vorgeschlagenen Indexkonfiguration  $\mathcal{I}_{II_{160}}$ . Um die Vergleichbarkeit der hier getesteten globalen Indexkonfigurationen zu verbessern, zeigt Abbildung 4.8 auf Seite 49 die Bearbeitungszeit des Workloads  $W$  in Sekunden.

Die vom DB2 Design Advisor vorgeschlagene globale Indexkonfiguration  $C_{adv_{160}} = \{\mathcal{I}_{I_{160}}, \mathcal{I}_{II_{160}}\}$  stellt also keine gute Lösung dar. Dem Argument, dass  $\mathcal{I}_{II_{160alt}}$  nur durch

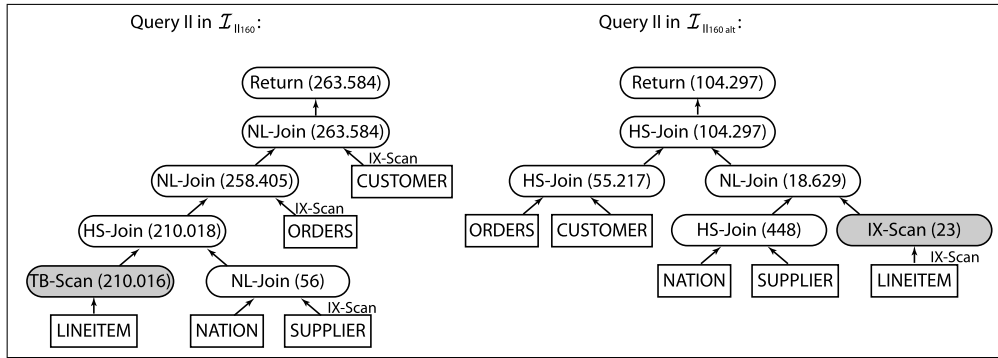


Abbildung 4.7: Anfragepläne Query II

(unerwünschten) zusätzlichen Berechnungsaufwand gefunden wurde, kann man entgegenhalten, dass TRY\_VARIATION die vorgegebene Zeitgrenze von 10 Minuten nicht ausnutzte. Die Berechnungen der Advisor-Lösung dauerte in allen Testläufen weniger als eine Minute.

Im Folgenden soll die Indexauswahl für eine Größenschranke von 60 Megabyte untersucht werden.

### Einschränkung auf 60 Megabyte

Mit der Einschränkung der Lösungsmenge auf eine Größe von 60 Megabyte liefert der DB2 Design Advisor erneut das vermutete Ergebnis. Die Indexe werden nach ihrem jeweiligen relativen Profit sortiert und ausgewählt. Es bestätigt sich also die Vermutung, dass kleine Indexe durch das Auswahlverfahren des DB2 Design Advisors bevorzugt werden. Die vorgeschlagene globale Indexkonfiguration  $C_{adv_{60}} = \{I_{I_{60}}, I_{II_{60}}\}$  mit  $I_{I_{60}} = \{I_{I_2}\}$  und  $I_{II_{60}} = I_{II_{160}}$  liefert bei einer Größe von 28,5 Megabyte einen Profit von etwa 35.000 Timerons für den Workload  $W$ .

Da  $I_{II_{60}} = I_{II_{160}}$  ist auch die Anfragebearbeitung für  $Q_{II}$  mitsamt dem Teilprofit von 27.243 Timerons identisch zur Konfiguration  $C_{adv_{160}}$ . Die Kosten  $cost(Q_I, I_{I_{60}})$  belaufen sich auf 210.392 Timerons. Für  $I_{I_{60}}$  ergibt sich also ein Profit von 7.392 Timerons für  $Q_I$ . Der Anfrageplan für  $Q_I$  ändert sich im Vergleich zu Abbildung 4.5 auf Seite 47 durch den Austausch des Teilplanes  $A$  mit dem Teilplan  $B$ .

Vergleicht man Profit von  $C_{adv_{60}} = 34.635$  Timerons mit dem der lokal-optimalen Lösung  $I_{I_{opt}} = 209.849$  Timerons, so erkennt man, dass erneut eine Verbesserung des Gesamtergebnisses unter Einhaltung der Größenschranke ( $size(I_{I_{opt}}) < 60MB$ ) möglich ist. Die alternative Indexkonfiguration  $C_{alt_{60}} = \{I_{I_{opt}}\}$  verbessert das durch den Advisor erzielte Ergebnis  $C_{adv_{60}}$  um etwa 175.000 Timerons. Auch hier sei auf Abbildung 4.8 verwiesen, um ein besseres Bild der Leistungsverbesserung zu erhalten.

### Praxistest

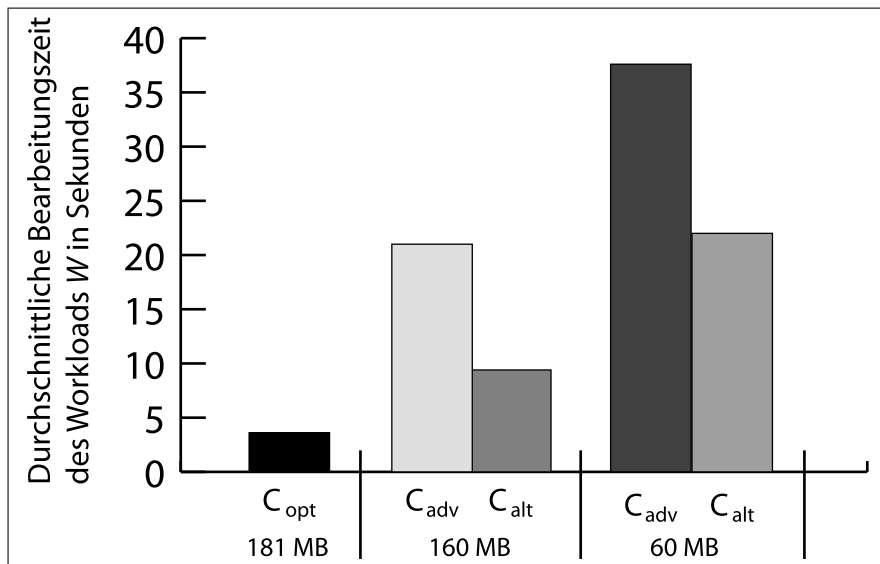
Bisher wurde für alle Indexkonfigurationen nur der geschätzte Kostenaufwand beziehungsweise Profit in Timerons berechnet. Da Timerons laut Definition [IBM08] (siehe Seite 41) nur einen geschätzten Aufwand wiedergeben, wurden die ermittelten globalen Indexkonfigurationen  $C$  materialisiert und unter Zuhilfenahme des *db2batch*-Tools [IBM08] in

Bezug auf die Bearbeitungsgeschwindigkeit des Workloads  $W$  untersucht. Bevor in Abbildung 4.8 die Ergebnisse dieser Untersuchungen dargestellt werden, sollen in Tabelle 4.4 die Indexkonfigurationen noch einmal zusammengefasst werden.

$C$	$\mathcal{I}$	$cost(W, C)$	$profit(W, C)$	$size(C)$ (MB)
$C_0$	$\emptyset$	508.611	0	0
$C_{opt}$	$\mathcal{I}_{I_{opt}} = \{I_{I_1}, I_{I_2}\},$ $\mathcal{I}_{II_{opt}} = \{II_{I_1}, II_{I_2}, II_{I_3}, II_{I_4}\}$	79.320	429.291	180,93
$C_{adv_{160}}$	$\mathcal{I}_{I_{opt}} = \{I_{I_1}, I_{I_2}\},$ $\mathcal{I}_{II_{160}} = \{II_{I_1}, II_{I_2}, II_{I_3}\}$	271.519	237.092	80,93
$C_{alt_{160}}$	$\mathcal{I}_{I_{opt}} = \{I_{I_1}, I_{I_2}\},$ $\mathcal{I}_{II_{160alt}} = \{II_{I_4}\}$	112.232	396.379	153,70
$C_{adv_{60}}$	$\mathcal{I}_{I_{60}} = \{I_{I_2}\},$ $\mathcal{I}_{II_{60}} = \{II_{I_1}, II_{I_2}, II_{I_3}\}$	473.976	34.635	28,43
$C_{alt_{60}}$	$\mathcal{I}_{I_{opt}} = \{I_{I_1}, I_{I_2}\},$	298.762	209.849	53,70

Tabelle 4.4: Indexkonfigurationen für  $W$ 

Kosten und Profite sind in der Tabelle erneut in Timerons angegeben, während die Größe in Megabyte beschrieben wird. Im Folgenden wird nun die durchschnittliche Bearbeitungszeit des Workloads  $W$  in Sekunden angezeigt. Die Ergebnisse entsprechen der durchschnittlichen Bearbeitungszeit ermittelt aus jeweils 5 Testläufen pro Indexkonfiguration. Um die Auswirkungen der Pufferverwaltung zu minimieren, wurde das Datenbanksystem vor jedem Testlauf neu gestartet. Die Bearbeitungszeit von  $W$  ohne Indexe betrug in den Tests durchschnittlich 41,8 Sekunden.

Abbildung 4.8: Durchschnittliche Bearbeitungszeiten  $W = \{Q_I, Q_{II}\}$ 

Die Abbildung zeigt, dass die Aufwandsabschätzung des Optimizers den tatsächlichen Bearbeitungsaufwand für Anfragen sehr gut widerspiegelt. Ein Entscheidungs-

dell zur Indexauswahl kann also auf Basis der Kostenabschätzungen in Timerons arbeiten. Es bestätigt sich außerdem, dass die alternativen Indexkonfigurationen der gegebenen Größenschranken den Workload schneller abarbeiten als die vorgeschlagenen Indexkonfigurationen des DB2 Design Advisors. Die Advisor-Lösung  $C_{adv_{60}}$  liefert beispielsweise einen Vorteil von wenigen Sekunden, während die alternative Lösung  $C_{alt_{60}}$  ähnlich gut ist wie die Advisor-Lösung  $C_{adv_{160}}$ .

#### 4.1.4 Zusammenfassung

In Kapitel 4.1 wurde zunächst das Rucksackproblem kurz beschrieben. In diesem Zusammenhang wurde die Abhängigkeit von Objekten eines Rucksackes diskutiert. Anschließend wurden Indexe auf Abhängigkeiten hin untersucht. Dass der Profit von Indexen vom Vorhandensein anderer Indexe abhängt, wurde dabei in einigen Beispielen gezeigt. Bei der anschließenden Analyse des DB2 Design Advisor stellte sich heraus, dass das Auswahlverfahren keine optimalen Ergebnisse liefert. Die Auswahlfehler entstehen durch die Profitzuweisung einzelner Indexe. Da der DB2 Design Advisor seine Indexauswahl vom Profit einzelner Indexe beziehungsweise deren relativen Profites abhängig macht, ist davon auszugehen, dass auch Indexabhängigkeiten unberücksichtigt bleiben. Obwohl mit TRY\_VARIATION eine Methode entwickelt wurde, verschiedene Objektkombinationen auf deren Profit hin zu testen, wurden nur Lösungen gefunden, die sich durch einfache Analyse der Anfragepläne verbessern ließen.

Als Zwischenschritt bei der Erstellung einer Indexauswahl erstellt der DB2 Advisor im RECOMMEND\_INDEXES-Modus lokale-optimale Indexkonfigurationen für jedes einzelne Query  $Q$  eines gegebenen Workloads. Man kann diese Indexkonfigurationen als kleine Rucksäcke von bestimmtem Nutzen/Profit verstehen. Beim Versuch, Objekte aus diesen Rucksäcken zu entfernen, kommt es zu einer Fehleinschätzung des Profites der einzelnen Objekte. Den Rucksack aus dem Beispiel des Abschnittes 4.1.1 würde der DB2 Design Advisor wohl immer zuerst mit Messer und Decke befüllen beziehungsweise Brot und Brotaufstrich entfernen, da diese Objekte die Leichtesten sind und trotzdem den Profit des optimalen Rucksackes bekämen. Der hungrige Wandersmann wird mit dieser Auswahl allerdings nicht zufrieden sein.

## 4.2 Konzept

Motiviert durch die Ergebnisse aus Kapitel 4.1 wurde im Rahmen dieser Arbeit ein alternatives Indexauswahlverfahren entwickelt, welches im Folgenden vorgestellt wird. Nach der Formulierung von Anforderungen und Zielen, kommt es zur Beschreibung des entwickelten Auswahlverfahrens. Anschließend daran werden die Nachteile des Verfahrens diskutiert. Abschließend soll untersucht werden, ob sich das neue Verfahren als online-Lösung ähnlich dem QUIET-System verwenden lässt.

### 4.2.1 Anforderungen und Ziele

Ziel des Verfahrens ist es zu zeigen, dass Lösungen von Indexauswahlverfahren, die dem des DB2 Design Advisors entsprechen, durch:

1. Die Vermeidung von Indexfehlauswahlen auf Grund von fehlerhaften Einzelprofitberechnungen und
2. Die Berücksichtigung von Indexabhängigkeiten der Form  $profit(Q, \mathcal{I}) > \sum_{I \in \mathcal{I}} profit(I_n)$  in lokal-optimalen Indexkonfigurationen  $\mathcal{I}$

verbessert werden können.

Die wesentliche Anforderung an das neue Verfahren ist es somit, die beiden genannten Punkte bei der Auswahl von Indexen zu berücksichtigen. Weitere Anforderungen leiten sich durch den angestrebten Vergleich mit dem DB2 Design Advisor und dem ISP ab. Es muss beispielsweise möglich sein Größenschranken für die Lösung vorzugeben. Ferner soll die Lösungsfindung mit möglichst geringem Aufwand erfolgen. Insbesondere gilt es zusätzlichen Aufwand durch EXPLAIN- beziehungsweise EVALUATE\_INDEXES Anfragen zu vermeiden.

Die Menge von Indexen aus denen das Verfahren eine Lösung konstruiert, muss durch das Datenbanksystem selbst ermittelt werden. Für die Anwendung des Verfahrens ist also ein Optimizer notwendig, der über einen RECOMMEND\_INDEXES oder vergleichbaren Modus verfügt und für gegebene Queries Indexe empfiehlt, die als optimal angesehen werden können.

## 4.2.2 LISA - Einleitung

Es soll an dieser Stelle der grundlegende Gedanke des entwickelten Verfahrens, welches im Folgenden *LISA* bezeichnet wird, vorgestellt werden. *LISA* steht kurz für „Local Indexset Assortment“ und verrät damit den Grundgedanken der Indexauswahl auf Basis lokal-optimaler Indexkonfigurationen.

Als lokal-optimale Indexkonfiguration  $\mathcal{I}$  wurde die Gesamtheit der Indexe  $I$  definiert, die vom Optimizer im RECOMMEND\_INDEXES-Modus für ein einzelnes Query  $Q$  vorgeschlagen werden. Der Profit einer solchen Indexkonfiguration  $\mathcal{I}$  lässt sich durch einfache Subtraktion  $cost(Q) - cost(Q, \mathcal{I})$  bestimmen. Durch die Verwendung des Optimizers zum Erstellen von  $\mathcal{I}$  ist garantiert, dass die Indexe  $I \in \mathcal{I}$  bei der Anfragebearbeitung auch tatsächlich verwendet werden. Zwar besteht die Möglichkeit durch externe Kostenmodelle Indexkonfigurationen zu finden, die einen höheren Profit versprechen, es ist jedoch nicht sichergestellt, dass die Indexe dieser Konfiguration vom Optimizer bei der Anfragebearbeitung genutzt werden (siehe [FST88]).

Bei der Auswahl einzelner Indexe  $I$  aus der Konfiguration  $\mathcal{I}$  ergibt sich nun das Problem, welchen Einfluss einzelne Indexe auf den Profit der Konfiguration haben. Wie in Abschnitt 4.1.3 zu sehen war, kommt es hier zu großen Unterschieden zwischen den Indexen. Es wurde gezeigt, dass Profitzuweisung des DB2 Design Advisors ungenau ist. Auch andere Berechnungsverfahren (siehe Abschnitt 3.3) können Einzelindexprofite nur schätzen. Problematisch sind ausserdem Abhängigkeiten zwischen Indexen, die bei der Berechnung von Profiten einzelner Indexe unberücksichtigt bleiben.

Die Berechnung von Einzelprofiten könnte durch Analyse von Anfragepläne erfolgen. Beim Vergleich des ohne Indexe erzeugten Anfrageplanes mit dem Anfrageplan unter einer Indexkonfiguration  $\mathcal{I}$  könnten besonders kostenintensive Operationen identifiziert und die Auswirkungen eines einzelnen Indexes quantifiziert werden. In Abbildung 4.7 ist

jedoch zu beobachten, dass sich Anfragepläne mitunter stark verändern, falls Indexe aus einer Konfiguration entfernt werden. Zuverlässigen Ergebnisse können somit also nicht gewonnen werden. Das Problem der Indexabhängigkeiten bleibt auch durch die Analyse von Anfrageplänen ungelöst.

Auf Grund der Probleme, die beim Zerlegen lokal-optimaler Indexkonfigurationen auftreten, entstand die Idee, mit LISA ein Indexauswahlverfahren zu schaffen, welches auf diese Zerlegung verzichtet. Gegenstand der Indexauswahl sind damit keine einzelnen Indexe mehr, sondern nur noch lokal-optimale Indexkonfigurationen  $\mathcal{I}$ . Abbildung 4.9 illustriert noch einmal die angesprochenen Probleme.

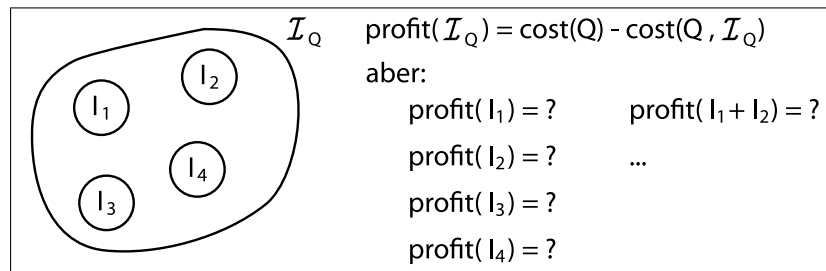


Abbildung 4.9: Lokal-optimale Indexkonfigurationen

Das ein Auswahlverfahren wie LISA bessere Ergebnisse als die des DB2 Design Advisors liefern kann, wird beim Vergleich der globalen Indexkonfigurationen  $C_{adv60}$  und  $C_{alt60}$  aus Abschnitt 4.1.3 deutlich. Die Lösung  $C_{alt60}$  besteht in diesem Fall aus einer lokal-optimalen Indexkonfiguration und erzielt einen größeren Profit als eine Auswahl einzelner Indexe aus den Indexkonfigurationen  $\mathcal{I}_{I_{opt}}$  und  $\mathcal{I}_{II_{opt}}$ .

### 4.2.3 LISA - Statistiken lokaler Indexkonfigurationen

Aus der Verwendung lokaler Indexkonfigurationen als Objekte der Indexauswahl ergibt sich die Notwendigkeit neue Statistiken und Entscheidungsgrößen einzuführen. Die Betrachtung von Indexgrößen und Indexprofiten wie es im DB2 Design Advisor der Fall ist, reicht nicht aus, um das Problem der Indexauswahl in LISA abzubilden.

Die einfachste Lösung wäre es, in Analogie zum DB2 Design Advisor, anstatt für Indexe nun Größe und Profit von Indexkonfigurationen zu speichern. Das eine solche Lösung zu ungenauen Ergebnissen führt, wird in Abbildung 4.10 sichtbar.

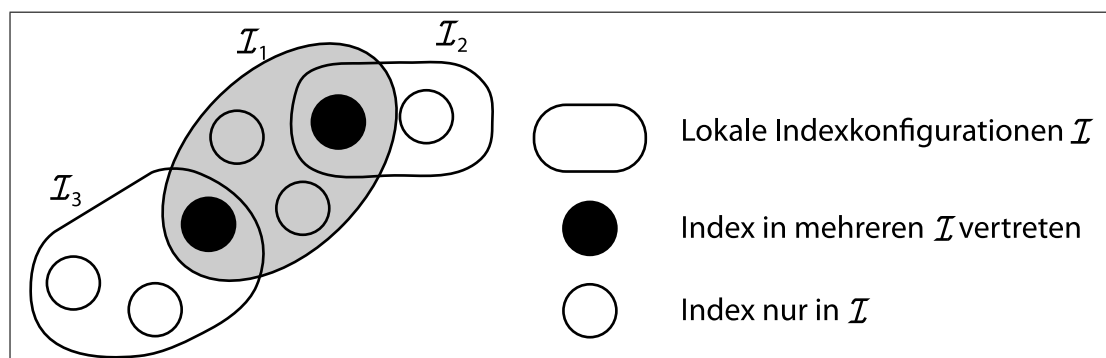


Abbildung 4.10: Überlappung lokaler Indexkonfigurationen

Neben Abhängigkeiten zwischen Indexen entstehen durch LISA neue Abhängigkeiten in Form von Überlappungen zwischen lokalen Indexkonfigurationen. Die Materialisierung der Konfiguration  $\mathcal{I}_1$  in Abbildung 4.10 hat in zweierlei Hinsicht Auswirkungen auf die Konfigurationen  $\mathcal{I}_2$  und  $\mathcal{I}_3$ . Wird  $\mathcal{I}_1$  ausgewählt, so verringert sich der Speicherplatzbedarf der beiden anderen gezeigten Konfigurationen um die Größe der gemeinsamen, durch die jeweils schwarzen Kreise gekennzeichneten, Indexe. Sollten  $\mathcal{I}_2$  oder  $\mathcal{I}_3$  ebenfalls für die Lösung ausgewählt werden, darf es zu keiner doppelten Zählung der Größe des jeweils überlappenden Indexes kommen. Auch hinsichtlich des Profites einer lokalen Indexkonfiguration entstehen Probleme durch Überlappungen. In LISA sollen deswegen zwei verschiedene Möglichkeiten bezüglich der Profitanpassung bei Überlappungen betrachtet werden.

Zur schnellen Lösungsfindung wird keine Profitanpassung in Überlappungssituationen vorgenommen. Es kommt also lediglich zu einer Anpassung der Konfigurationsgröße. Um bessere Lösungen zu finden, kann durch LISA aber auch eine Neuberechnung des Profites einer Konfiguration im EVALUATE\_INDEXES-Modus durchgeführt werden. Dieser Schritt macht es notwendig, die mit der Konfiguration verbundene Anfrage, samt deren Kosten ohne Indexe, zu speichern. Ist ein Index  $I_{mat}$  einer nicht ausgewählten Konfiguration  $\mathcal{I}_n$  in einer ausgewählten Konfiguration  $\mathcal{I}_a$  vorhanden, so ergibt sich der neue Profit wie folgt:

$$profit(Q, \mathcal{I}_n)_{neu} = cost(Q) - cost(Q, \mathcal{I}_n \setminus I_{mat})$$

Im schlimmsten, wenn auch sehr unwahrscheinlichen, Fall, überlappen sich alle lokalen Indexkonfigurationen so, dass bei Erstellung der Lösungsmenge  $C = \{\mathcal{I}_a, \mathcal{I}_b, \dots\}$  das Hinzufügen einer neuen lokalen Konfiguration  $\mathcal{I}_n$  zur Neuberechnung aller Konfigurationen  $\mathcal{I} \notin C$  führt. In einem solchen Fall sind für  $n$  lokale Indexkonfigurationen also:

$$\sum_{k=1}^n (n - k)$$

EXPLAIN-Anfragen, in Form des EVALUATE\_INDEXES-Modus, notwendig. Durch Indexkonfigurationen, die für mehrere Queries optimal sind, kann sich diese Summe weiter erhöhen. Um die Laufzeit von LISA kontrollieren zu können, kann die maximale Anzahl `MAX_EXPLAIN` der (RE-)EXPLAIN-Anfragen vorgegeben werden. Diese Idee leitet sich vom `TRY_VARIATION`-Algorithmus des DB2 Design Advisors ab, dem eine maximale Laufzeit vorgegeben werden kann.

Abbildung 4.11 beschreibt konkret, wie bei der Auswahl der grau hinterlegten Indexkonfiguration  $\mathcal{I}_1$  (siehe auch Abbildung 4.10) auf Überlappungen reagiert wird.

Um eine effiziente Neuberechnung von Profit und Größe lokaler Indexkonfigurationen zu gewährleisten, müssen bestimmte Informationen über Indexkonfigurationen, und Indexe selbst, vorhanden sein. Wie in Abbildung 4.11 zu sehen ist, müssen Indexkonfigurationen ihre zugehörigen Indexe „kennen“ und für Indexe muss hinterlegt sein, welchen Konfigurationen sie angehören. Indexinformationen sind des weiteren erforderlich um die Größe von lokalen Indexkonfigurationen zu berechnen und festzustellen ob zwei lokale Indexkonfigurationen identisch sind. Letzteres ist notwendig, da für verschiedene Anfragen  $Q$  die gleichen Indexe  $I$ , also auch gleiche lokal-optimale Indexkonfigurationen  $\mathcal{I}$ , empfohlen werden können. Weitere Statistiken wie zum Beispiel Profit von Indexkonfi-

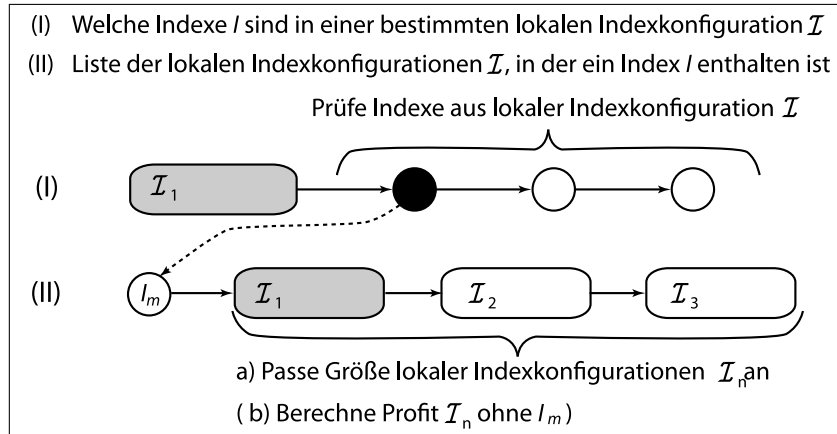


Abbildung 4.11: Behandlung von Überlappungen

gurationen oder Informationen, ob ein Indexe beziehungsweise eine Indexkonfiguration der Lösung  $C$  bereits angehört, ergeben sich trivial.

In Tabelle 4.5 sind die Informationen über Indexe  $I$  und Indexkonfigurationen  $\mathcal{I}$  aufgeführt, die von LISA für die Indexauswahl benötigt werden.

Objekt	Information	Beschreibung
$\mathcal{I}$	Profit	aktueller Profit (ggf. Anpassung durch RE-EXPLAINS)
	(Rest)Größe	Größe aller Indexe $I : I \in \mathcal{I} \wedge I \notin C$
	Auswahl/Materialisierung	$\mathcal{I} \in C?$
	Indexe	Welche Indexe gehören zu $\mathcal{I}$
	Queries	Für welche Queries wurde $\mathcal{I}$ berechnet
$I$	Größe	Indexgröße $size(I)$
	Auswahl/Materialisiert	$I \in C$
	Konfigurationen	Welche Konfigurationen $\mathcal{I}$ enthalten $I$ ?
	Spalten	Über welche Spalten ist $I$ definiert
	Relation	Auf welcher Relation ist $I$ definiert?

Tabelle 4.5: Statistiken für Indexkonfigurationen und Indexe

Es ist zu beachten, dass die Lösung  $C$  schrittweise durch Hinzufügen von Indexkonfigurationen  $\mathcal{I}$  erstellt wird.  $\mathcal{I} \notin C$  bedeutet also, dass die lokal-optimale Indexkonfiguration  $\mathcal{I}$  noch nicht der Lösung  $C$  angehört, im weiteren Verlauf der Lösungserstellung aber noch zu  $C$  hinzugefügt werden kann.

Nach dieser kurzen Vorstellung LISA's und dem Aufzeigen der benötigten Statistiken, wird im nächsten Abschnitt gezeigt, wie LISA für einen gegebenen Workload  $W$  eine Indexempfehlung  $C$  erstellt.

#### 4.2.4 LISA - Indexauswahlverfahren

In diesem Abschnitt wird beschrieben, wie die Indexauswahl in LISA durchgeführt wird. Abbildung 4.12 skizziert grob den Ablauf der Indexauswahl.

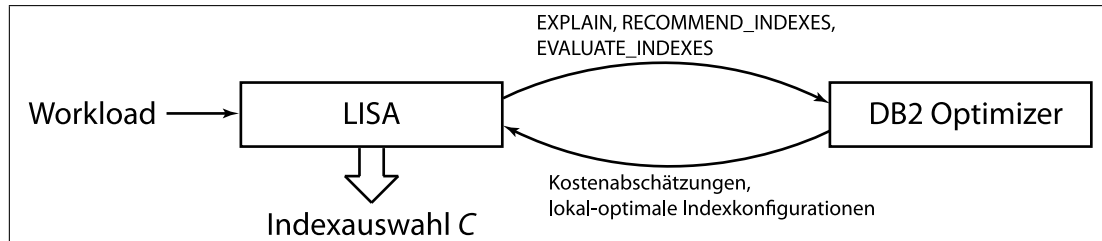


Abbildung 4.12: LISA - Übersicht

Einen gegebenen Workload reicht LISA an den DB2 Optimizer weiter. Dieser berechnet im RECOMMEND\_INDEXES-Modus die lokal-optimalen Indexkonfigurationen einzelner Anfragen  $Q$ , aus denen LISA die Lösung in Form der Indexauswahl  $C$  erstellt. Der Prozess der Lösungsfindung läuft wie folgt ab:

1. Beziehe den Workload  $W$ .
2.  $R = \emptyset, T = \emptyset$
3. Für jedes Query  $Q$  in  $W$ :
  - (a) Ermittle  $cost(Q)$  durch Ausführen von  $Q$  im EXPLAIN-Modus.
  - (b) Ermittle  $\mathcal{I}_Q$  und  $cost(Q, \mathcal{I}_Q)$  durch den RECOMMEND\_INDEXES-Modus.
  - (c) Berechne Profit  $profit(Q, \mathcal{I}_Q)$  und Größe  $size(\mathcal{I}_Q)$ .
  - (d) Prüfe ob eine Konfiguration  $\mathcal{I}_X \in R$  gibt, die  $\mathcal{I}_Q$  entspricht:
    - Falls ja: Addiere  $profit(Q, \mathcal{I}_Q)$  zu  $\mathcal{I}_X$ .
    - Falls nein:  $R = R \cup \mathcal{I}_Q$
  - (e) Für alle Indexe  $I_n \in \mathcal{I}_Q$ :
    - Ermittle  $size(I_n)$ .
    - Falls  $I_n \in T$ : Füge  $I_Q$  als Konfiguration  $I_n$ 's hinzu.
    - Falls  $I_n \notin T$ : Füge  $I_Q$  als Konfiguration  $I_n$ 's hinzu,  $T = T \cup I_n$ .
4. Solange eine Konfiguration  $\mathcal{I}_n \in R \wedge \mathcal{I}_n \notin C$  existiert für die gilt  $size(C) + size(\mathcal{I}_n) \leq S$  (mit  $S$  als Größenschranke):
  - (a) Füge die gemäss der Entscheidungsgröße  $\mathcal{E}(\mathcal{I})$  aktuell beste, in Bezug auf  $S$  gültige, Indexkonfiguration  $\mathcal{I}_m \notin C$  der Lösung  $C$  hinzu.
  - (b) Berechne  $size(\mathcal{I})$ , für alle  $\mathcal{I}$  die Überlappungen mit  $\mathcal{I}_m$  haben und nicht bereits zur Lösung  $C$  gehören, neu.
  - (c) Falls  $MAX\_EXPLAIN$  noch nicht erreicht: Berechne  $profit(\mathcal{I})$  mit Hilfe des EVALUATE\_INDEXES-Modus für alle  $\mathcal{I}$  neu, die Überlappungen mit  $\mathcal{I}_m$  haben und  $C$  noch nicht angehören.

5. Gib  $C$  als Lösung aus.

Nach dieser ersten Beschreibung des Ablaufes der Indexauswahl in LISA, sollen einzelne Punkte im Detail betrachtet werden. Zunächst sollen jedoch die neu eingeführten Symbole  $R$ ,  $T$  und  $\mathcal{E}(\mathcal{I})$  kurz erläutert werden.

In der Menge  $R$  umfasst alle lokal-optimalen Indexkonfigurationen  $\mathcal{I}$ , die durch den RECOMMEND\_INDEXES-Modus für die einzelnen Queries des Workloads erzeugt werden. Wie in Punkte 3 d) zu sehen ist, werden identische Indexkonfigurationen nicht doppelt in  $R$  aufgenommen. Die Menge  $T$  enthält alle Indexe, die in Schritt 3 b) erzeugt werden. Im Gegensatz zu  $R$  und  $T$  beschreibt  $\mathcal{E}(\mathcal{I})$  keine Menge, sondern die Größe, nach der die Konfigurationsauswahl erfolgt.  $\mathcal{E}(\mathcal{I})$  wurde eingeführt, da die Konfigurationsauswahl experimentell nach anderen Gesichtspunkten als nur dem Profit einer Konfiguration  $\mathcal{I}$  durchgeführt werden soll.

### Datenerhebung

Der Ablauf von LISA lässt sich in die zwei Teilbereiche *Datenerhebung* und *Konfigurationsauswahl* untergliedern. Bei der Datenerhebung wird für jedes Query  $Q$  des Workloads  $W$  die lokal-optimale Indexkonfiguration  $\mathcal{I}_Q$  ermittelt. Anschließend erfolgt die Erhebung der in Tabelle 4.5 (siehe Seite 54) gezeigten Daten. Die Größe  $size(\mathcal{I}_Q)$  kann über die Größe der in  $\mathcal{I}_Q$  enthaltenen Indexe berechnet werden. Die Berechnung von  $profit(Q, \mathcal{I}_Q)$  ist aus den vorigen Kapitel bekannt. Bevor die Konfiguration  $\mathcal{I}_Q$  der Menge  $R$  hinzugefügt werden kann, muss überprüft werden, ob es bereits eine identische Konfiguration  $\mathcal{I}_X \in R$  gibt. Zwei lokale Indexkonfigurationen  $\mathcal{I}_Q$  und  $\mathcal{I}_X$  gelten als identisch, wenn jeder Index aus  $\mathcal{I}_Q$  in  $\mathcal{I}_X$  und umgekehrt auch jeder Index aus  $\mathcal{I}_X$  in  $\mathcal{I}_Q$  enthalten ist.

Wie in Schritt 3 d) zu sehen ist, wird die bestehende Konfiguration  $\mathcal{I}_X$  um die Daten von  $\mathcal{I}_Q$  angereichert, falls die beiden Konfigurationen identisch sind. Das aktuell bearbeitete Query  $Q$  wird in einem solchen Fall  $\mathcal{I}_X$  als optimiertes Query angefügt. Der Gesamtprofit  $profit(\mathcal{I}_X)$  ergibt sich aus der Summe von Profiten  $profit(Q, \mathcal{I}_X)$  aller Queries  $Q$ , die durch  $\mathcal{I}_X$  optimiert werden.

Nachdem die Bearbeitung  $\mathcal{I}_Q$ 's erfolgt ist, werden die Indexe  $I_Q \in \mathcal{I}_Q$  betrachtet. Sind die Informationen über Größe, Indexattribute und die indexierte Tabelle eingeholt, so wird auch für Indexe geprüft, ob sie bereits für andere Queries des Workloads vorgeschlagen worden, also in der Menge  $T$  enthalten sind. Ist ein Index  $I_Q \in \mathcal{I}_Q$  bereits als  $I_T$  in  $T$  enthalten, so wird  $\mathcal{I}_Q$  als  $I_T$  enthaltende Indexkonfiguration angefügt. Die Indexkonfiguration  $\mathcal{I}_Q$  überlappt sich in einem solchen Fall mit mindestens einer anderen Indexkonfiguration. Abbildung 4.13 veranschaulicht die Schritte 3 a) bis 3 e) für ein Query  $Q_2$ .

Die einzelnen Daten über Indexkonfigurationen beziehungsweise Indexe können im Falle von DB2 direkt aus den ADVISE- und EXPLAIN-Tabellen gelesen werden, die durch den RECOMMEND\_INDEXES-Modus gefüllt werden. Der Vergleich von Indexkonfigurationen und Indexen muss möglichst effizient erfolgen. In Kapitel 5 ist zu sehen, wie dieser Vergleich bei der Implementierung LISA's im Rahmen dieser Arbeit durchgeführt wird.

Die durch LISA erhobenen Daten, werden in dieser Arbeit gemäß der Abbildung 4.14 in Relationen gespeichert. In der Relation *Indexe* sind weitgehend die Informationen

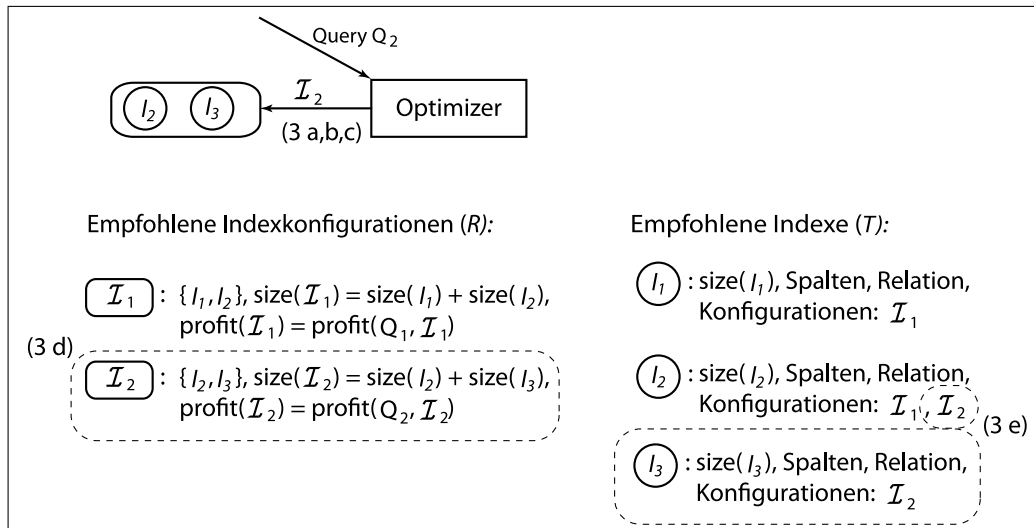


Abbildung 4.13: Hinzufügen neuer lokal-optimaler Indexkonfigurationen

aus Tabelle 4.5 gespeichert. Das Attribut *I\_NAMEDB2* ist notwendig, um Queries im EVALUATE\_INDEXES-Modus unter bestimmten Indexkonfigurationen zu bearbeiten (siehe Kapitel 5).

In der Relation *Konfigurationen* bedarf zunächst das Attribut *K\_PROFIT* einer Beschreibung. *K\_PROFIT* entspricht dem Gesamtprofit  $profit(\mathcal{I})$  einer Indexkonfiguration  $\mathcal{I}$ , der in den Ausführungen zu Schritt 3 d) definiert wurde. Dieser Profit ändert sich durch *Aging* (siehe Abschnitt 4.2.6), durch neue Queries  $Q$ , für die  $\mathcal{I}$  als optimal gilt (Schritt 3 d)) oder durch Profitneuberechnungen im EVALUATE\_INDEXES-Modus (4 c)). Die Zuordnung von Indexten zu einer Konfiguration erfolgt über die Relation *Konfigdefinition* durch Wertepaare von *I\_IDINDEX* und *K\_IDKONFIG*.

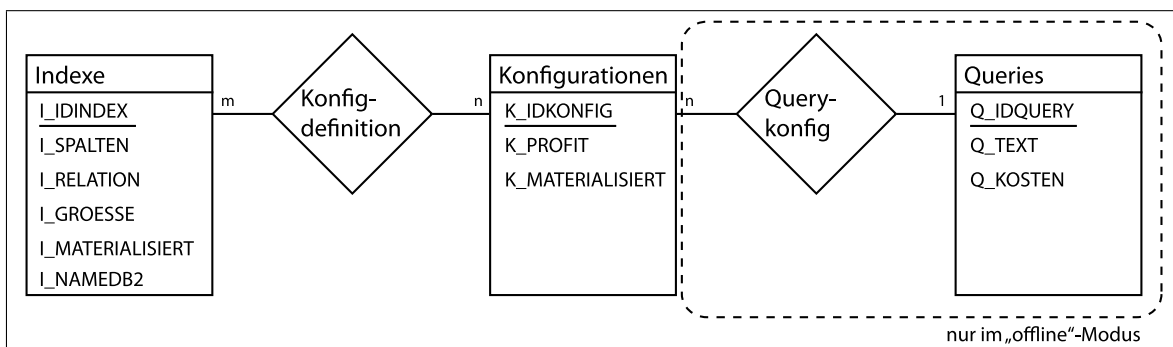


Abbildung 4.14: ER-Modell der Datenspeicherung in LISA

Die Relationen *Querykonfig* und *Queries* sind nur im offline-Modus von Bedeutung. Da Profitneuberechnungen im online-Modus nicht vorgesehen sind, ist es unnötig Informationen über Queries zu speichern. Sollen hingegen im offline-Modus Profitneuberechnungen im EVALUATE\_INDEXES-Modus stattfinden, so wird der Querytext *Q\_TEXT* benötigt. In *Q\_KOSTEN* werden ausserdem die in Schritt 3 a) ermittelten Kosten gespeichert, um eine Profitneuberechnung durchführen zu können. Die Relation *Querykonfig* stellt nun die Verbindung, durch Wertepaare *Q\_IDQUERY* und *K\_IDKONFIG*, zwischen Indexkonfigurationen und Queries her.

Die bei der Umsetzung des gezeigten Modells in dieser Arbeit verwendeten DB2-Datentypen sind:

- **Integer:** I\_IDINDEX, K\_IDKONFIG, Q\_IDQUERY
- **Small-Integer:** I\_MATERIALISIERT, K\_MATERIALISIERT
- **Big-Integer:** I\_GROESSE, K\_PROFIT\_MAX, Q\_KOSTEN
- **Varchar(128):** I\_RELATION, I\_NAMEDB2
- **Varchar(3500):** I\_SPALTEN, Q\_TEXT

Es wurde versucht die Menge der gespeicherten Daten möglichst gering zu halten, um durch die Pflege selbiger nur wenig Aufwand zu erzeugen. Die Datentypen *Integer*, *Small-Integer*, und *Big-Integer* haben eine Größe von 4, 2 und 8 Bytes. Die Größe der Attribute eines Datensatzes in der *Konfigurationen*-Relation entsprechen beispielsweise 14 Bytes. Die Größe der Relationen *Indexe* und *Queries* hängt stark von den jeweiligen Ausprägungen der Attribute *LSPALTEN* beziehungsweise *Q\_TEXT* ab. Ein Datenmodell mit geringerem Speicherplatzbedarf beim selben Informationsgehalt wurde im Laufe dieser Arbeit nicht gefunden.

### Auswahl lokal-optimaler Indexkonfigurationen

Nachdem alle Queries des Workloads durch den RECOMMEND\_INDEXES-Modus bearbeitet wurden, beginnt die eigentliche Erstellung der Lösungsmenge  $C$ . Wie bereits erwähnt, erfolgt die Erstellung  $C$ 's durch schrittweises Hinzufügen lokal-optimaler Indexkonfigurationen  $\mathcal{I}$  zur Lösungsmenge  $C$ . Die Auswahl von Indexkonfigurationen erfolgt nach der Entscheidungsgröße  $\mathcal{E}(\mathcal{I})$ , die in dieser Arbeit die folgenden drei verschiedenen Ausprägungen haben soll:

1.  $\mathcal{E}(\mathcal{I}) = \textit{profit}(\mathcal{I})$ : Für die Konfigurationsauswahl ist der Gesamtprofit einer Indexkonfiguration  $\mathcal{I}$  relevant. Bei gleichem Profit entscheiden Größe  $\textit{size}(\mathcal{I})$  und Anzahl der Überlappungen  $\mathcal{O}(\mathcal{I})$  in der genannten Reihenfolge über die Auswahl.
2.  $\mathcal{E}(\mathcal{I}) = \frac{\textit{profit}(\mathcal{I})}{\textit{size}(\mathcal{I})}$ : Die Konfigurationsauswahl erfolgt nach dem Verhältnis  $\frac{\textit{profit}(\mathcal{I})}{\textit{size}(\mathcal{I})}$ . Als zweites und drittes Auswahlkriterium gelten analog zu 1. gegebenenfalls Größe und die Anzahl der Überlappungen.
3.  $\mathcal{E}(\mathcal{I}) = \mathcal{O}(\mathcal{I})$ : Die Auswahl von Indexkonfigurationen erfolgt anhand der Überlappungen, die für eine Konfiguration  $\mathcal{I}$  ermittelt werden. Bei gleicher Anzahl von Überlappungen entscheiden Gesamtprofit und anschließend Größe der Indexkonfigurationen über die Auswahl.

In allen drei gezeigten Fällen wird jeweils die Konfiguration mit dem größten Wert  $\mathcal{E}(\mathcal{I})$  gewählt. Die Berechnung der Überlappungen  $\mathcal{O}(\mathcal{I})$  einer Indexkonfiguration  $\mathcal{I}$  erfolgt über die in  $\mathcal{I}$  enthaltenen Indexe  $I$ . Für jeden Index  $I \in \mathcal{I} \wedge I \notin C$  wird ermittelt, wie vielen Indexkonfigurationen er angehört. Die jeweils erhaltene Anzahl wird um eins vermindert, da sich  $\mathcal{I}$  nicht selbst überlappt, und über alle Indexe aufaddiert. Abbildung

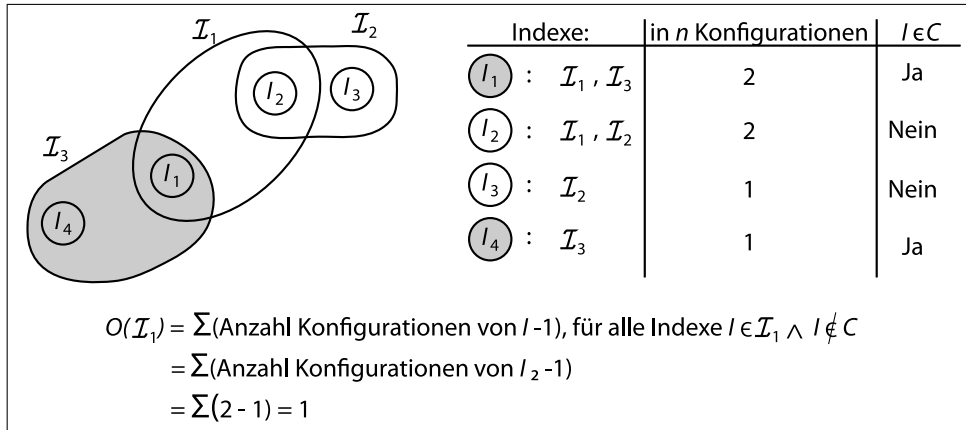


Abbildung 4.15: Berechnung  $\mathcal{O}(\mathcal{I})$

4.15 beschreibt die Berechnung der Überlappungen der Konfiguration  $\mathcal{I}_1$ . Die grau hinterlegte Konfiguration  $\mathcal{I}_3$  ist bereits in  $C$  enthalten, so dass diese Überlappung von  $\mathcal{I}_1$  für die Auswahl nicht mehr berücksichtigt wird.

Die drei genannten Möglichkeiten  $\mathcal{E}(\mathcal{I})$  zu berechnen verfolgen verschiedene Ziele. Der erste Fall verspricht die beste Lösung  $C$ , erfordert aber eine ständige Profitneuberechnungen der Indexkonfigurationen im EVALUATE.INDEXES-Modus. Im zweiten Fall:  $\mathcal{E}(\mathcal{I}) = \frac{\text{profit}(\mathcal{I})}{\text{size}(\mathcal{I})}$  wird versucht komplette Indexkonfigurationen auszuwählen. Durch Überlappungen teilausgewählte Indexkonfigurationen haben auf Grund ihrer verringerten Größe einen höheren Wert  $\mathcal{E}(\mathcal{I})$ , was eine Auswahl wahrscheinlicher macht. Auf Grund der beschriebenen Probleme der Indexabhängigkeit und Einzelprofitberechnung von Indexen ist es erstrebenswert komplette Konfigurationen auszuwählen, wenn keine Profitneuberechnung von Indexkonfigurationen stattfindet. Die dritte Variante der Berechnung  $\mathcal{E}(\mathcal{I})$ 's versucht Indexe auszuwählen, die an der Optimierung vieler Queries beteiligt sind. Eine nähere Beschreibung dieser Variante erfolgt in Abschnitt 4.2.5.

Nachdem die Auswahl einer Indexkonfiguration in Schritt 4 a) erfolgt ist, kommt es in LISA zur Neuberechnung von Größe und Profit überlappender Indexkonfigurationen (4 b) und 4 c)). Eine erste Beschreibung des Ablaufes der Neuberechnungen wurde bereits in Abbildung 4.11 auf Seite 54 gegeben. Nach der Auswahl einer Konfiguration  $\mathcal{I}_n$  werden alle Indexe  $I \in \mathcal{I}_n$  durchlaufen und für jeden Index  $I$  wird geprüft, in welchen Konfigurationen er enthalten ist. Dabei sind nur Konfigurationen  $\mathcal{I} \notin C$  von Interesse. Die Größe der überlappenden Konfigurationen wird um die Größe des jeweiligen Indexes  $I$  verringert (4 b)). Falls die Anzahl  $MAX\_EXPLAIN$  der erlaubten RE-EXPLAINS noch nicht überschritten ist, werden in Schritt 4 c) die Profite der  $\mathcal{I}_n$  überlappenden Indexkonfigurationen  $\mathcal{I}$  durch den EVALUATE.INDEXES-Modus neu berechnet. Die Berechnung erfolgt mit allen Indexen  $I \notin C$  der überlappenden Konfigurationen. Die beiden Schritte der Neuberechnung sind in Abbildung 4.16 dargestellt. Nach der Auswahl einer Konfiguration  $\mathcal{I}_n$  sind folgenden Schritte durchzuführen:

1. Für alle Indexe  $I \in \mathcal{I}_n \wedge I \notin C$ :
  - (a) Setze Auswahl-/Materialisierungsinfo für  $I$  auf *wahr*.
  - (b)  $C = C \cup I$

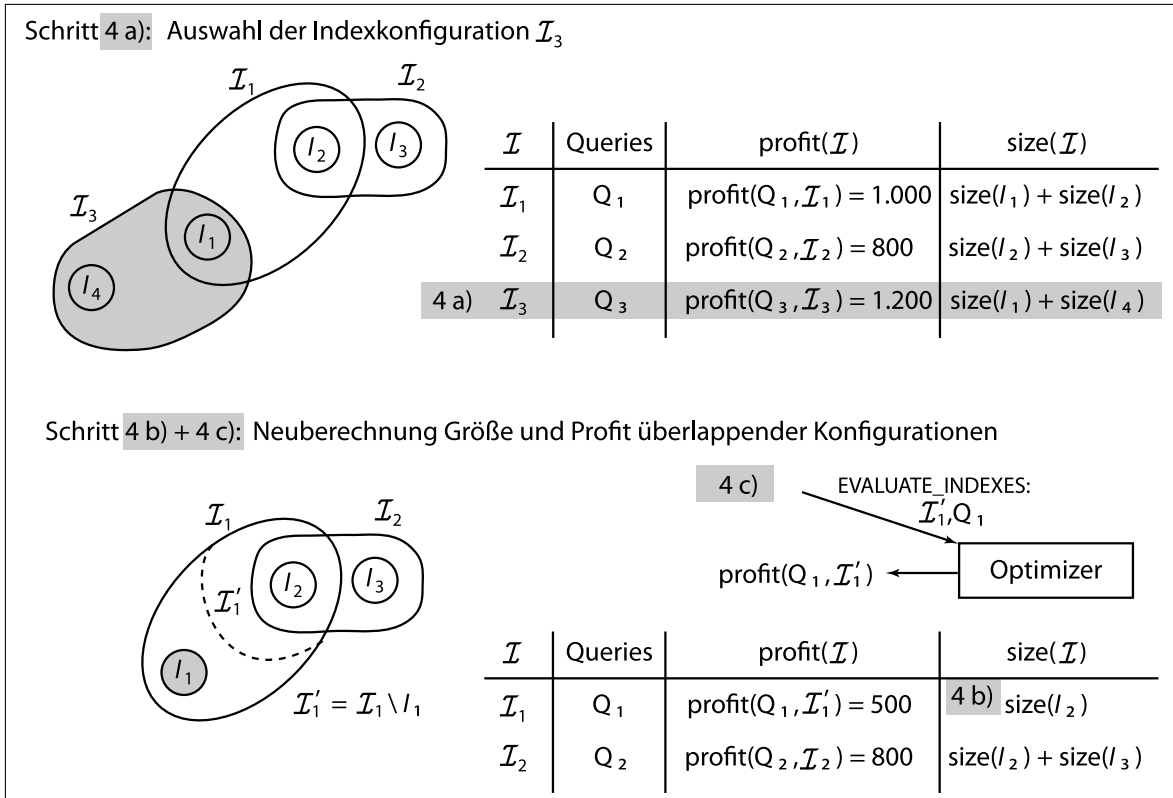


Abbildung 4.16: Neuberechnungen nach Auswahl einer Indexkonfiguration

(c) Für alle Indexkonfigurationen  $\mathcal{I} \supseteq I \wedge \mathcal{I} \notin C$ :

- i.  $size(\mathcal{I}) = size(\mathcal{I}) - size(I)$ .
- ii. Falls MAX\_EXPLAIN noch nicht erreicht:  $profit(\mathcal{I}) = profit(Q, \mathcal{I} \setminus \{I \in \mathcal{I} \wedge I \notin C\})$ .

2. Setze Auswahl-/Materialisierungsinfo für  $\mathcal{I}_n$  auf *wahr*.

## 4.2.5 LISA - Vor- und Nachteile

Die Vorteile LISA's leiten sich aus den in Abschnitt 4.2.1 formulierten Zielen ab. Diese bestanden aus der Berücksichtigung von Abhängigkeiten zwischen Indexen und dem Vermeiden einer IndexfehlAuswahl auf Grund fehlerhaft berechneter Einzelprofite. Durch die Verwendung von lokal-optimalen Indexkonfigurationen in einem Auswahlverfahren ergeben sich jedoch auch Probleme, die im Folgenden beschrieben werden.

### Erhöhter Datenbedarf

Durch die Verwendung von Indexkonfigurationen entsteht ein erhöhter Datenbedarf im Vergleich zu einem Verfahren wie dem im DB2 Design Advisor. Zusätzlich zu Indexinformationen müssen in auch LISA Daten zu Indexkonfigurationen verwaltet werden was mit einem höheren Aufwand sowohl bei der Lösungserstellung, als auch bei der Datenspeicherung einhergeht.

## Präfixindexe und Indexmerging

Da LISA auf Basis von Indexkonfigurationen arbeiten soll, wird auf die Suche nach *Präfixindexten*, wie sie noch im DB2 Design Advisor stattfand, verzichtet. Auch ein *Index-Merging* findet nicht statt. Zwar könnten durch Berücksichtigung der beiden genannten Punkte bessere Lösungen erstellt werden, zugunsten der Einfachheit LISA's wird in dieser Arbeit jedoch darauf verzichtet.

## Größenschränke

Bei sehr klein gewählten Größenschränken können nur wenige Indexkonfigurationen  $\mathcal{I}$  ausgewählt werden. Durch die höhere Flexibilität bei der Betrachtung nur einzelner Indexe kann in einem solchen Fall vermutlich eine bessere Lösung erzielt werden, wenn nur Indexe, anstatt ganzer Konfigurationen betrachtet werden.

## Mehrfach verwendete Indexe

Mehrfach verwendete Indexe sind Indexe, die an der Optimierung mehrerer Queries eines Workloads beteiligt sind. Ein solcher Index ist vermutlich mit einem hohen Gesamtprofit verbunden. Im DB2 Design Advisor wird jedem Index der gesamte Profit aller lokal-optimalen Indexkonfigurationen zugewiesen denen er angehört. Die Aufnahme eines solchen Indexes in die Lösungsmenge ist also wahrscheinlich. In LISA wird hingegen nur der Profit von Indexkonfigurationen betrachtet. Indexe die vielen Konfigurationen angehören werden möglicherweise unterbewertet. Das beschriebene Problem ist in Abbildung 4.17 illustriert.

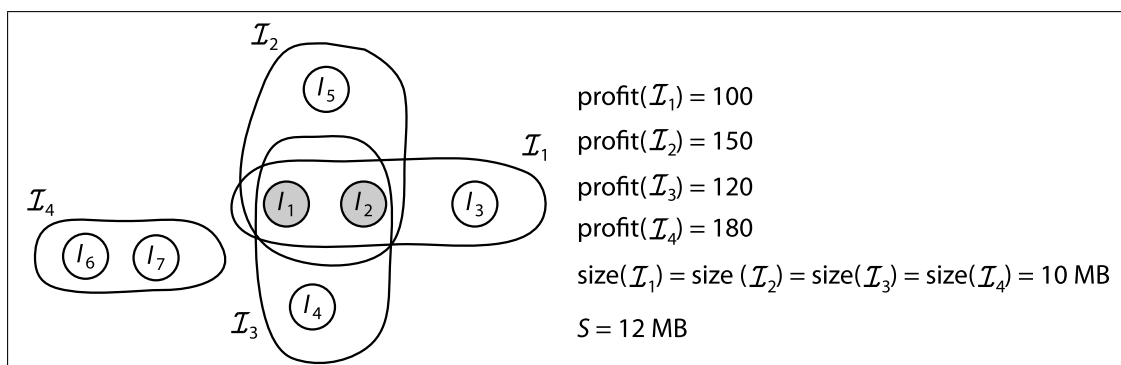


Abbildung 4.17: LISA - Nachteile

Durch die Größenschränke  $S$  von 12 Megabyte kann nur eine Indexkonfiguration durch LISA gewählt werden. Erfolgt die Auswahl nach dem Profit  $\text{profit}(\mathcal{I})$ , so wird die Konfiguration  $\mathcal{I}_4$  mit den Indexten  $I_6$  und  $I_7$  gewählt. In Bezug auf den gesamten Workload ist die Auswahl der Indexe  $I_1$  und  $I_2$  vermutlich die bessere Alternative, da sie mehrere Queries optimieren. Um in einer solchen Situation bessere Lösungen zu erstellen, soll in LISA versuchsweise die Auswahl nach Anzahl der Überlappungen erfolgen.

## Abhängigkeiten zwischen Indexkonfigurationen

Die Abhängigkeiten durch Überlappungen werden durch die Auswahlkriterien LISA's nur ungenügend behandelt. Als Beispiel kann erneut die Abbildung 4.17 dienen. Für den Fall, dass die Indexe  $I_3, I_4$  und  $I_5$  jeweils eine Größe von 0,5 Megabyte haben, könnten die Indexkonfigurationen  $\mathcal{I}_1, \mathcal{I}_2$  und  $\mathcal{I}_3$  ausgewählt werden. Da LISA bei der Konfigurationsauswahl in Schritt 4 a) allerdings nur einzelne Konfigurationen betrachtet, würde im Fall von  $\mathcal{E}(\mathcal{I}) = profit(\mathcal{I})$  und  $\mathcal{E}(\mathcal{I}) = \frac{profit(\mathcal{I})}{size(\mathcal{I})}$  die nur Konfiguration  $\mathcal{I}_4$  gewählt werden.

Eine Erweiterung LISA's könnte darin bestehen sich überlappende Indexkonfigurationen als eine Einzelne zu betrachten. Bezogen auf Abbildung 4.17 würden die Konfigurationen  $\mathcal{I}_1, \mathcal{I}_2$  und  $\mathcal{I}_3$  dann in einer virtuellen Konfiguration  $\mathcal{I}_{virt}$  mit  $profit(\mathcal{I}_{virt}) = 370$  und  $size(\mathcal{I}_{virt}) = 11.5MB$  zusammengefasst. Ein solches Zusammenfassen von Konfigurationen ist allerdings mit weiterem Aufwand verbunden und kann im Falle vieler Überlappungen zu so großen  $\mathcal{I}_{virt}$  führen, die eine vorgegebene Größenschranke von vornherein überschreiten. Das bilden aller möglichen virtuellen Konfigurationen führt wiederum zur kombinatorischen Explosion in Bezug auf die Anzahl der möglichen Konfigurationen.

## Anzahl Indexkonfigurationen

Die Anzahl an Indexkonfigurationen wird bei vielen ähnlichen Queries höher sein, als die Anzahl der Indexe. Der Platzbedarf der Datenspeicherung steigt damit entsprechend. Auch die Anzahl der sich Überlappenden Indexkonfigurationen steigt im Fall vieler ähnlicher Queries an. Die Neuberechnung von Konfigurationsgrößen und -profiten kann in einem solchen Fall zu sehr hohem Aufwand führen und zum Engpass werden. Besonders im Hinblick auf den möglichen Einsatz LISA's als online-Verfahren ergeben sich hierdurch Probleme.

### 4.2.6 LISA - Online

Zum Abschluss der Betrachtungen zu LISA soll noch kurz untersucht werden, inwieweit sich LISA als online-Verfahren eignet.

Prinzipiell könnte LISA ähnlich dem QUIET-System (siehe Abschnitt 3.3) umgesetzt werden. Konzepte wie das *Aging* und das Ändern der aktuell materialisierten Indexe  $mat(C)$  erst ab einer bestimmten Minimaldifferenz (in Bezug auf den Profit) sind auch auf lokale Indexkonfigurationen anwendbar. Das Auswahlkriterium  $\mathcal{E}(\mathcal{I})$  müsste sich, wie in QUIET, allein auf den Profit einer Indexkonfiguration beziehen. Auf eine Profitneuberechnung im EVALUATE\_INDEXES-Modus sollte im online-Verfahren auf Grund des zusätzlichen Aufwandes verzichtet werden. Es ist deshalb auch unnötig, Informationen über Queries zu speichern (siehe Abbildung 4.14 auf Seite 57).

Im online-Betrieb kommt es, wie bereits in 4.2.5 erwähnt, vermutlich zu vielen sich ähnelnder Konfigurationen, aber auch zur mehrfachen Ausführung gleicher Queries. Es entstehen hier also mitunter sehr viele Konfigurationen die verwaltet werden müssen. Der Abgleich, ob eine Konfiguration bereits existiert, wird entsprechend teurer. In Abschnitt 4.2.5 wurde ausserdem darauf hingewiesen, dass Indexe, die in mehreren Konfigurationen enthalten sind, unterbewertet werden. Gerade im Fall vieler ähnlicher Konfigurationen besteht die Gefahr einer suboptimalen Indexauswahl.

Werden effiziente Lösungen für die oben genannten Probleme gefunden, so ist LISA durchaus als online-Verfahren tauglich. Die Implementation LISA's im Rahmen dieser Arbeit beschränkt sich jedoch auf die Indexauswahl im offline-Modus.

### 4.3 Zusammenfassung

Im ersten Teil von Kapitel 4 wurde untersucht, ob ein Rucksackverfahren auf Basis einzelner Indexe eine optimale Lösung für das ISP bieten kann. Als konkretes Verfahren dazu wurde der DB2 Design Advisor untersucht. In den Vorbetrachtungen dieser Untersuchungen wurden Abhängigkeiten zwischen Indexen beschrieben und an Beispielen dargestellt. Bei der näheren Analyse des DB2 Design Advisors konnten keine Hinweise darauf gefunden werden, dass der DB2 Design Advisor Indexabhängigkeiten bei der Indexauswahl berücksichtigt.

Ein Zwischenschritt auf dem Weg der Lösungserstellung im DB2 Design Advisor, ist die Bestimmung einer Menge von Indexen, die eine bestimmte Anfrage mit den geringstmöglichen Kosten bearbeiten. Diese Indexmenge wurde als lokal-optimale Indexkonfiguration (kurz Indexkonfiguration oder nur Konfiguration) bezeichnet und wird durch den RECOMMEND\_INDEXES-Modus erstellt. Im weiteren Ablauf des DB2 Design Advisors werden aus diesen Indexkonfigurationen einzelne Indexe ausgewählt. An einem Beispiel wurde gezeigt, dass diese Zerlegung der Indexkonfigurationen problematisch ist, da der Profit einer Konfiguration keine Schlüsse auf den Profit einzelner Indexe zulässt. Da der DB2 Optimizer keine Informationen über den Profit einzelner Indexe ausgibt, wird im Design Advisor jedem Index der Profit der gesamten Indexkonfiguration zugewiesen. Im gezeigten Beispiel kam es dadurch zu Indexfehlauswahlen.

Im zweiten Teil dieses Kapitels wurde mit LISA ein Auswahlverfahren vorgestellt, welches bei der Indexauswahl auf den lokal-optimalen Indexkonfigurationen basiert. Durch diese Vorgehensweise wird den Indexabhängigkeiten in Indexkonfigurationen Rechnung getragen und die Indexfehlauswahl auf Grund fehlerhaft geschätzter Indexeinzelprofite vermieden. Durch die Verwendung von Indexkonfigurationen entsteht aber eine neue Art der Abhängigkeit in Form von Überlappungen. Überlappungen entstehen, wenn ein Index in mehreren Konfigurationen vorkommt. Die Auswahl einer Konfiguration hat dadurch Auswirkungen auf alle anderen Konfigurationen, die sich Indexe mit ihr teilen. In LISA wird in Form von Neuberechnungen der Konfigurationsgröße und des Konfigurationsprofiten auf Überlappungen reagiert.

Im Vergleich zum DB2 Design Advisor werden in LISA mehr Daten bei der Lösungserstellung benötigt. Durch die Neuberechnungen von Konfigurationsgröße und -profit entsteht ausserdem ein gewissen Mehraufwand. Ob sich dieser Mehraufwand lohnt, soll im Folgenden Kapitel bei der Evaluierung untersucht werden.



# Kapitel 5

## Implementierung und Evaluierung

### 5.1 LISA - Implementierung

Im Rahmen dieser Arbeit wurde das Konzept von LISA in einer Java<sup>1</sup>-Application umgesetzt, die auf einer DB2-Datenbank aufsetzt. Der wesentliche Aufbau des Programmes und dessen Ablauf werden im Folgenden kurz beschrieben. Das entstandene Programm wird dabei *JLISA* genannt.

#### 5.1.1 JLISA - Aufbau

JLISA benutzt das in Abbildung 4.14 (siehe Seite 57) gezeigte Datenmodell. Die für LISA relevanten Objekte: *Indexe*, *Indexkonfigurationen* und *Queries* wurden in JLISA als Klassen umgesetzt. Die Relationen *Konfigdefinition* und *Querykonfig* werden als *LinkedList* (deutsch: verkettete Liste) in der Klasse *IndexConfig* gespeichert.

Die drei in JLISA entstandenen Klassen *Index*, *IndexConfig* und *Query* enthalten alle Attribute der Tabellen aus Abbildung 4.14. In der Klasse *Index* werden zusätzlich die Konfigurationen, denen ein Index angehört, in einer *LinkedList* gespeichert. Da es im Programmablauf von JLISA erforderlich wird *Index* und *IndexConfig* zu sortieren, erben diese beiden Klassen von *Comparable* und implementieren die Methode *compareTo()*. Weitere Methoden der Klassen werden, falls benötigt, im Abschnitt zum Programmablauf beschrieben.

Während der Laufzeit JLISAs wird in der Regel nicht auf die Datenbanktabellen aus Abbildung 4.14 zugegriffen. Alle Objekte befinden sich zur Laufzeit im Arbeitsspeicher und werden erst am Ende der Bearbeitung in die Datenbank geschrieben.

#### 5.1.2 JLISA - Programmablauf

Der Ablauf JLISAs entspricht im Wesentlichen dem in Abschnitt 4.2.4 vorgestellten Algorithmus. Die Einstellung der Parameter *MAX\_EXPLAIN* und der Größenschranke *S* erfolgen über eine Konfigurationsdatei. Auch die Berechnung von  $\mathcal{E}(\mathcal{I})$  wird in dieser Datei eingestellt. Der zu bearbeitende Workload wird JLISA in Form einer Textdatei

---

<sup>1</sup><http://www.sun.com/java/>

übergeben. Beim Start von Lisa werden zunächst die Konfigurationsdatei und gegebenenfalls die Workloaddatei eingelesen. Indexe, die durch früheres Ausführen von JLISA entstanden, werden beim Programmstart gelöscht. Anschließend erfolgt die Ausführung des Prozesses aus Abschnitt 4.2.4, der im Folgenden schrittweise beschrieben wird.

### Beziehe Workload - $R = \emptyset, T = \emptyset$

Der Workload wird, wie bereits geschrieben, aus einer Datei gelesen. Die Datei kann beliebig viele Queries enthalten, die jeweils durch „-“ getrennt werden müssen. Die Mengen  $R$  und  $T$  werden in JLISA durch *TreeMaps* repräsentiert. *TreeMaps* arbeiten in etwa nach dem Prinzip eines B-Baum-Indexes. Das Suchen, Hinzufügen und Löschen geschieht mit einem Aufwand von jeweils  $\log(n)$ .

### Datenerhebung

Schritt 3 in LISA ist die Datenerhebung. Für jedes Query  $Q$  des gegebenen Workloads  $W$  wurden hierbei verschiedene Daten ermittelt. Auch JLISA arbeitet den Workload nach diesem Muster ab.

$Q$  wird zunächst im EXPLAIN-Modus an den Optimizer weitergeleitet. Der Optimizer stellt die Kosten  $cost(Q)$  in der Tabelle *EXPLAIN\_OPERATOR* zur Verfügung. Diese Kosten werden von JLISA gelesen und in einem neuen Query-Objekt gespeichert. Anschließend wird  $Q$  im RECOMMEND\_INDEXES-Modus ausgeführt. Die empfohlene Indexkonfiguration  $\mathcal{I}_Q$  wird vom Optimizer in der Tabelle *ADVISE\_INDEX* gespeichert. Die Kosten  $cost(Q, \mathcal{I}_Q)$  werden erneut in der *EXPLAIN\_OPERATOR*-Tabelle gespeichert und von JLISA gelesen. Die empfohlenen, nicht bereits in der Datenbank vorhandenen, Indexe werden von JLISA aus der *ADVISE\_INDEX*-Tabelle gelesen. Anhand der Indexspalten wird nun überprüft, ob ein Index bereits in der *TreeMap*  $T$  enthalten ist. Ist ein Index aus *ADVISE\_INDEX* bereits in  $T$  enthalten, so wird das entsprechende Index-Objekt beziehungsweise eine Referenz darauf aus der *TreeMap*  $T$  gelesen und in einer temporären Liste gespeichert. Für neue Indexe wird ein Index-Objekt erzeugt, welches ebenfalls der temporären Liste zugefügt wird. Neue Indexe werden ausserdem der *TreeMap*  $T$  hinzugefügt. Suchschlüssel in  $T$  sind immer die Indexspalten.

*ADVISE\_INDEX* enthält bereits die meisten der von LISA beziehungsweise JLISA benötigten Informationen - einzig die Indexgrößen müssen berechnet werden. Da Indexe in DB2  $B^+$ -Bäume sind, erfolgt die Berechnung über die Anzahl von Baumknoten und der Seitengröße des Dateisystems.

Die temporär erstellte Liste wird nun nach Indexen sortiert. Der Aufwand hierfür ist  $n \cdot \log(n)$ . Die Sortierung erfolgt nach *LIDINDEX*. Aus den sortierten Indexen wird nun ein Schlüssel erstellt, der den kommagetrennten ID's entspricht und für jede Indexkonfiguration eindeutig ist. Mit diesem Schlüssel wird in der *TreeMap*  $R$  nach einem *IndexConfig*-Objekt gesucht. Wird kein Eintrag in  $R$  gefunden, wird ein neues *IndexConfig*-Objekt erstellt und mit dem ermittelten Schlüssel zu  $R$  hinzugefügt. Die Indexe des neuen *IndexConfig*-Objektes sind in der temporären Liste enthalten und werden der neuen *IndexConfig* bei der Erzeugung übergeben. Anhand der übergebenen Indexe wird auch die Größe der neuen Indexkonfiguration berechnet und im entsprechenden *IndexConfig*-Objekt gespeichert.

Hat JLISA das IndexConfig-Objekt  $\mathcal{I}_Q$  für das aktuelle Query  $Q$  erstellt, oder aus  $R$  gelesen, so wird das zuvor für  $Q$  erstellte Query-Objekt der IndexConfig  $\mathcal{I}_Q$  als Query hinzugefügt und der Profit  $profit(Q, \mathcal{I}_Q)$  aufaddiert.

Ist die Bearbeitung des IndexConfig-Objektes abgeschlossen, so wird jedem Index aus der temporär gespeicherten Indexliste die IndexConfig  $\mathcal{I}_Q$  als Indexkonfiguration hinzugefügt. Die Bearbeitung eines Queries  $Q$  ist damit abgeschlossen.

Sind alle Queries des gegebenen Workloads auf die oben beschriebene Weise bearbeitet worden, sind folgende Daten in JLISA gespeichert:

- **Query-Objekte:** Für jedes Query des Workloads ist ein Query-Objekt erzeugt worden. In diesem Objekt sind Q\_IDQUERY, Q\_TEXT und Q\_KOSTEN (aus dem EXPLAIN-Modus) gespeichert.
- **Index-Objekte:** Für jeden Index, der für den Workload empfohlen wurde, existiert ein Index-Objekt. In diesem Objekt sind I\_GROESSE, I\_NAMEDB2, I\_SPALTEN, I\_RELATION, I\_IDINDEX, I\_MATERIALISIERT (=falsch) gespeichert. Ausserdem enthält jedes Index-Objekt eine verkettete Liste *index\_configs*, die alle Indexkonfigurationen (IndexConfig-Objekte) enthält, denen der Index angehört.
- **IndexConfig-Objekte:** Für jede Indexkonfiguration  $\mathcal{I}$  existiert ein IndexConfig-Objekt. In diesem IndexConfig-Objekt sind gespeichert: K\_IDKONFIG, K\_PROFIT (über alle Queries) und K\_MATERIALISIERT (=falsch). In IndexConfig-Objekten gibt es des Weiteren zwei verkettete Listen *config\_indexes* und *config\_queries*, die die Index- beziehungsweise Query-Objekte enthalten, die mit der Indexkonfiguration verbunden sind und mit K\_GROESSE eine Variable, die die aktuelle Konfigurationsgröße speichert.
- **Menge  $R$ :** Enthält Verweise auf alle IndexConfig-Objekte.
- **Menge  $T$ :** Enthält Verweise auf alle Index-Objekte.

Ist die Datenerhebung abgeschlossen, werden die erhobenen Daten in die entsprechenden Datenbanktabellen geschrieben. Die Relationen *Konfigdefinition* und *Querykonfig* werden dabei durch die entsprechenden Listen in den IndexConfig-Objekten gefüllt.

Wird JLISA erneut mit dem gleichen Workload ausgeführt, können die Daten aus den Tabellen gelesen werden. Die EXPLAIN- und RECOMMEND\_INDEXES-Anfragen können in dieser Situation entfallen.

### Auswahl lokal-optimaler Indexkonfigurationen

Sind die erforderlichen Daten durch JLISA erhoben, oder aus den Datenbanktabellen gelesen, beginnt die Lösungserstellung durch Auswahl lokal-optimaler Indexkonfigurationen.

Die TreeMap  $R$  wird dazu in die LinkedList *auswahlliste* umgewandelt. In *auswahlliste* sind also alle Indexkonfigurationen für den Workload enthalten. Die Auswahl der Konfigurationen erfolgt nun wie bereits in Abschnitt 4.2.4 beschrieben.

Aus *auswahlliste* wird das IndexConfig-Objekt  $\mathcal{I}$  mit dem größten  $\mathcal{E}(\mathcal{I})$ -Wert gewählt, dessen Größe kleiner-gleich MAX\_SIZE ist. K\_MATERIALISIERT wird für dieses

IndexConfig-Objekt auf *wahr* gesetzt. Anschließend werden für jedes Index-Objekt  $I$ , welches in *config\_indexes* des gewählten IndexConfig-Objektes vorhanden ist, die folgenden Schritte durchgeführt:

1. Setze  $I.I\_MATERIALISIERT$  auf *wahr*.
2. Für jedes IndexConfig-Objekt  $\mathcal{I}_I$  in  $I.index\_configs$  mit  $\mathcal{I}_I.K\_MATERIALISIERT = falsch$ :
  - (a) Ermittle neues  $K\_PROFIT$  für  $\mathcal{I}_I$  durch Ausführung aller Queries in  $\mathcal{I}_I.config\_queries$  im EVALUATE\_INDEXES-MODUS. Benutze dabei nur die Indexe in  $\mathcal{I}_I.config\_indexes$ , die  $I.MATERIALISIERT = falsch$  haben.
  - (b)  $\mathcal{I}_I.K\_GROESSE = \mathcal{I}_I.K\_GROESSE - I.I\_GROESSE$
3.  $MAX\_SIZE = MAX\_SIZE - \mathcal{I}.K\_GROESSE$ .

Sind diese Schritte für alle Index-Objekte  $I$  des aktuell gewählten IndexConfig-Objektes  $\mathcal{I}$  durchgeführt, so wird  $\mathcal{I}$  aus *auswahlliste* entfernt und das nächste IndexConfig-Objekt  $\mathcal{I}$  mit  $\mathcal{I}.K\_GROESSE \leq MAX\_SIZE$  und dem größten  $\mathcal{E}(\mathcal{I})$ -Wert gewählt. JLISA bricht die Indexauswahl ab, wenn *auswahlliste* keine Elemente mehr enthält, oder kein IndexConfig-Objekt mehr existiert, dessen Größe kleiner als  $MAX\_SIZE$  ist.

Um Schritt 2 a) des oben geschriebenen Ablaufes durchzuführen, wird in der ADVISE\_INDEX-Tabelle das Attribute *USE\_INDEX* (deutsch: benutze Index) auf *JA* beziehungsweise *NEIN* gesetzt. Mit jeder Ausführung des RECOMMEND\_INDEXES-Modus wird ausserdem  $MAX\_EXPLAIN$  um eins verringert. Profitneuberechnungen finden nur statt, wenn  $MAX\_EXPLAIN$  größer als 0 ist.

## 5.2 Evaluierung

Nach der Implementierung von JLISA wurden einige Testberechnungen auf Basis des TPC-H Benchmarks, mit der Größe von einem Gigabyte, durchgeführt und die Ergebnisse mit denen des DB2 Design Advisors verglichen. Die Indexe auf den Primärschlüsseln der Tabellen des TPC-H Benchmarks blieben während aller Testläufe erhalten. Wird im Folgenden von Testläufen ohne Indexe gesprochen, bezieht sich dies nur auf virtuelle Indexe. Die Primärindexe sind auch in diesen Testläufen vorhanden.

Bei den ersten Berechnungen wurde festgestellt, dass Query 17 des TPC-H Workloads ohne Indexe geschätzte Kosten von 55 Millionen Timeron erzeugt. Der Rest des Workloads hatte in der Summe gerade noch 4 Millionen Timeron geschätzte Kosten. Um die Ergebnisse durch diesen Ausreisser nicht zu verfälschen, wurde Query 17 bei den hier gezeigten Ergebnissen und Berechnungen ausgeschlossen. Auch Query 15 wurde aus dem Workload entfernt, da die Kostenabschätzungen problematisch war. In Query 15 wird eine View erzeugt und gelöscht.

Die Bearbeitungszeiten in Sekunden wurden mit dem db2batch-Tool ermittelt. Es wird in dieser Arbeit jeweils das arithmetische Mittel aus 5 Wiederholungen verwendet.

### 5.2.1 Vorbetrachtungen

Zunächst sollen die Ergebnisse des unoptimierten Workloads (Tabelle 5.1) betrachtet werden und die für den Workload ermittelten Indexempfehlungen und Konfigurationen gezeigt werden.

TPC-H Query	Kosten Timeron	Bearbeitungszeit (Sekunden)
1	210.575	23,04
2	28.313	10,15
3	261.551	17,26
4	132.036	43,12
5	235.072	16,21
6	210.547	20,15
7	262.026	18,15
8	269.092	17,59
9	308.105	29,87
10	261.342	18,15
11	65.370	2,96
12	246.113	25,05
13	46.179	3,15
14	217.587	22,15
16	137.621	1,06
18	212.495	19,88
19	217.784	1,81
20	218.069	486,55
21	377.737	62,52
22	58.387	4,15
Summe:	3.917.614	838,82

Tabelle 5.1: Antwortzeiten unoptimierter Workload

Bei der Ausführung aller Queries im RECOMMEND\_INDEXES-Modus wurden die in Tabelle 5.2 gezeigten Indexkonfigurationen erstellt. Informationen über die einzelnen Indexe sind im Anhang zu finden.

Um alle Konfigurationen beziehungsweise Indexe materialisieren zu können, ist ein Speicherplatz von rund 2,3 Gigabyte erforderlich. Damit ist der Speicherplatz für die empfohlenen Indexe doppelt so groß, wie die Relationen des TPC-H Benchmarks (dieser Arbeit) selbst. Anhand der Indexe lässt sich erkennen, dass es einige Überlappungen zwischen den Indexkonfigurationen gibt.

Es fällt auf, dass es Indexkonfigurationen mit negativen Profiten gibt. Diese Konfigurationen werden durch LISA niemals gewählt. In der Praxis kann durch diese Konfigurationen dennoch eine Verbesserung der Antwortzeit erreicht werden. Für LISA ist dies zum Zeitpunkt der Optimierung aber nicht zu erkennen, so dass hier die Möglichkeit einer Indexfehlauswahl besteht.

Konfig-ID	Indexe	Größe (KB)	Queries	Profit (Timeron)
1	-	0	1, 13	0
2	1,2,3,4,5,6,7,8	68.812	2	27.293
3	9,10	48.301	3	11.614
4	11,12	148.636	4	-31.471
5	13,14,15,16,17,18	288.250	5	100.221
6	19,20	84.495	6	169.877
7	13,16,21	111.489	7	194.261
8	5,14,16,22,23,24,25	334.876	8	217.855
9	16,26	26.693	9	5.886
10	18,27	105.790	10	132.984
11	16,28,29	28.697	11	51.258
12	30	266.898	12	181.791
13	31	249.927	14	204.386
14	32,33	17.342	16	7.180
15	34,35	60.087	18	134
16	36,37	60.956	19	209.849
17	28,38,39,40,41	175.047	20	217.994
18	28,42,43,44,45	233.344	21	-27.337
19	46,47	10.699	22	43.208

Tabelle 5.2: Indexkonfigurationen Workload

### Weitere Anmerkungen

Da JLISA drei verschiedene Auswahlkriterien  $\mathcal{E}(\mathcal{I})$  unterstützt, gelten fortan die folgenden Abkürzungen:

- **Lisa1:** Beschreibt die Auswahl nach:  $\mathcal{E}(\mathcal{I}) = profit(\mathcal{I})$
- **Lisa2:** Beschreibt die Auswahl nach:  $\mathcal{E}(\mathcal{I}) = \frac{profit(\mathcal{I})}{size(\mathcal{I})}$
- **Lisa3:** Beschreibt die Auswahl nach Überlappungen:  $\mathcal{E}(\mathcal{I}) = \mathcal{O}(\mathcal{I})$

Der DB2 Design Advisor liefert bei der Workloadoptimierung, vermutlich durch TRY\_VARIATION, verschiedene Ergebnisse. Im Rahmen dieser Arbeit wurde der DB2 Design Advisor für jede Größenschranke jeweils drei Mal ausgeführt. Aus diesen drei Testläufen wurde dann das beste Ergebnis materialisiert und mit db2batch getestet. Die Laufzeit für TRY\_VARIATION wurde immer mit zehn Minuten vorgeben, vom Design Advisor aber niemals ausgeschöpft.

### 5.2.2 Erster Testlauf 300 Megabyte

In einem ersten Testlauf wurde eine Größenschranke von 300 Megabyte vorgegeben. Es stellte sich heraus, dass die Optimierung von Query 20 (unoptimierte Bearbeitungszeit rund 486 Sekunden) durch die lokal-optimale Indexkonfiguration 17 eine Beschleunigung von etwa 470 Sekunden erbrachte. Von Lisa1 wurde die Konfiguration 17 komplett und

von Lisa2 teil-materialisiert, so dass Query 20 im Durchschnitt 470 beziehungsweise 447 Sekunden schneller ausgeführt werden konnte. Die Lösung des DB2 Design Advisors unter der Größenschranke von 300 Megabyte beschleunigte Query 20 hingegen gar nicht. Für die Abarbeitung des gesamten Workloads ergaben sich folgende Zeiten:

- **Lisa1:** 335 Sekunden
- **Lisa2:** 353 Sekunden
- **DB2 Design Advisor:** 763 Sekunden

Da Query 20 unoptimiert über die Hälfte der Bearbeitungszeit des gesamten Workloads in Anspruch nimmt (vergleiche Tabelle 5.1), wurde es in den weiteren Betrachtungen aussen vor gelassen.

### 5.2.3 Lösungen 150 Megabyte

Für die Lösungen der folgenden Unterabschnitte sind in Tabellenform jeweils das Verfahren, die Indexe beziehungsweise Indekonfigurationen, der geschätzte Profit und die durch db2batch ermittelte Laufzeit angegeben.

Unter der Größenschranke von 150 Megabyte wurden die folgenden Indexe beziehungsweise Indekonfigurationen gewählt.

	Indexe	Konfig.	Profit	Laufszeit
Advisor	3, 4, 5, 10, 13, 15, 16, 19, 22, 24, 25, 28, 32, 33, 37, 46, 47	-	601.269	327 s
Lisa1	-	7, 11	245.527	337 s
Lisa2	-	6, 11, 13, 18	271.523	283 s
Lisa3	-	7, 11	245.527	337 s

Tabelle 5.3: Indexauswahl 150 MB

Durch die Indekonfiguration 7 konnte Query 7 um etwa 16 Sekunden beschleunigt werden. Die Indekonfiguration 11 führt zu einem Zeitgewinn von etwa 2,5 Sekunden. Unter Lisa1 und Lisa3 konnte der Workload mit den gewählten Indekonfigurationen in 337 Sekunden abgearbeitet werden.

Es ist anzumerken, dass die gewählten Konfigurationen durch Überlappungen auch Auswirkungen auf die Bearbeitungszeit anderer Queries haben. Unter Umständen wird ein materialisierter Index  $I$  durch ein Query  $Q$  genutzt, obwohl der  $I$  im RECOMMEND\_INDEXES-Modus nicht für  $Q$  empfohlen wurde. Auf Grund dieser Umstände verbessert sich die Bearbeitungszeit des Workloads um einen größeren Wert, als der Summe der Beschleunigungen optimierter Queries.

In Lisa2 ist vor allem die Indekonfiguration 18 interessant. Durch die Indekonfiguration 18 wird die Bearbeitungszeit von Query 21 auf etwa 16 Sekunden reduziert. Das entspricht einer Verbesserung von knapp 46 Sekunden.

### 5.2.4 Lösungen 300 Megabyte

	Indexe	Konfig.	Profit	Laufzeit
Advisor	1, 3, 5, 10, 14, 15, 16, 19, 20, 22, 24, 25, 27, 28, 29, 32, 35, 36, 37	-	1.014.641	308 s
Lisa1	-	7, 15, 18	455.471	276 s
Lisa2	-	6, 7, 9, 11, 13, 18	471.680	256 s
Lisa3	-	5, 18	143.429	291 s

Tabelle 5.4: Indexauswahl 300 MB

Unter der Größenschranke von 300 Megabyte nehmen nun auch Lisa1 und Lisa3 die Indexkonfiguration 18 in die Lösungsmenge auf, wodurch eine Verbesserung der Bearbeitungszeit des Workloads erreicht werden kann.

Lisa2 kann durch Hinzufügen der Konfigurationen 7 und 9 die entsprechenden Queries um 16 beziehungsweise 9 Sekunden beschleunigen. Die Gesamtlaufzeit wird auf 256 Sekunden verbessert.

### 5.2.5 Lösungen 700 Megabyte

	Indexe	Konfig.	Profit	Laufzeit
Advisor	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 19, 20, 21, 22, 24, 25, 26, 27, 28, 29, 32, 33, 34, 35, 37, 46, 47	-	1.428.014	224 s
Lisa1	-	7, 8, 11, 13, 15, 18	677.131	239 s
Lisa2	-	2, 3, 6, 7, 9, 10, 11, 13, 15, 18	861.235	237 s
Lisa3	-	5, 8, 9, 11, 13	382.584	296 s

Tabelle 5.5: Indexauswahl 700 MB

Es fällt hier zunächst auf, dass der DB2 Design Advisor offensichtlich nur Indexe nutzt, die auch JLISA zur Verfügung stehen und durch den RECOMMEND\_INDEXES-Modus erzeugt werden.

Die Bearbeitungszeit des Workloads ist hier mit der Lösung vom DB2 Design Advisor am besten. Die Indexauswahl nach Anzahl der Überlappungen (Lisa3) liefert unter 700 Megabyte eine sehr schlechte Lösung.

Es bestätigt sich hier das Problem, dass durch LISA weniger Queries optimiert werden, als bei Auswahlverfahren auf Basis einzelner Indexe.

### 5.3 Zusammenfassung

Lisa wurde durch das Programm JLISA umgesetzt und anhand des TPC-H Benchmarkes mit den Ergebnissen des DB2 Design Advisors verglichen. Die Resultate sind in der nachfolgenden Abbildung dargestellt und wurden in den vorhergehenden Abschnitten genauer beschrieben.

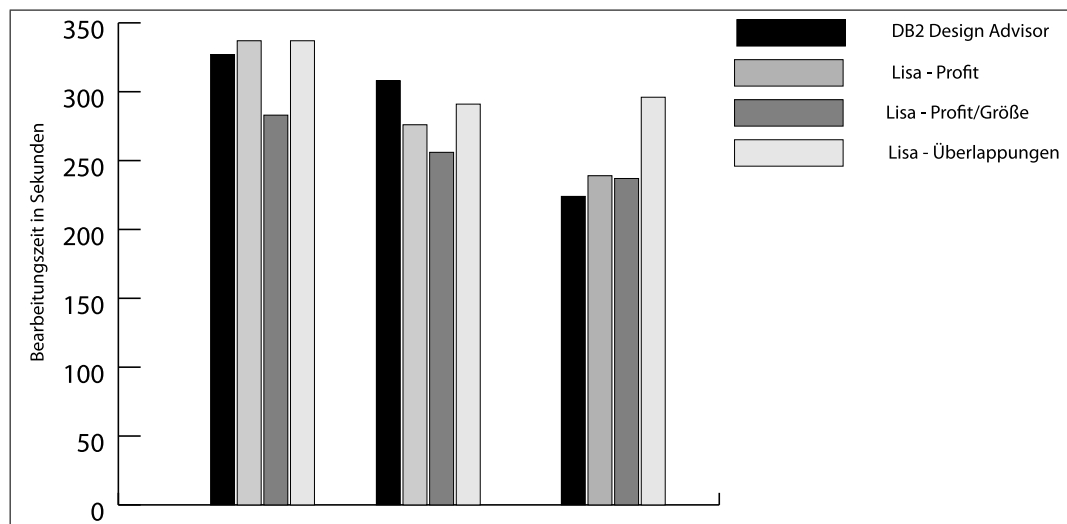


Abbildung 5.1: Lisa - DB2 Design Advisor

Die besten Ergebnisse erzielt Lisa unter mittleren Größenschranken, bezogen auf die Größe der optimierten Relationen.



## Kapitel 6

# Zusammenfassung und Ausblick

Aktuelle Ansätze zur Lösung des Index-Selection-Problems im Rahmen des Datenbank-Selbsttuning arbeiten als Rucksackverfahren auf Basis einzelner Indexe. Am Beispiel des DB2 Design Advisors wurde in dieser Arbeit untersucht, welche Probleme sich dadurch in Bezug auf eine optimale Indexauswahl ergeben. Der DB2 Design Advisor erstellt beim Berechnen einer Indexauswahlempfehlung für jedes Query eine Menge von Indexen, die als optimal angesehen werden. Aus diesen lokal-optimalen Indexkonfigurationen werden im weiteren Verlauf der Lösungserstellung einzelne Indexe ausgewählt. Es wurde festgestellt, dass Abhängigkeiten zwischen Indexen bei dieser Auswahl zum Teil unberücksichtigt bleiben und die Profitberechnung einzelner Indexe in lokal-optimalen Indexkonfigurationen problematisch ist.

Mit LISA wurde ein Verfahren entwickelt, welches bei der Indexauswahl nur gesamte lokal-optimale Indexkonfigurationen berücksichtigt. Indexabhängigkeiten in lokal-optimalen Indexkonfigurationen wird damit Rechnung getragen und die Profitberechnung für einzelne Indexe entfällt. Um bessere Ergebnisse erzielen zu können, werden in LISA Überlappungen von Indexkonfigurationen, die durch gleiche Indexe in verschiedenen Konfigurationen entstehen, behandelt. Neben der Anpassung der Konfigurationsgrößen wird in LISA eine Neuberechnung des Profites von Indexkonfigurationen durchgeführt. Diese Profitneuberechnung findet immer dann statt, wenn eine Indexkonfiguration teilmaterialisiert wird. Der neue Profit wird dann mit den noch nicht materialisierten Indexen berechnet.

LISA wurde in JLISA implementiert und für den TPC-H Workload getestet. Die Ergebnisse unter verschiedenen Größenschranken wurden dabei mit denen des DB2 Design Advisors verglichen. Bei kleinen Größenschranken konnte JLISA bessere Ergebnisse erzielen, als der DB2 Design Advisor. Mit wachsenden Größenschranken gingen diese Vorteile jedoch verloren, da die Indexauswahl JLISAs weniger Queries optimieren konnte, als die Indexauswahl des DB2 Design Advisors. Auf Grund der Eigenschaft immer nur ganze Indexkonfigurationen auszuwählen, sind die Ergebnisse LISAs abhängig vom jeweils optimierten Workload.

Ein Einsatz LISAs als online-Verfahren wurde in einem kurzen Abschnitt diskutiert, in JLISA aber nicht berücksichtigt. Da LISA auf externe Optimizer angewiesen ist und zu zusätzlichen EXPLAIN-Anfragen führt, verursacht LISA im online-Einsatz vermutlich einen unvermeidbaren Aufwand.

Die Indexauswahl bezogen auf lokal-optimale Indexkonfigurationen könnte problemlos

im DB2 Design Advisor implementiert werden. Das Testen verschiedener Indexkombinationen durch TRY\_VARIATION (siehe Kapitel 3) könnte entfallen. Stattdessen könnte die für diesen Schritt vorgesehene Zeit für die Größen- und Profitneuberechnungen überlappender Queries genutzt werden.

In zukünftigen Weiterentwicklungen LISAs sollte der Versuch unternommen werden, die Indexkonfigurationen sinnvoll zu zerlegen um kleiner Objekte bei der Indexauswahl zur Verfügung zu haben. Auf diese Weise könnten durch LISA mehrere Queries optimiert werden. Problematisch ist hierbei aber wieder die Profitberechnung der geteilten Indexkonfigurationen. Auch neue Berechnungen des Wertes  $\mathcal{E}(\mathcal{I})$  sind vorstellbar.

# Anhang A

## Anhang

### A.1 TPC-H

Der TPC-H Benchmark [Cou08] wurde entwickelt um die Leistungsfähigkeit von Datenbanksystemen zu messen. Er wurde vom *Transaction Processing Performance Council* entworfen und gilt als einer der Standardbenchmarks im Datenbankbereich. Die Tabellen des Benchmarks sind wie folgt definiert:

```
TABLE NATION ( N_NATIONKEY INTEGER NOT NULL, N_NAME CHAR(25)
NOT NULL, N_REGIONKEY INTEGER NOT NULL, N_COMMENT VAR-
CHAR(152))
```

```
TABLE REGION ( R_REGIONKEY INTEGER NOT NULL, R_NAME CHAR(25)
NOT NULL, R_COMMENT VARCHAR(152))
```

```
TABLE PART ( P_PARTKEY INTEGER NOT NULL, P_NAME VARCHAR(55)
NOT NULL, P_MFGR CHAR(25) NOT NULL, P_BRAND CHAR(10) NOT
NULL, P_TYPE VARCHAR(25) NOT NULL, P_SIZE INTEGER NOT NULL,
P_CONTAINER CHAR(10) NOT NULL, P_RETAILPRICE DECIMAL(15,2) NOT
NULL, P_COMMENT VARCHAR(23) NOT NULL )
```

```
TABLE SUPPLIER ( S_SUPPKEY INTEGER NOT NULL, S_NAME CHAR(25)
NOT NULL, S_ADDRESS VARCHAR(40) NOT NULL, S_NATIONKEY INTEGER
NOT NULL, S_PHONE CHAR(15) NOT NULL, S_ACCTBAL DECIMAL(15,2) NOT
NULL, S_COMMENT VARCHAR(101) NOT NULL)
```

```
TABLE PARTSUPP ( PS_PARTKEY INTEGER NOT NULL, PS_SUPPKEY IN-
TEGER NOT NULL, PS_AVAILQTY INTEGER NOT NULL, PS_SUPPLYCOST DE-
CIMAL(15,2) NOT NULL, PS_COMMENT VARCHAR(199) NOT NULL )
```

```
TABLE CUSTOMER ( C_CUSTKEY INTEGER NOT NULL, C_NAME VAR-
CHAR(25) NOT NULL, C_ADDRESS VARCHAR(40) NOT NULL, C_NATIONKEY
INTEGER NOT NULL, C_PHONE CHAR(15) NOT NULL, C_ACCTBAL DECI-
MAL(15,2) NOT NULL, C_MKTSEGMENT CHAR(10) NOT NULL, C_COMMENT
VARCHAR(117) NOT NULL)
```

```
TABLE ORDERS ( O_ORDERKEY INTEGER NOT NULL, O_CUSTKEY INTE-
GER NOT NULL, O_ORDERSTATUS CHAR(1) NOT NULL, O_TOTALPRICE DECI-
MAL(15,2) NOT NULL, O_ORDERDATE DATE NOT NULL, O_ORDERPRIORITY
CHAR(15) NOT NULL, O_CLERK CHAR(15) NOT NULL, O_SHIPPRIORITY INTE-
```

GER NOT NULL, O\_COMMENT VARCHAR(79) NOT NULL)

TABLE LINEITEM ( L\_ORDERKEY INTEGER NOT NULL, L\_PARTKEY INTEGER NOT NULL, L\_SUPPKEY INTEGER NOT NULL, L\_LINENUMBER INTEGER NOT NULL, L\_QUANTITY DECIMAL(15,2) NOT NULL, L\_EXTENDEDPRICE DECIMAL(15,2) NOT NULL, L\_DISCOUNT DECIMAL(15,2) NOT NULL, L\_TAX DECIMAL(15,2) NOT NULL, L\_RETURNFLAG CHAR(1) NOT NULL, L\_LINESTATUS CHAR(1) NOT NULL, L\_SHIPDATE DATE NOT NULL, L\_COMMITDATE DATE NOT NULL, L\_RECEIPTDATE DATE NOT NULL, L\_SHIPINSTRUCT CHAR(25) NOT NULL, L\_SHIPMODE CHAR(10) NOT NULL, L\_COMMENT VARCHAR(44) NOT NULL)

In den Tabellen existieren die Folgenden Schlüssel. Auf den Primärschlüsseln wird in DB2 automatisch ein Primärindex erzeugt.

REGION: PRIMARY KEY (R\_REGIONKEY)

NATION: PRIMARY KEY (N\_NATIONKEY), FOREIGN KEY NATION\_FK1 (N\_REGIONKEY) references REGION

PART: PRIMARY KEY (P\_PARTKEY)

SUPPLIER: PRIMARY KEY (S\_SUPPKEY), FOREIGN KEY SUPPLIER\_FK1 (S\_NATIONKEY) references NATION

PARTSUPP: PRIMARY KEY (PS\_PARTKEY,PS\_SUPPKEY), FOREIGN KEY PARTSUPP\_FK2 (PS\_PARTKEY) references PART, FOREIGN KEY PARTSUPP\_FK1 (PS\_SUPPKEY) references SUPPLIER

CUSTOMER: PRIMARY KEY (C\_CUSTKEY), FOREIGN KEY CUSTOMER\_FK1 (C\_NATIONKEY) references NATION

LINEITEM: PRIMARY KEY (L\_ORDERKEY,L\_LINENUMBER), FOREIGN KEY LINEITEM\_FK1 (L\_ORDERKEY) references ORDERS, FOREIGN KEY LINEITEM\_FK2 (L\_PARTKEY,L\_SUPPKEY) references PARTSUPP

ORDERS: PRIMARY KEY (O\_ORDERKEY), FOREIGN KEY ORDERS\_FK1 (O\_CUSTKEY) references CUSTOMER

Im Rahmen dieser Arbeit wurden diese Tabellen mit 1 Gigabyte Daten gefüllt. Die Datensatzverteilung ist dabei wie folgt:

- CUSTOMER: 150.000 Datensätze
- LINEITEM: 6.001.215 Datensätze
- NATION: 25 Datensätze
- ORDERS: 1.500.000 Datensätze
- PART: 200.000 Datensätze
- PARTSUPP: 800.001 Datensätze
- REGION: 5 Datensätze
- SUPPLIER: 10.000 Datensätze

---

---

Auf den beschriebenen Tabellen werden im TPC-H Benchmark 22 Standard-Queries ausgeführt, deren Parameter zufällig erzeugt werden können. Im Rahmen dieser Arbeit wurden jedoch nur 20 dieser Queries verwendet.

## A.2 Explain Tables

In den EXPLAIN-Tabellen werden Informationen über Anfragepläne und der damit verbundenen Kosten gespeichert. JLISA nutzt die Tabelle EXPLAIN\_OPERATOR, um die geschätzten Kosten von Queries auszulesen. Die ADVISE-Tabellen speichern Informationen, die bei der Arbeit des DB2 Design Advisors entstehen, oder von diesem benötigt werden. JLISA nutzt die ADVISE\_INDEX-Tabelle um empfohlene Indexe zu erhalten. Der Aufbau der beiden Tabellen kann dem DB2 Informationszentrum [IBM08] entnommen werden.

## A.3 Testsystem

Die in dieser Arbeit verwendete Datenbank *DB2 Universal Database 9.5 Express-C* lief unter Windows XP 32Bit auf einem PC-System: Intel Core2Duo E8400, 2GB-Ram, 160GB SATA-II Festplatte.

## A.4 Indexabhängigkeiten I

Anfragepläne für *select \* from partsupp, part where part.p\_partkey = partsupp.ps\_partkey* unter verschiedenen Indexkonfigurationen. In Klammern notiert ist der kummulierte Aufwand des Planes bis zur und einschliesslich der aktuellen Operation. Siehe [IBM08] für weitere Informationen zu Anfrageplänen in DB2.

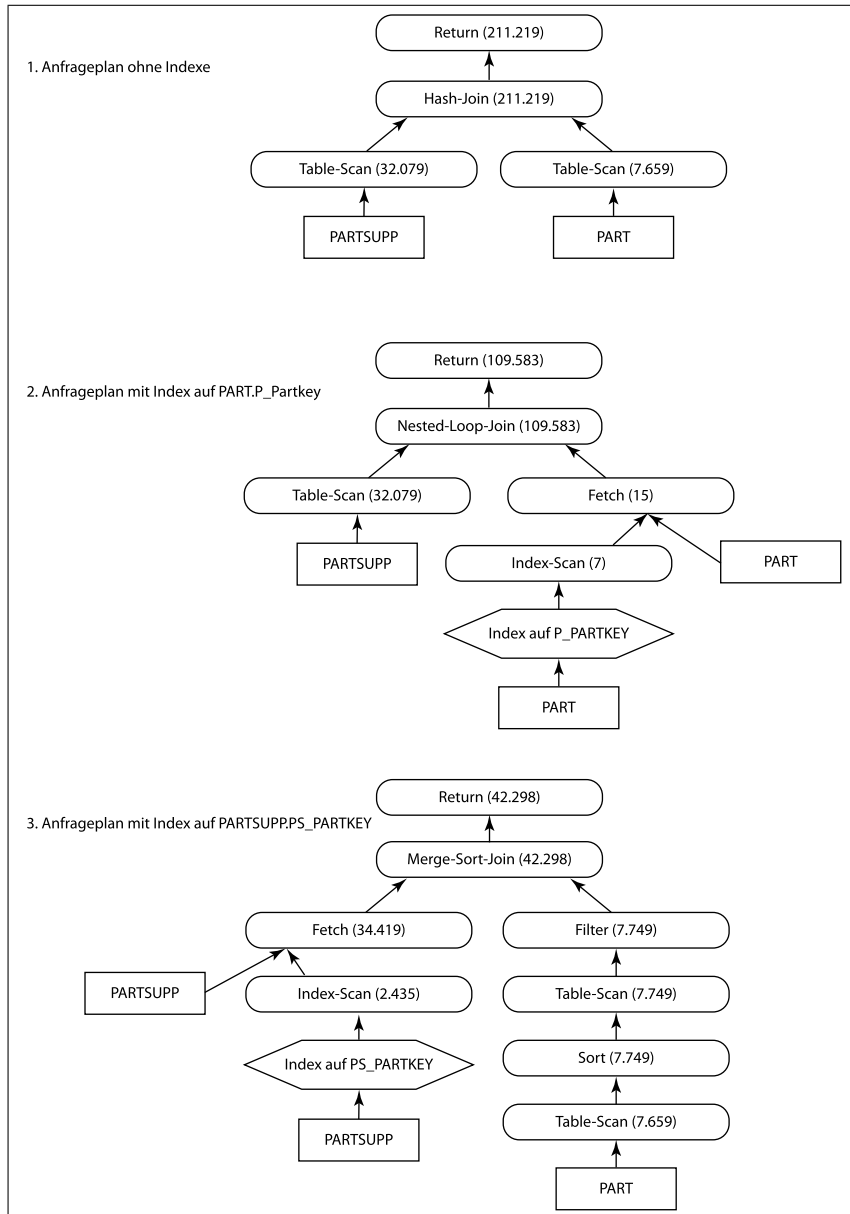


Abbildung A.1: Indexabhängigkeiten

## A.5 Indexabhängigkeiten II

Anfragepläne für *select \* from lineitem2, lineitem where lineitem2.L\_orderkey = lineitem.L\_orderkey* in Analogie zu Anhang A.4. Zur Vereinfachung wurden die Kosten einiger Teilbäume hier gleichgesetzt, obwohl es in der Praxis zu geringen Differenzen kam.

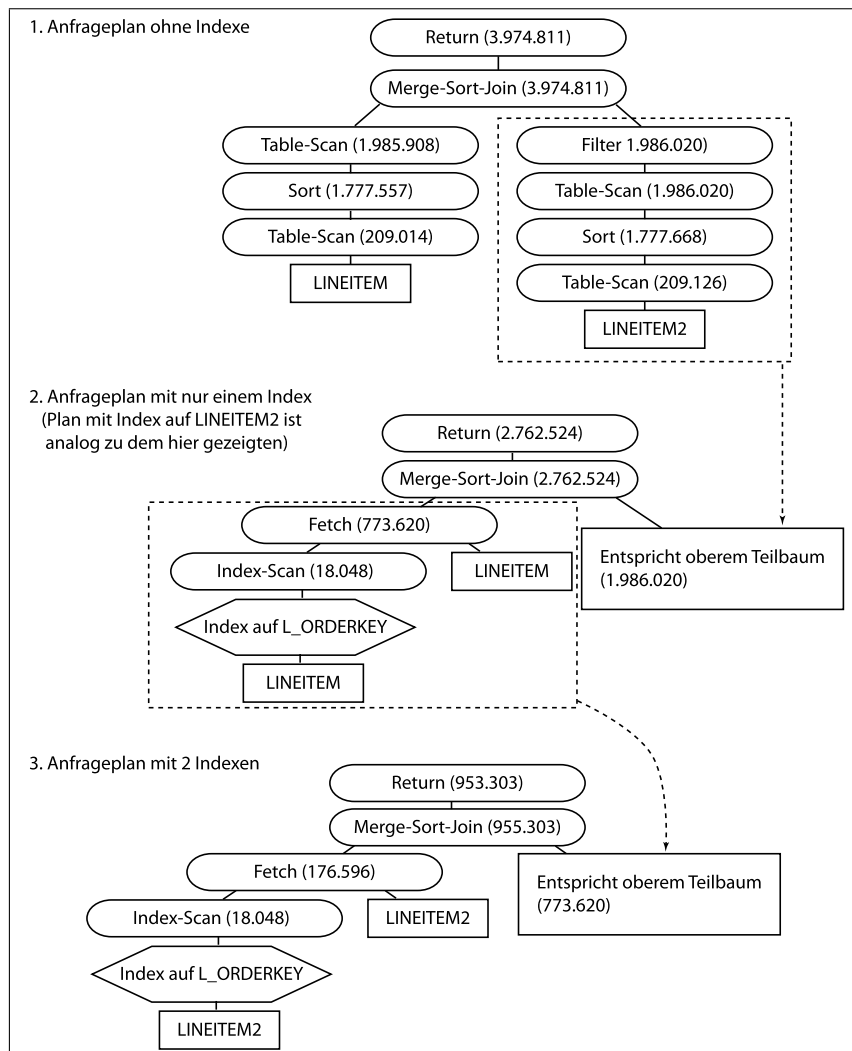


Abbildung A.2: Indexabhängigkeiten II

## A.6 Indexempfehlungen für den verwendeten Workload

Für den Workload in Kapitel 5 wurden folgende Indexe empfohlen. Plus und Minus vor dem Spaltennamen beschreiben die Sortierung.

Index-ID	Spalten	Größe (KB)
1	+P_SIZE +P_MFGR +P_TYPE +P_PARTKEY	18.780
2	+PS_PARTKEY +PS_SUPPLYCOST +PS_SUPPKEY	23.634
3	+S_SUPPKEY -S_NATIONKEY	188
4	+R_NAME -R_REGIONKEY	25
5	+N_REGIONKEY +N_NATIONKEY	20
6	+PS_SUPPLYCOST +PS_PARTKEY +PS_SUPPKEY	23.634
7	+S_SUPPKEY - S_ACCTBAL+S_COMMENT +S_PHONE +S_ADDRESS +S_NAME +S_NATIONKEY	2.506
8	+R_REGIONKEY -R_NAME	25
9	+O_ORDERDATE +O_SHIPPRIORITY +O_ORDERKEY +O_CUSTKEY	44.299
10	+C_MKTSEGMENT +C_CUSTKEY	4.002
11	+O_ORDERDATE +O_ORDERPRIORITY +O_ORDERKEY	59.979
12	+L_ORDERKEY +L_RECEIPTDATE +L_COMMITDATE	88.657
13	+C_CUSTKEY - C_NATIONKEY	2.661
14	+R_NAME +R_REGIONKEY	25
15	+N_REGIONKEY +N_NAME +N_NATIONKEY	20
16	+S_NATIONKEY +S_SUPPKEY	188
17	+L_SUPPKEY +L_DISCOUNT +L_EXTENDEDPRICE +L_ORDERKEY	249.927
18	+O_ORDERDATE +O_ORDERKEY +O_CUSTKEY	35.429
19	+L_SHIPDATE -L_DISCOUNT	44.587
20	+L_SHIPDATE +L_QUANTITY	39.908
21	+L_SUPPKEY +L_SHIPDATE +L_DISCOUNT +L_EXTENDEDPRICE +L_ORDERKEY	108.640

22	+O_ORDERDATE +O_CUSTKEY +O_ORDERKEY	35.429
23	+L_PARTKEY +L_DISCOUNT +L_EXTENDEDPRICE +L_ORDERKEY +L_SUPPKEY	286.744
24	+C_NATIONKEY +C_CUSTKEY	2.661
25	+P_TYPE +P_PARTKEY	9.809
26	+O_ORDERKEY O_ORDERDATE	- 2.6505
27	+L_RETURNFLAG	70.361
28	+N_NAME +N_NATIONKEY	20
29	+PS_SUPPKEY +PS_AVAILQTY +PS_SUPPLYCOST +PS_PARTKEY	28.489
30	+L_SHIPMODE +L_RECEIPTDATE +L_SHIPDATE +L_COMMITDATE +L_ORDERKEY	266.898
31	+L_SHIPDATE +L_DISCOUNT +L_EXTENDEDPRICE +L_PARTKEY	249.927
32	+P_SIZE +P_BRAND +P_TYPE +P_PARTKEY	3.196
33	+PS_PARTKEY PS_SUPPKEY	- 14.146
34	+O_ORDERKEY O_TOTALPRICE +O_ORDERDATE +O_CUSTKEY	- 53.407
35	+C_CUSTKEY -C_NAME	6.680
36	+L_SHIPMODE +L_SHIPINSTRUCT +L_DISCOUNT +L_EXTENDEDPRICE +L_QUANTITY +L_PARTKEY	59.519
37	+P_BRAND +P_SIZE +P_CONTAINER +P_PARTKEY	1.437
38	+P_NAME -P_PARTKEY	13.403
39	+S_SUPPKEY +S_NAME +S_ADDRESS +S_NATIONKEY	1.006
40	+L_PARTKEY +L_SUPPKEY	141.717

41	+PS_PARTKEY +PS_SUPPKEY +PS_AVAILQTY	18.901
42	+O_ORDERSTATUS +O_ORDERKEY	19.866
43	+S_NATIONKEY +S_NAME +S_SUPPKEY	564
44	+L_SUPPKEY +L_COMMITDATE +L_RECEIPTDATE +L_ORDERKEY	106.447
45	+L_ORDERKEY +L_SUPPKEY +L_COMMITDATE +L_RECEIPTDATE	106.447
46	+C_ACCTBAL +C_CUSTKEY +C_PHONE	1.107
47	+O_CUSTKEY	9592

Tabelle A.1: Indexe Workload

---

---

# Literaturverzeichnis

- [AB04] Andreas Bauer, H. G.: *Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung*. dpunkt-Verlag, Heidelberg, 2. Auflage, 2004.
- [AH01] Andreas Heuer, K.-U. S., G. S.: *Datenbanken kompakt*. MITP-Verlag GmbH, Bonn, 1. Auflage, 2001.
- [BC06] Bruno, N.; Chaudhuri, S.: To tune or not to tune? a lightweight physical design alerter. In *VLDB*, S. 499–510, 2006.
- [BC07] Bruno, N.; Chaudhuri, S.: An online approach to physical design tuning. In *ICDE*, S. 826–835, 2007.
- [BM72] Bayer, R.; McCreight, E. M.: Organization and maintenance of large ordered indices. *Acta Inf.*, Band 1, S. 173–189, 1972.
- [CFM95] Caprara, A.; Fischetti, M.; Maio, D.: Exact and approximate algorithms for the index selection problem in physical database design. *IEEE Trans. Knowl. Data Eng.*, Band 7, Nr. 6, S. 955–967, 1995.
- [CN98] Chaudhuri, S.; Narasayya, V. R.: Microsoft index tuning wizard for sql server 7.0. In *SIGMOD Conference*, S. 553–554, 1998.
- [CN99] Chaudhuri, S.; Narasayya, V. R.: Index merging. In *ICDE*, S. 296–303, 1999.
- [Com78] Comer, D.: The difficulty of optimum index selection. *ACM Trans. Database Syst.*, Band 3, Nr. 4, S. 440–445, 1978.
- [Cou08] Council, T. P. P.: Transaction processing performance council. <http://www.tpc.org/default.asp>, 2008. [Online; accessed 01-March-2008].
- [FST88] Finkelstein, S. J.; Schkolnick, M.; Tiberio, P.: Physical database design for relational databases. *ACM Trans. Database Syst.*, Band 13, Nr. 1, S. 91–128, 1988.
- [GMUW00] Garcia-Molina, H.; Ullman, J. D.; Widom, J.: *Database System Implementation*. Prentice-Hall, 1. Auflage, 2000.
- [H87] Härder, T.: Realisierung von operationalen schnittstellen. In *Datenbank-handbuch*, S. 163–335. 1987.

- [IBM08] IBM Corporation: Ibm db2 9.5 information center for linux, unix, and windows. <http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp>, 2008. [Online; accessed 01-Juli-2008].
- [Knu03] Knuth, D. E.: *The art of computer programming Vol. 3*. Addison-Wesley, Boston, 2. Auflage, 2003.
- [KPP04] Kellerer, H.; Pferschy, U.; Pisinger, D.: *Knapsack Problems*. Springer, Berlin, Germany, 2004.
- [LSS07] Lühning, M.; Sattler, K.-U.; Schallehn, E.; 0002, K. S.: Autonomes index tuning - dbms-integrierte verwaltung von soft indexen. In *BTW*, S. 152–171, 2007.
- [OOW93] O’Neil, E. J.; O’Neil, P. E.; Weikum, G.: The lru-k page replacement algorithm for database disk buffering. In *SIGMOD Conference*, S. 297–306, 1993.
- [Ora03] Oracle Corporation: *Performance Tuning using the SQLAccess Advisor, Oracle White Paper*. 2003. [Online; accessed 01-August-2008].
- [SAP08] SAP Deutschland AG & Co. KG: Sap deutschland - unternehmensanwendungen, software-lösungen und services. <http://www.sap.com/germany/index.epx>, 2008. [Online; accessed 01-Juli-2008].
- [SB03] Shasha, D.; Bonnet, P.: *Database Tuning*. Morgan Kaufmann Publishers, San Francicso, 1. Auflage, 2003.
- [SD06] Sanjoy Dasgupta, U. V., C. H. P.: *Algorithms*. McGraw-Hill, 1. Auflage, 2006.
- [SGS03] Sattler, K.-U.; Geist, I.; Schallehn, E.: Quiet: Continuous query-driven index tuning. In *VLDB*, S. 1129–1132, 2003.
- [SH99] Saake, G.; Heuer, A.: *Datenbanken: Implementierungstechniken*. MITP-Verlag, 1. Auflage, 1999.
- [SKS97] Silberschatz, A.; Korth, H. F.; Sudarshan, S.: *Database System Concepts, 3rd Edition*. McGraw-Hill Book Company, 3. Auflage, 1997.
- [SSG04] Sattler, K.-U.; Schallehn, E.; Geist, I.: Autonomous query-driven index tuning. In *IDEAS*, S. 439–448, 2004.
- [SSV96] Scheuermann, P.; Shim, J.; Vingralek, R.: Watchman : A data warehouse intelligent cache manager. In *VLDB*, S. 51–62, 1996.
- [Vos99] Vossen, G.: *Datenbankmodelle, Datenbanksprachen und Datenbankmanagementsysteme*. Oldenbourg, München, 3. Auflage, 1999.
- [VZZ<sup>+</sup>00] Valentin, G.; Zuliani, M.; Zilio, D. C.; Lohman, G. M.; Skelley, A.: Db2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, S. 101–110, 2000.

- 
- 
- [WHMZ94] Weikum, G.; Hasse, C.; Moenkeberg, A.; Zabback, P.: The comfort automatic tuning project, invited project review. *Inf. Syst.*, Band 19, Nr. 5, S. 381–432, 1994.
- [ZRL<sup>+</sup>04] Zilio, D. C.; Rao, J.; Lightstone, S.; Lohman, G. M.; Storm, A. J.; Garcia-Arellano, C.; Fadden, S.: Db2 design advisor: Integrated automatic physical database design. In *VLDB [VZZ<sup>+</sup>00]*, S. 1087–1097.



# Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 18. Februar 2009

Kersten Kühne

