

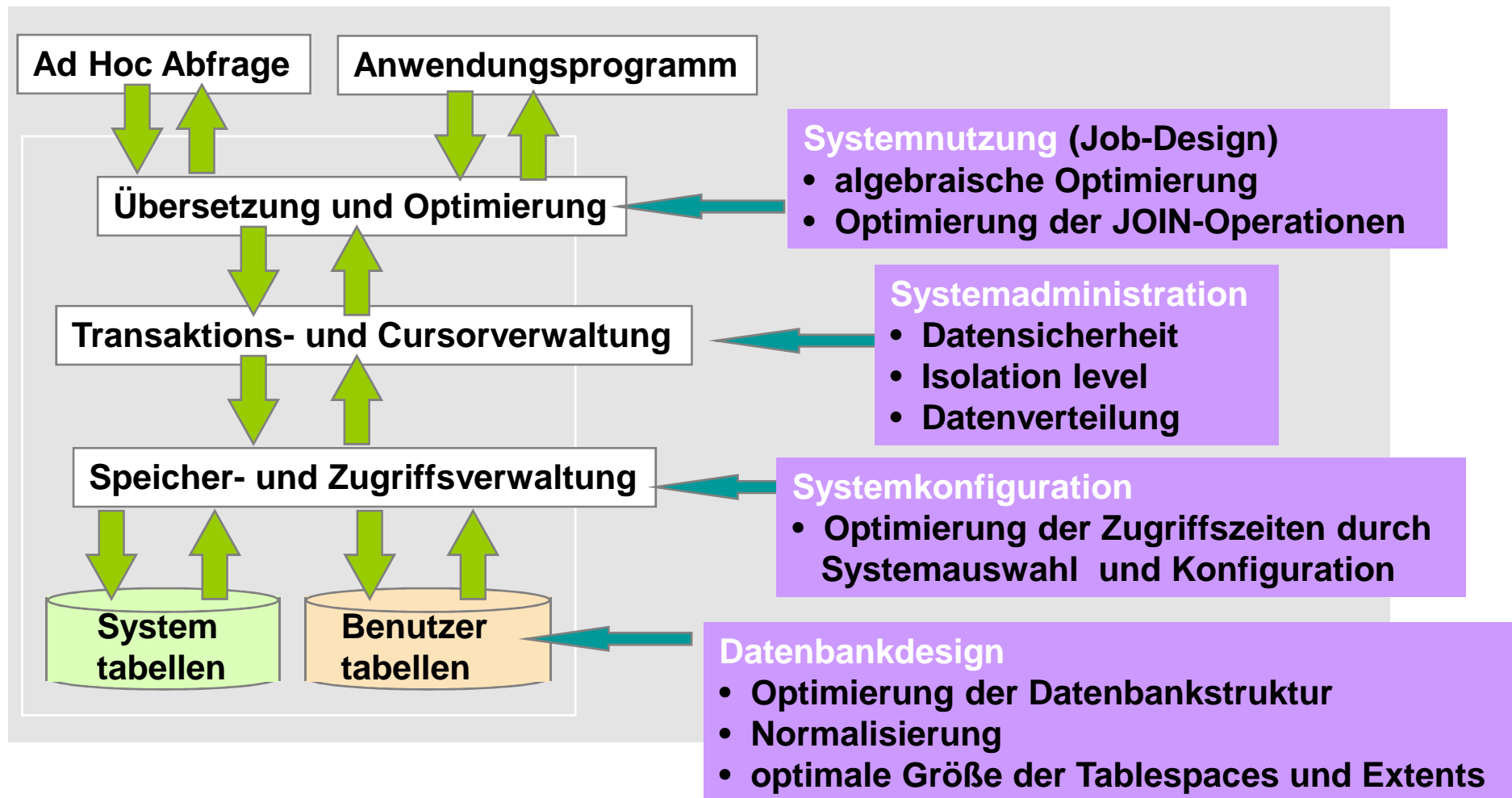
WP Datenbankdesign



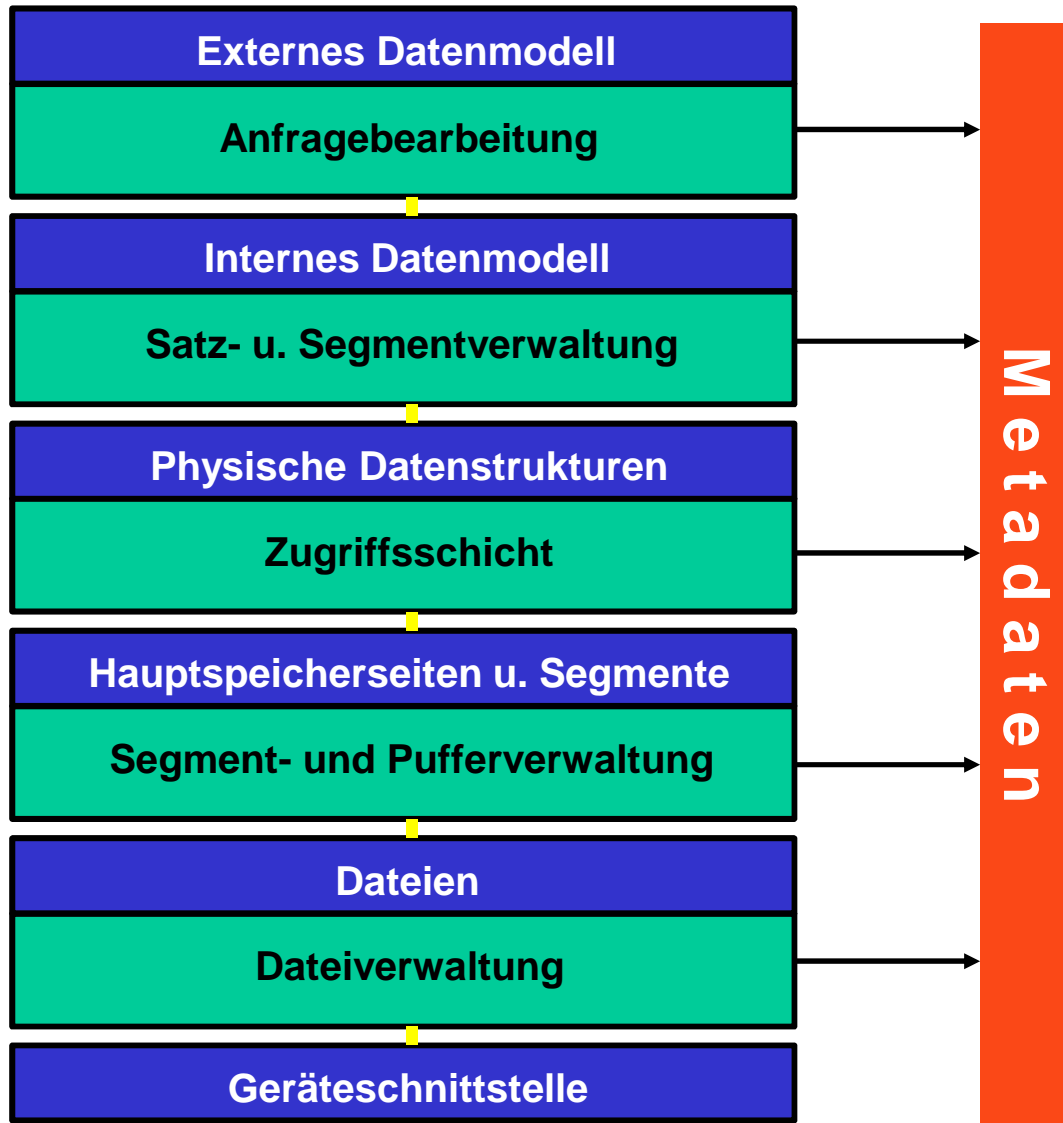
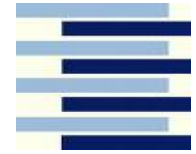
Kapitel 3: Datenbankoptimierung

- Überblick
- Datenbankdesign
- Ablauf bei der Ausführung von SQL
- Optimierung von SQL Befehlen

Einflussmöglichkeiten bei SQL-Abfragen



5-Schichten-Referenzmodell



Jede Schicht trifft Entscheidungen auf Basis von

1. Wissen über Daten

- Semantische Nähe
- Konstruktoren

2. Wissen über Anfragen

- Unterstützung von Selektionen, Sortierungen, Joins usw. durch Indexstrukturen
- Zusammenfassen häufig zugegriffener Attribute in gemeinsamen Dateien
- Pufferung häufig gelesener Daten im Hauptspeicher

Datenbankdesign (physisch)



- Parameter und Zuordnung eines Tablespace zu einem Datenträger
- Größe und Anzahl der Dateien eines Tablespace
- Zuordnung einer Tabelle oder eines Indexes zu einem Tablespace
⇒ *Partitionierung*
- Indizierung einer Tabellenspalte oder mehrerer Spalten
- Anlegen von Tabellen in Index- oder Hash-Clustern

Beispiele für Systemparameter

Blockgröße

Intern verwendeter Code (Unicode, US ASCII, ...)

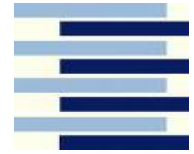
Größe der Bereiche für DB-Cache, internes Sortieren, ...

Isolation Level für SQL

Anzahl paralleler Transaktionen

Übersicht über Oracle-Parameter: show parameters;
show sga;

Tuning der Datenbank



- Blockgröße mind. (8 KB)
- Auf ausreichend großen Datencache achten (Trefferquote beobachten)
- REDO-LOG-Dateien nicht zu klein wählen (je nach Datenaufkommen ca. 10 bis 200 MB)
- Anzahl Rollback-Segmente ausreichend groß wählen: Anzahl **gleichzeitiger** Transaktionen = Anzahl Rollback-Segmente
- Überflüssige Checkpoints vermeiden:
Parameter `log_checkpoint_interval=1000000`
Parameter `log_checkpoint_timeout=0`
- Indizes auf Tabellen, in denen häufig gelöscht wird, regelmäßig reorganisieren: `alter index <index> rebuild` (möglichst, wenn keine Anwender arbeiten; auf ausreichend TEMP-Platz achten)
- Durch geeignete Storage Parameter die Anzahl der Extents nicht zu groß werden lassen (maximal einige Hundert)

www.datenbank-tuning.de

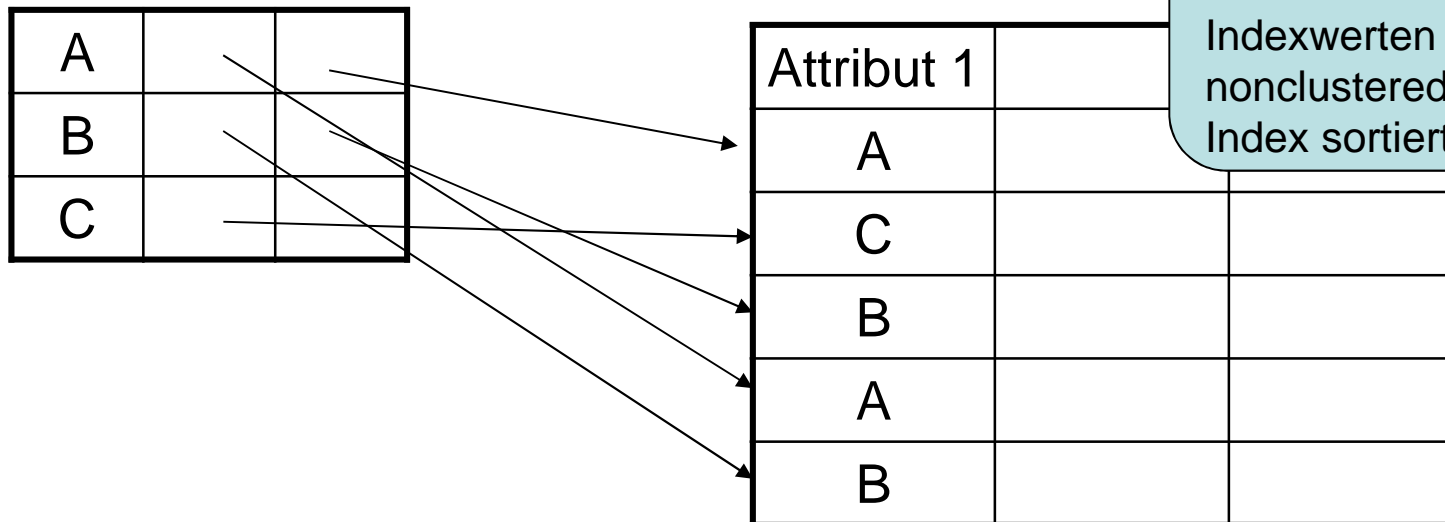
Indizes



Vorteil eines Index: Vermeidung von Full Table Scans

create [unique] [clustered | nonclustered]
index *indexname* on tablename (Spalte1 {Spalte}) ..

Index auf Attribut 1 ROWID



unique: nur 1 Datensatz je Indexwert.
clustered: Datensätze nach Indexwerten sortiert.
nonclustered: nicht nach Index sortiert.

1. Für Primary-Key Spalten wird automatisch ein Index angelegt.
2. Bei verketteten Indizes sollte die selektivste Spalte in der ersten Position stehen.

Empfehlungen für einen Index



Spricht für Index	Spricht gegen Index
Häufiges Select mit Indexspalten als Suchbedingung	Häufiges Update in der Tabelle
Sortierung nach Indexspalten wird häufig benötigt	
	Abfragen der Indexspalten liefern in der Regel mehr als 50% der Tupel zurück
Tabelle ist relativ groß	Tabellengröße < 80 KB
Viele unterschiedliche Werte in der Indexspalte	Es existieren nur wenig unterschiedliche Werte für die Indexspalte
Spaltenwerte sind relativ gut gestreut	Einige Spaltenwerte treten extrem gehäuft auf
Durchschnittslänge eines Indexwertes ist relativ klein	Durchschnittslänge relativ groß
Wichtige Programme laufen ohne Index nicht akzeptabel	Wichtige Programme laufen mit Index nicht akzeptabel

Bitmap Index



Statt Indextabelle mit Pointern (Rowld) je Attributwert eine Bitliste

Kennzeichen	Marke	Farbe
HH - A 100	VW	rot
HH - XY 33	Ford	blau
M - BY 112	VW	blau
KI - V 77	Ford	rot
HL – BZ 44	VW	grün

Create Bitmap Index TypIndex on Auto (Marke)

VW (1 0 1 0 1)

Ford (0 1 0 1 0)

Create Bitmap Index FarbeIndex on Auto(Farbe)

Rot (1 0 0 1 0)

Blau (0 1 1 0 0)

Grün (0 0 0 0 1)

Frage: Wann ist eine Bitliste speicherplatzgünstiger als eine Indextabelle?

N = Anzahl der Tupel,

|A| = Anzahl der Attributwerte des Attributs A

L = Länge eines Indexhinweises (Rowld bei Oracle) in Bit.

Indextabellen



Normalfall: Index und Tabelle getrennte Objekte, Adressierung über RowId
Bei Indextabellen sind die Tupel im Index-Baum enthalten.

```
Create table FB_LookUp
( FBNr          Varchar2(5) primary key,
  FbName        Varchar2(30),
  Dekan         VarChar2(10),
  Organization Index    Storage ... )
```

⇐ **Indexspalte**

Reine Indextabellen sollten dort eingesetzt werden, wo ...

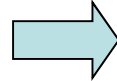
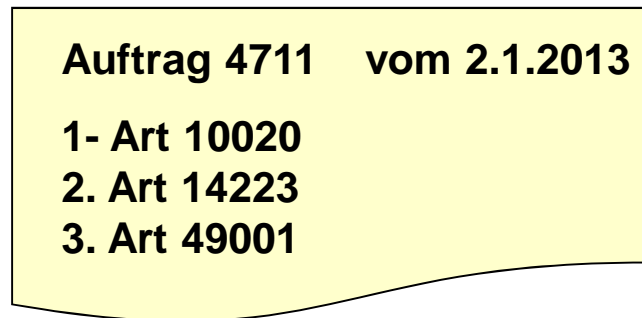
1. nur wenige Spalten vorhanden sind
2. die Werte regelmäßig über den PK ausgewählt werden
3. kaum (besser keine) Aktualisierungen, Löschungen stattfinden.

- Indextabellen enthalten keine RowIds
- Ihre Struktur ist nicht änderbar
- Es können keine weiteren (normalen) Indizes dafür erstellt werden.

Cluster



Optimierung der Speicherung von Owner- und Membersätzen (1:n-Beziehung). Alle Sätze, die den gleichen Cluster-Key haben, werden in einem physischen Datenblock gespeichert.



4711	2.1.2013	...
4711	10020	...
4711	14223	...
4711	49001	...
...

In einem
DB-Block

Create Cluster Auftrag (AuftrNr number(4)) Size 512 tablespace ts_user ;

Create Table Auftragskopf
(AuftrNr Number(4),
...
...)

Cluster Auftrag (AuftrNr);

Create Table Auftragsposition
(AuftrNr Number(4),
PosNr Number(3),
...)

Cluster Auftrag (AuftrNr);

Create Index Idx_Auftrag on Cluster Auftrag;

Datenbankdesign (logisch)



Eigenschaften normalisierter relationaler Datenbanken:

1. Schutz von Einfüge- und Löschanomalien
2. Redundanzfreiheit
3. Bildung vieler Relationen mit der Folge:
 - Schlüsselredundanz
 - Performanceprobleme
 - Widerspruch zur Objektorientierung, da Daten über ein Objekt evtl. auf mehrere Tabellen verteilt

Ziel des Modelltunings: Reduzierung der Anzahl physischer I/O's
Vereinfachung der SQL-Queries

Sollte erst durchgeführt werden, wenn andere Tuningmaßnahmen nicht greifen. Achtung: Verletzung der NF ?!

Optimierung durch Denormalisierung



Denormalisierung auf der externen Ebene:

1. Bildung von Views (insbes. Join)

Denormalisierung auf der konzeptionellen Ebene:

1. Zusammenfassung mehrerer Relationen zu einer
2. Gezieltes Schaffen von Redundanz
 - es wird ein hohes Zugriffspfadvolumen gespart
 - es handelt sich um wenige Datenelemente
 - die Änderungsrate ist gering
3. Einführung von Wiederholgruppen
 - das Vorkommen der Member-Records einer 1:N-Beziehung ist fest
 - das Vorkommen der Member-Records ist variabel, aber
 - ⇒ mit kleiner Satzlänge
 - ⇒ kleinem Min/Max-Bereich
 - ⇒ hohes Traversal-Volumen
4. Aufspalten einer Relation in mehrere
 - ein Teil wird häufig, ein anderer wenig verwendet



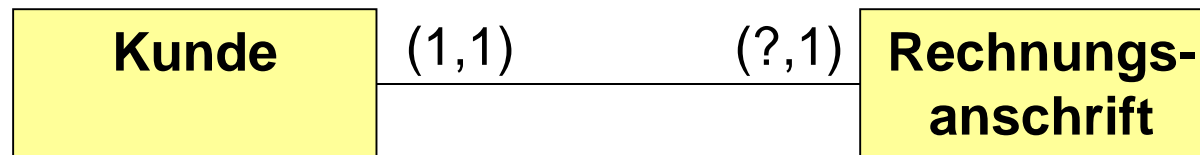
Beispiele zur Denormalisierung



1. Zusammenfassung mehrerer Relationen

a) es besteht eine $(1,1):(? ,1)$ - Beziehung

Beispiel:



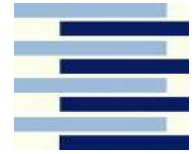
KUNDE(KDNR, NAME, ...)

RECHNUNGSANSCHRIFT(KDNR, STRASSE, PLZ, ORT)

KUNDE(KDNR, NAME, STRASSE, PLZ, ORT, ...)

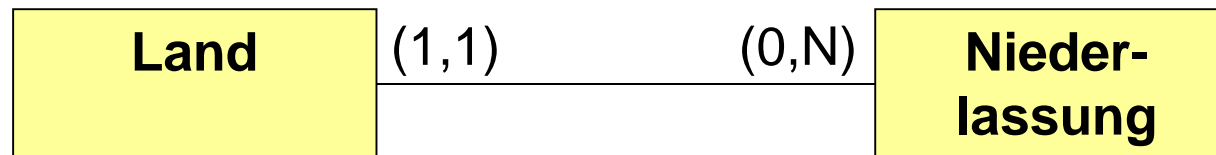
Bei $(1,1):(1,1)$ muss das sein

Beispiele zur Denormalisierung



- b) es besteht eine (1,1):(0,n)-Beziehung, wobei die Tupel auf der 1-Seite wenige Attribute mit geringer Länge haben

Beispiel:



LAND(LÄNDERKENNZ, LÄNDERNAME)
NIEDERLASSUNG(NLNR, LÄNDERKENNZ, ORT, ...)
NIEDERLASSUNG(NLNR, LÄNDERNAME, ORT, ...)

bei(1,1):(1,N) sollte das sein

Beispiele zur Denormalisierung



2. Gezieltes Schaffen von Redundanz

Beispiel: KUNDE(KDNR, NAME, GESAMTUMSATZ, ...)
BESTELLUNG(BESTNR, DATUM, KDNR, BESTELLWERT)

Achtung: Trigger zur Integritätssicherung notwendig

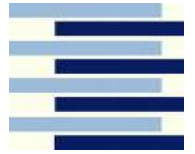
3. Einführung von Wiederholgruppen

es besteht eine (1,1):(0,n)- Beziehung mit wenigen Tupeln auf der n-Seite, die wenig Attribute und eine geringe Länge haben **Beurteilung ?**

Beispiel: KUNDE(KDNR, NAME, ...)
NIEDERLASSUNG(KDNR, PLZ, ORT)
KUNDE(KDNR, NAME, PLZ1, ORT1, PLZ2, ORT2, ...)

Beurteilung?

Beispiele zur Denormalisierung



4. Aufspaltung einer Relation in mehrere

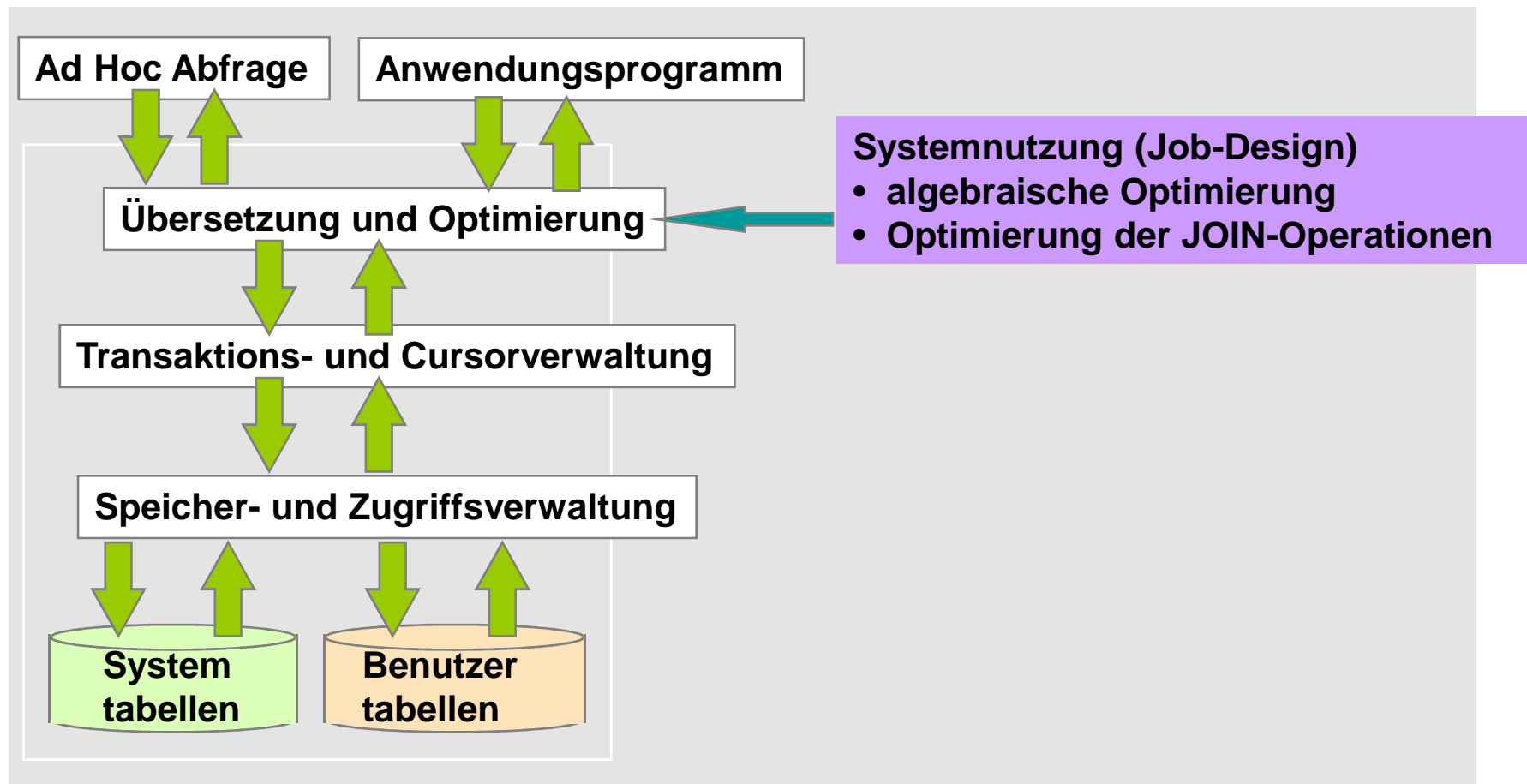
Ist dann sinnvoll, wenn nur auf bestimmte Teilmengen von Attributen zugegriffen wird.

Beispiel: MITARB(PERSNR, NAME, PLZ, ORT, GEHALT, ...)

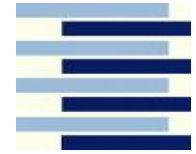
 MITARB_ADRESSE(PERSNR, STRASSE, PLZ, ORT)

 MITARB_GELD(PERSNR, GEHALT, BLZ, KONTONR, ...)

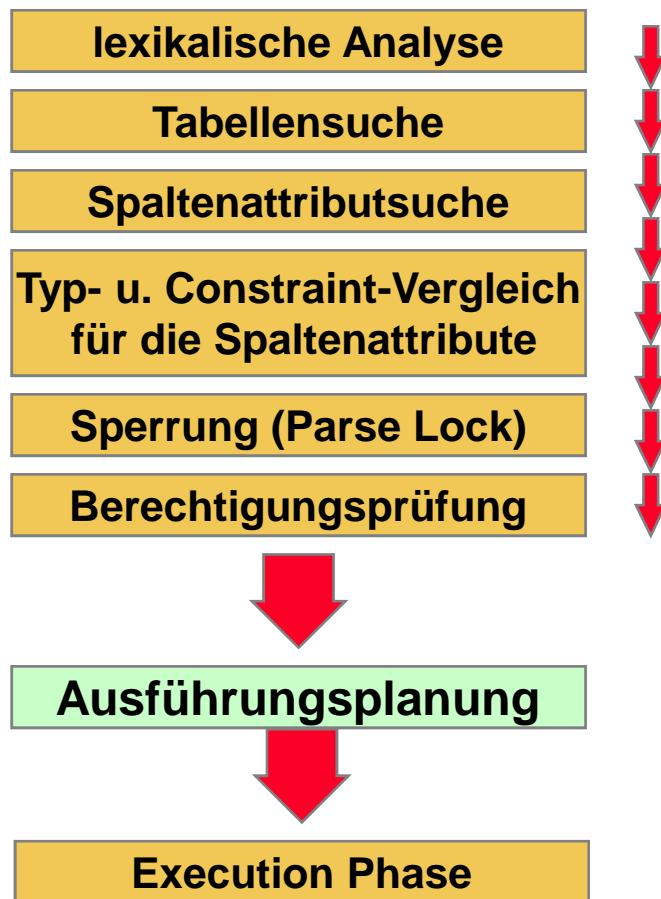
Einflussmöglichkeiten bei SQL-Abfragen



Abarbeitung eines SQL-Befehls



SQL-Befehle werden als Text an das Datenbankmanagementsystem übermittelt. Dieses muss den Text lesen und interpretieren (Parsing).



- **Parsing** benötigt etwa die Hälfte der Antwortzeit.
- Bekannte SQL-Befehle werden (von den meisten DBMS) wiedererkannt und müssen nicht erneut das Parsing durchlaufen (**Shared SQL**)
Allerdings müssen die Befehle **absolut identisch** sein!
- Für SQL-Befehle wird ein Hash-Wert errechnet und im **Library Cache** gespeichert.
- Nutzung durch **Bind Variables**, die anstelle von Werten im SQL-Text verwendet werden, verhindert ein erneutes Übersetzen.

Anweisungstuning



SQL ist deklarativ

⇒ mehrere syntaktische Konstrukte für dasselbe Ergebnis

Generell gilt: Vermeidung/Reduzierung von Plattenzugriffen

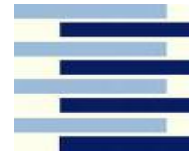
Plattenzugriff: 10 - 20 millisec

Zugriff im RAM: 1 - 2 microsec

Allgemeine Richtlinien:

1. Formulieren Sie gleiche SQL-Anweisungen in exakt gleicher Groß/Kleinschreibweise (SQL-Anweisungen in shared SQL area)
2. Nur Daten selektieren, die wirklich benötigt werden
3. Keine Ausdrücke über indizierte Spalten
4. Bei Performance Messungen caching-Effekt berücksichtigen! Bei der ersten Ausführung eines Befehls werden evtl. Daten über IO geladen. Ab dem zweiten Ausführen des selben Befehls befinden sich diese evtl. schon im Hauptspeicher).

Anfragetransformation

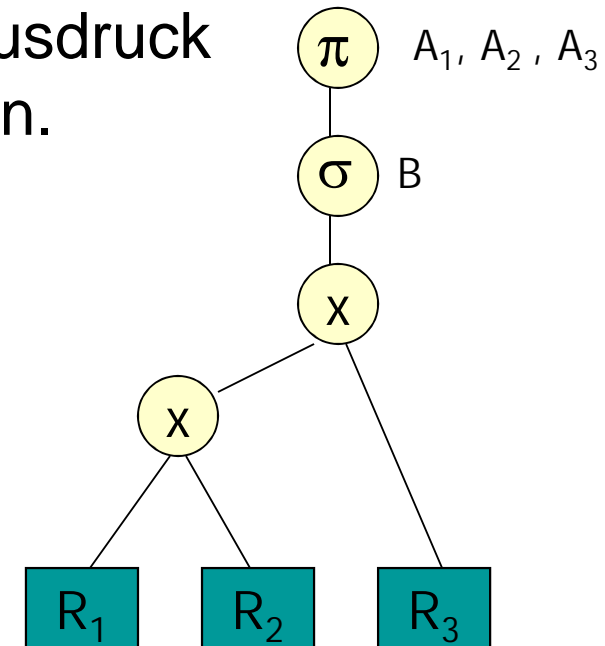


Vorgehen:

Umwandlung der deklarativen Anfrage („*was will ich*“) in einen relationenalgebraischen Ausdruck („*was muss ich tun*“) auf internen Dateien.

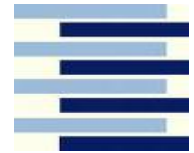
Grundmuster

SELECT A_1, A_2, \dots, A_n
FROM R_1, R_2, \dots, R_m
WHERE B



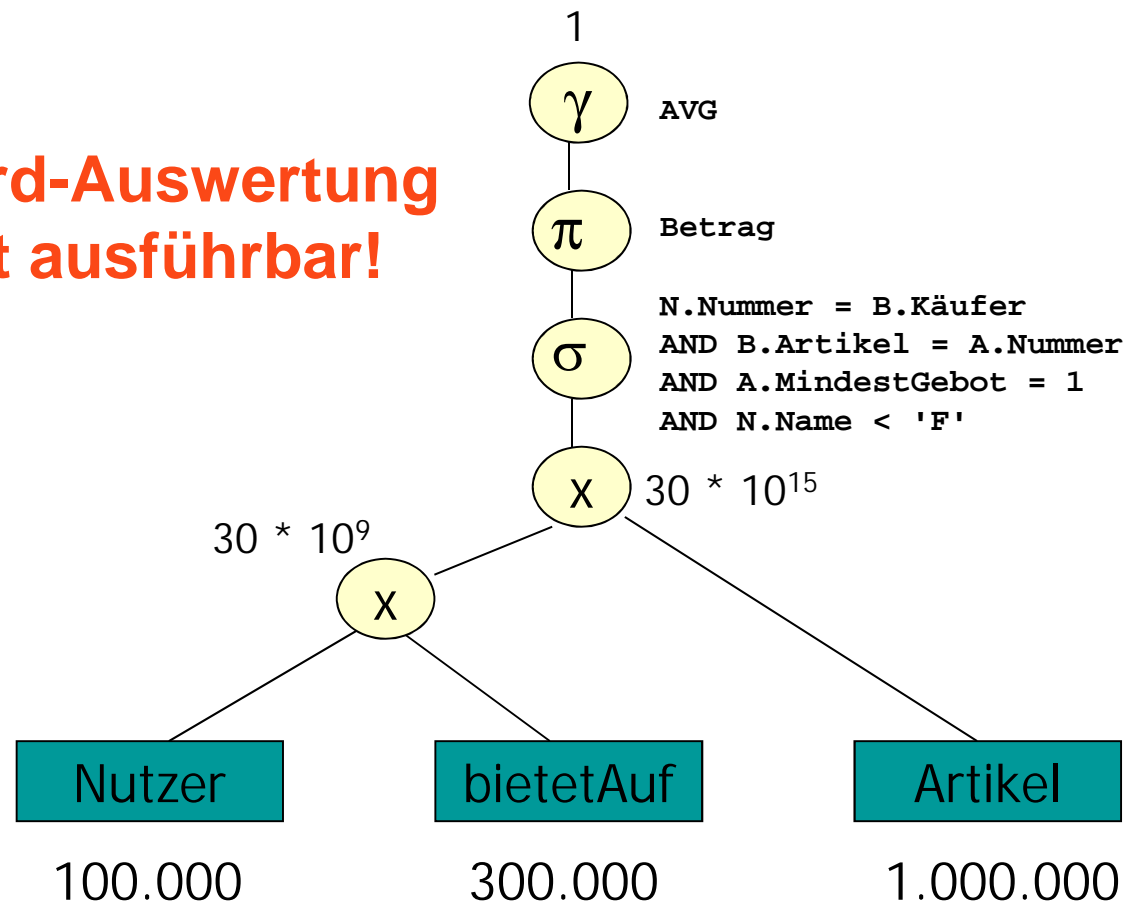
wird zu $f_{A_1, A_2, \dots, A_n} (\uparrow_B (R_1 \hat{\bowtie} R_2 \hat{\bowtie} \dots \hat{\bowtie} R_m))$

Beispiel



```
SELECT AVG(B.Betrag)
FROM Nutzer N, bietetAuf B, Artikel A
WHERE N.Nummer = B.Käufer AND B.Artikel = A.Nummer
AND A.MindestGebot = 1 AND N.Name < 'F'
```

**Standard-Auswertung
→ Nicht ausführbar!**

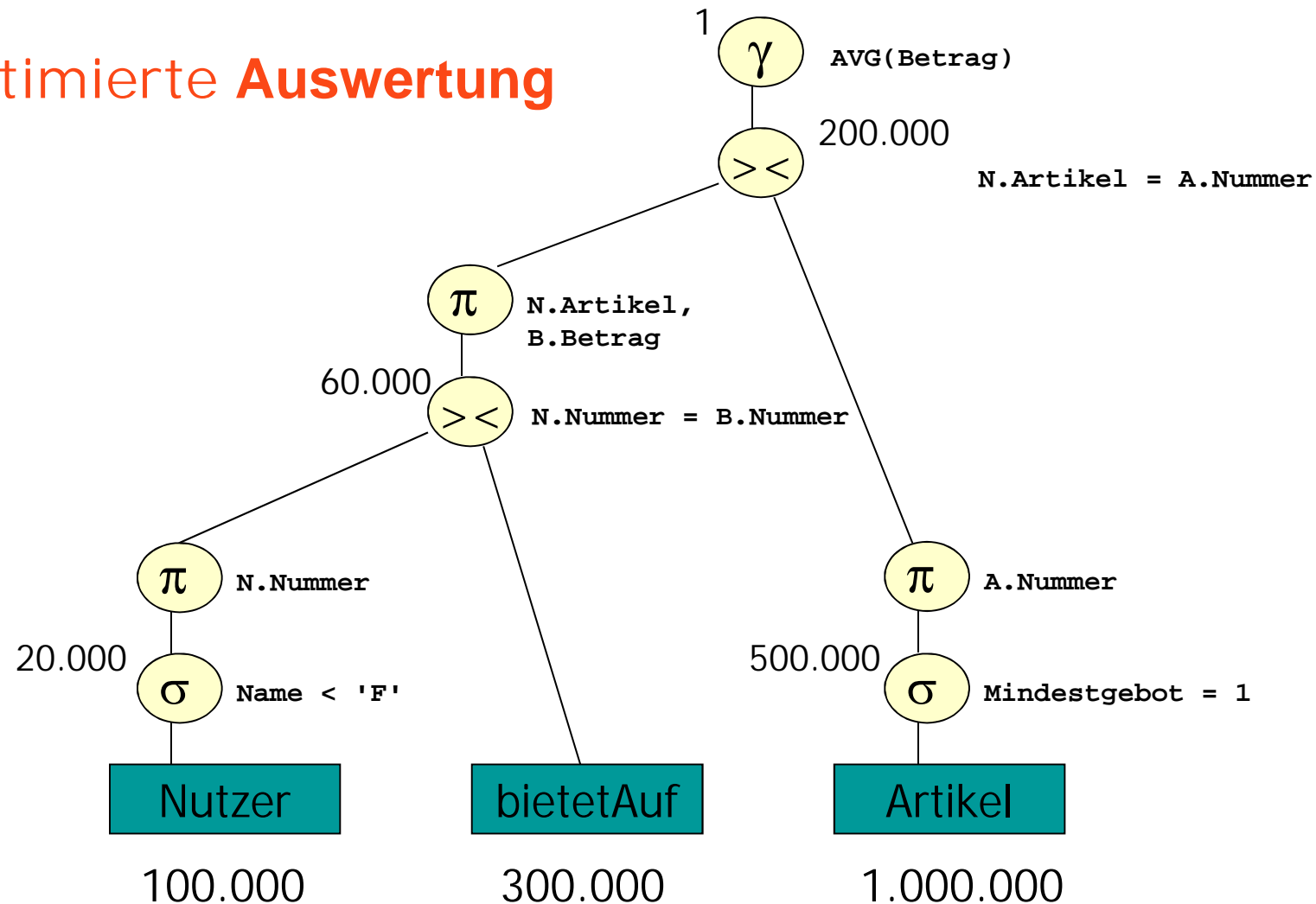


Bsp. nach M.
Klein, VL DB II

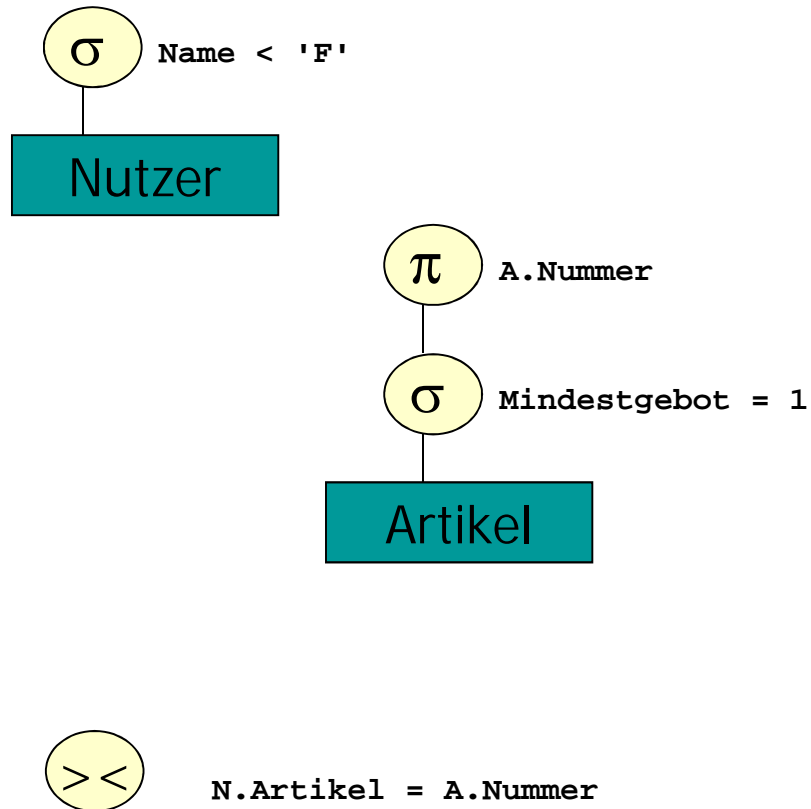
Ablauf: 1. Optimierung



Optimierte Auswertung



Ablauf: 2. Transformation

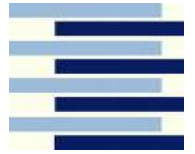


Als B*-Baum:
Durchlaufen der verketteten
Blätter von vorne bis „F“ erreicht.

Als sortierte Liste:
Ausgabe in nach Nummer
sortierter Reihenfolge



Als MergeJoin in linearer Zeit



Wie implementiert man die relationalen Operatoren?

Sortieren

Projektion

Selektion

Verbund (Join)

Kreuzprodukt



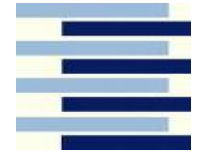
Für die folgenden
Folien Dank an ZKN

Operator „Sortieren“



- Sortieren ist eine der wichtigsten Algorithmenklassen der Informatik und sehr gut erforscht
- Bekannt sind sicherlich die klassischen Algorithmen für das Sortieren: Quicksort, Heapsort, (Bubblesort), usw.
- Diese Algorithmen haben gemeinsames Merkmal: Sie sortieren ihre Daten im Hauptspeicher
- Diese Algorithmen sind deshalb für das Sortieren in DBMS nicht einzusetzen, da dort die Daten nicht in den Hauptspeicher passen
- Deshalb neue Unterklasse von Algorithmen: **Externes Sortieren**
- Typische Idee für externe Sortieralgorithmen: Sortieren und Verschmelzen (sort and merge)
- Statt alle Daten in den Hauptspeicher zu laden und dort zu sortieren wird der Datenbestand in k Partitionen geteilt und diese k Partitionen werden getrennt in den Hauptspeicher (Puffer) geladen und dort mit einem der bekannten Algorithmen sortiert

Sortieren



- Nach der Sortierung der Partitionen existieren k sortierte Dateien
- Anschließend werden die sortierten Dateien solange verschmolzen, bis am Ende genau eine sortierte Datei entsteht.
- Der **Aufwand** für diese Algorithmenklasse liegt bei

$$2F * \log_{(M-1)} F$$

Hierbei ist

F die Größe der zu sortierenden Datei in Seiten und

M die Größe des bereitstehenden Puffers in Seiten

Dieser Aufwand ist somit - ähnlich dem des internen Sortierens - $O(n \log n)$

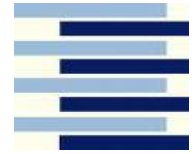
- Kommerzielle DBMS haben stark optimierte Algorithmen für diesen Hilfsoperator realisiert
- Der Hilfsoperator ist nicht nur für die Realisierung des SQL-Befehls „ORDER BY“ erforderlich, sondern wird auch für die Eliminierung von Duplikaten und die Berechnung des Verbunds eingesetzt

Projektion



- Die einfachste Realisierung der Projektion:
Durchlauf durch die zu projizierende Relation, und die nicht benötigten Attribute werden „weggeschmissen“
- **Problem:** Unsere Projektion soll auch ein „DISTINCT“ realisieren, d.h. doppelte Tupel eliminieren können
- **Deshalb neue Idee:**
Wir laufen durch die Relation, entfernen die nicht benötigten Attribute, schreiben die neue Relation raus, sortieren sie, und entfernen aus dieser sortierten Relation die doppelten Elemente
- **Aufwand:**
 1. Erster Durchlauf und rausschreiben: $2 * F$
 2. Sortieren: $2F * \log_{(M-1)} F$
 3. Doppelte Entfernen: $2 * F$
- **Optimierung:** Viel Arbeit beim Sortieren durchführen, dann ist Aufwand $2F * \log_{(M-1)} F$

Selektion



- Hier lediglich die einfachen Selektionsbedingungen (`.. WHERE x b y`)
 1. **Kein Index vorhanden:**

Durchlauf durch die gesamte Relation, Auswertung der Bedingung pro Tupel
 2. **B*-Baum-Index vorhanden:**

Nutzung des Index zum Auffinden des ersten passenden Attributs, dann sequentieller Durchlauf durch den Baum.
Idealfall: Clusterung, dann kommen die zusammen passenden Tupel gemeinsam in den Hauptspeicher. Ansonsten bis zu F Transfers erforderlich
 3. **Hash-Index vorhanden:**

Nutzung des Index zum Auffinden aller Attribute.
Idealfall: Clustering, siehe oben.
Keine Unterstützung für Bereichsanfragen
- Typisches DBMS realisiert mehrere Varianten, die es dynamisch auswählt.

Verbund



- Der Verbund ist aufwändiger als Projektion/Selektion:
 1. Die Größe des Ergebnisses kann deutlich über den Argumenten liegen (Selektion/Projektion: Höchstens Größe des Arguments)
 2. Der Aufwand kann quadratisch zum Argument sein (Selektion/Projektion: Höchstens Scan der Argumentrelation (linear))

Während quadratischer Aufwand für viele Algorithmen akzeptabel ist, sind solche Algorithmen für DBMS inakzeptabel (Beispiel: $1000 * 1000$ ergibt 1.000.000 Zugriffe, also bei 10 ms pro Zugriff ca. 10.000 Sekunden)
 3. Der Verbund wird sehr häufig ausgeführt (z.B. Primär-/Fremdschlüsselbeziehung) und der Benutzer erwartet eine effiziente Bearbeitung
- Bekannt sind drei grundsätzlich verschiedene Algorithmen zur Berechnung des Verbundes

Verbund: Nested loop



- **Grundidee:** Zur Berechnung des Verbunds **R Join S** mit $r.a = s.b$ lesen wir jedes Tupel von R hintereinander und vergleichen es mit jedem Tupel aus S
- **Pseudocode:**

```
foreach r in R do
    foreach s in S do
        if r.a = s.b then
            write (r,s)
        end if
    end foreach
end foreach
```
- Die Bestimmung des Aufwands für diesen Algorithmus nutzt folgende Bezeichner:
 - β_r und β_s bezeichnen die Anzahl der Blöcke von R und S
 - τ_r bezeichnet die Anzahl der Tupel in R

Aufwand für Nested loop



1. Für jedes Tupel in R werden alle Daten aus S gelesen, also $\tau_r * \beta_s$
 2. Zusätzlich muss die Relation R eingelesen werden, also β_r draufaddieren
 3. Aufaddiert: $\beta_r + \tau_r * \beta_s$ oder etwas abstrakter: $O(n*n)$
- **Beispiel:**
 $\beta_r = 1000$, $\beta_s = 100$ und $\tau_r = 10.000$ (also eine sehr kleine DB)
Ergebnis: 1.001.000 Seitenzugriffe, d.h. ca. 166 Minuten
 - **Wichtig:** Die Reihenfolge der Schleifen ist relevant!
Obiges Beispiel mit $\tau_s = 100$ und umgekehrter Schleife ergibt nur noch 100.100 Seitenzugriffe, d.h. keine 2 Minuten
 - **Generell:** Größere Relation in die innere Schleife
 - **Variation 1:** Kein Durchlauf pro Tupel, sondern pro Seite (**Block-nested-loop**), dadurch Aufwandreduktion auf $\beta_r + \beta_r * \beta_s$
 - **Variation 2:** Ausnutzen eines Index (Index-nested-loop), um statt des Scans in der inneren Schleife den Index zu nutzen. Sinnvoll mit clustered Indexen oder hoher Selektivität

Verbund: Sort-Merge-Join



- **Grundidee:** Zunächst jede der beiden Argument-Relationen bezüglich des Verbund-Attributs sortieren und anschließend die beiden Argumente durch einen sequentiellen Durchlauf verschmelzen
- **Pseudocode:**

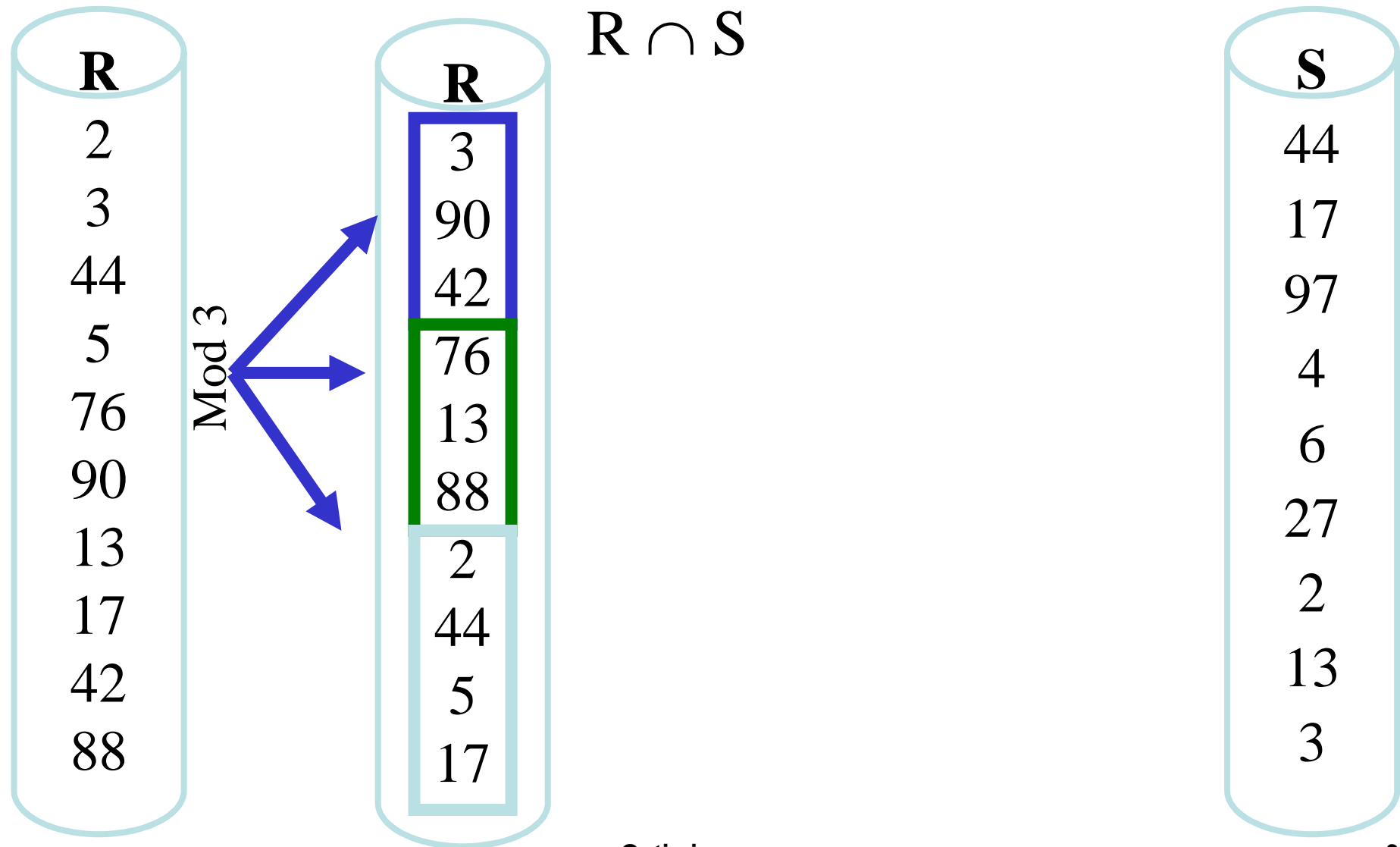
```
sort r on a
sort s on b
r := getFirst(r)
s := getFirst(s)
while not(EOF(r)) and not(EOF(s)) do
    while not(EOF(r)) and (r.a < s.b) do
        r := getNext(r)
    while not(EOF(s)) and (r.a > s.b) do
        s := getNext(s)
    if r.a = s.b then write(r,s)
    end if
```
- **Aufwand** wird dominiert von den Sortieroperationen, d.h. $O(n)$ oder $O(n \log n)$ wenn noch sortiert werden muss.

Verbund: Hash-Join

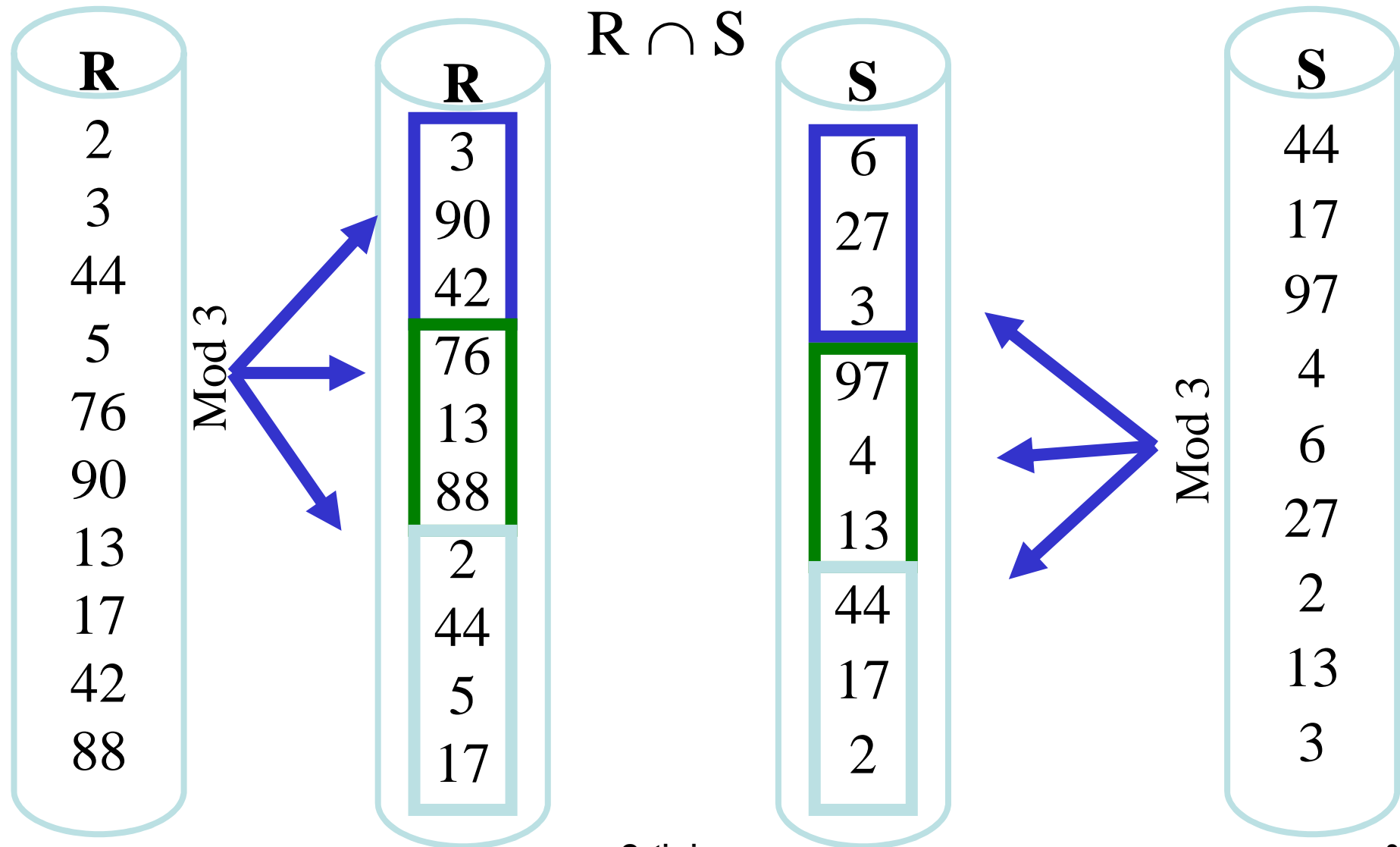
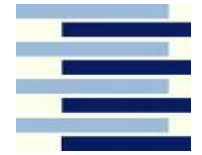


- **Grundidee:** Wir hashen beide Argumentrelationen mit Hilfe der gleichen Hashfunktion in Buckets. Gleiche Tupel auf dem Verbundattribut werden in gleichen Buckets landen und werden anschließend nach einem erneuten Durchlauf durch die Buckets verschmolzen.
- **Aufwand:** r und s jeweils einlesen, in Buckets ausschreiben und anschließend gemeinsam verschmelzen, d.h. $3 * r$ und $3 * s$, somit $O(n)$
- Damit ist Hash-Join der **Algorithmus der Wahl** für die Berechnung des Verbunds
- **Einschränkung:** Funktioniert nur bei einem Äqui-Join, d.h. wenn die Join-Bedingung die Gleichheit zweier Attribute ist
- Insbesondere die "Nicht-Gleichheit" und "größer/kleiner" als Verbundbedingungen disqualifizieren einen Verbund für die Realisierung durch den Hash-Join-Algorithmus
- Nutzung häufig für die Berechnung von Zwischenergebnissen.

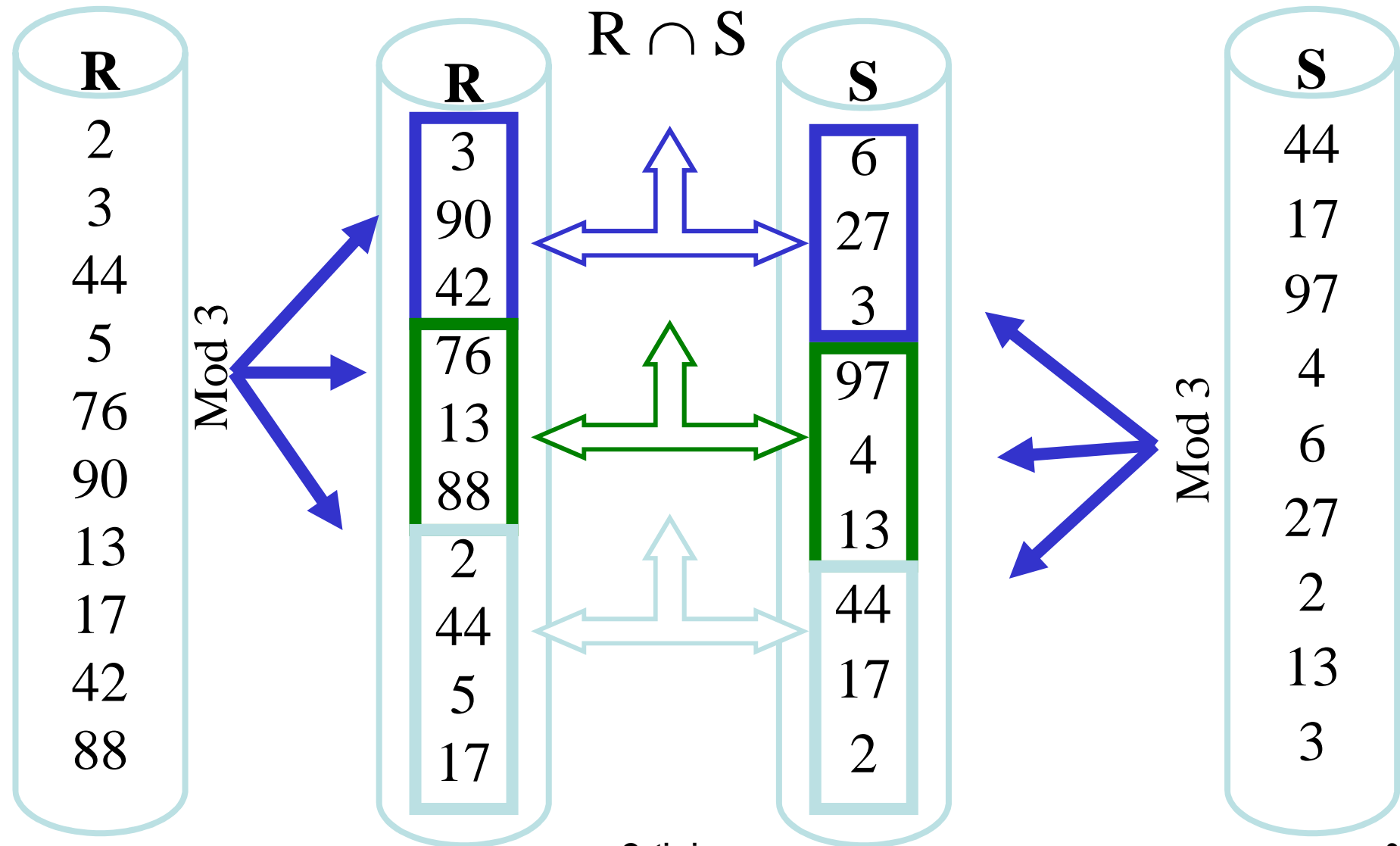
Beispiel Hash-Join



Beispiel Hash-Join



Beispiel Hash-Join



Optimierung



Für die Ausführung der übersetzten SQL-Befehle erstellt das DBMS einen Ausführungsplan, wobei vom DBMS Optimierungsregeln eingesetzt werden.

Regelbasierte Optimierung

Die SQL-Befehle werden nach festen Regeln analysiert und die Reihenfolge der Operationen danach festgelegt.

Aufwandbezogene Optimierung

Hierbei werden interne Statistiken und Strukturmerkmale der Datenbank (z.B. Indizes) analysiert um den schnellsten Weg zu den gesuchten Daten zu finden.

Manuelle Optimierung

Durch einen Kommentar im SELECT Befehl kann ein Ausführungshinweis an den Optimierer gegeben werden.

```
SELECT /*+ FULL (...) */ name, . FROM ..
```

Über den Befehl EXPLAIN PLAN sind bei Oracle (nicht leicht zu lesende) Informationen über das tatsächlich gewählte Optimierungsverfahren zu erhalten.

Einführende Beispiele



gegeben:

ADRESSEN (NAME, LAND, PLZ, ORT, SEX) **ARTIKEL**(ARTNR, PREIS)

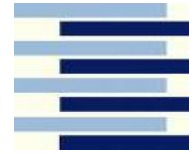
Indizes anlegen und diese in der Where-Klausel auch komplett angeben

```
CREATE INDEX ADR_LANDORT ON ADRESSEN (LAND, ORT)
```

```
SELECT NAME FROM ADRESSEN WHERE ORT='Hamburg'
```

```
SELECT NAME FROM ADRESSEN WHERE LAND='D' AND ORT='Hamburg'
```

Einführende Beispiele



gegeben:

KUNDEN (NAME, LAND, PLZ, ORT, SEX)

Verwendung optimaler Indizes erzwingen

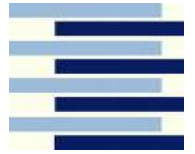
```
CREATE INDEX ADR_NAME ON KUNDEN(NAME)
```

```
CREATE INDEX ADR_SEX ON KUNDEN(SEX)
```

```
SELECT NAME FROM ADRESSEN WHERE NAME LIKE 'Mei%'  
                                AND SEX = 'M'
```

HINT: `SELECT /*+ INDEX(kunden name) */ * from kunden
where name like 'J%' and sex ='M'`

Einführende Beispiele



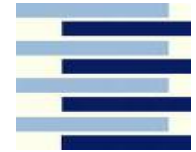
gegeben:

ADRESSEN (NAME, LAND, PLZ, ORT, SEX) **ARTIKEL**(ARTNR, PREIS)

Evtl. unnötige Spalten mitselektieren um Order-by zu beschleunigen

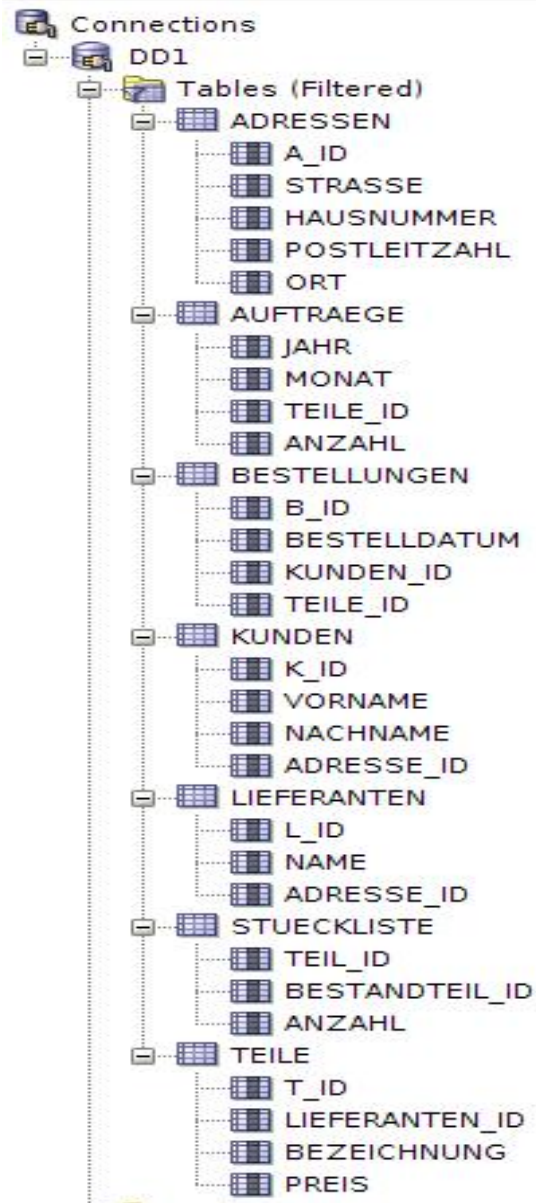
```
CREATE INDEX ARD_ORT ON ADRESSEN (ORT)
SELECT NAME FROM ADRESSEN ORDER BY ORT
SELECT NAME, ORT FROM ADRESSEN ORDER BY ORT
```


SQL-Tuning



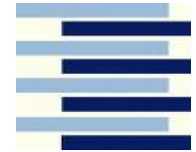
- Verwenden des kostenbasierten Optimierers mit aktuellen Analysedaten für alle Tabellen (Befehl "analyze table").
- bei Verwendung von "or" ggf. RULE einschalten oder "UNION" verwenden
- an häufig benutzten kleinen Tabellen den Parameter "cache" setzen => wird bei full table scans nicht aus cache entfernt (alter table <tabelle> cache;>)
- auf "vergessene" Indizes achten (insbesondere bei "einfachen" Queries)
- wenig selektive Indizes vermeiden
- bei zusammengesetzten Indizes: erste Spalte muss eingeschränkter sein
- keine Verwendung von Indizes bei: is null, <>, like '%abc'
- Vermeidung des NOT-Operators
- Benutzung von Joins anstelle Sub-Selects
- besser "exists"-Subquery als "in"-Subquery verwenden

Beispiel



Wie hoch ist der
Gesamtumsatz pro Kunde?


Query mit =



0.058 seconds DD1

Worksheet Query Builder

```
1 SELECT k_id, SUM(preis), nachname FROM
2 (SELECT k.vorname, k.nachname, k.k_id, t.preis FROM kunden k, teile t, bestellungen b
3 WHERE b.teile_id = t.t_id AND b.kunden_id = k.k_id) GROUP BY k_id, nachname
4
```

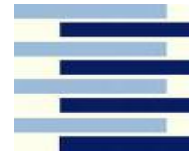


Script Output x Query Result x Explain Plan x

SQL | 0.058 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			8
HASH		GROUP BY	8
NESTED LOOPS			7
NESTED LOOPS			5
TABLE ACCESS	BESTELLUNGEN	FULL	3
TABLE ACCESS	KUNDEN	BY INDEX ROWID	1
INDEX	PK_KUNDEN	UNIQUE SCAN	0
Access Predicates		B.KUNDEN_ID=K	
INDEX	PK_TEILE	UNIQUE SCAN	0
Access Predicates		B.TEILE_ID=T.T_ID	
TABLE ACCESS	TEILE	BY INDEX ROWID	1


Query mit like



0.053 seconds

Worksheet Query Builder

```
1 SELECT k_id, SUM(preis), nachname FROM
2 (SELECT k.vorname, k.nachname, k.k_id, t.preis FROM kunden k, teile t, bestellungen b
3 WHERE b.teile_id LIKE t.t_id AND b.kunden_id LIKE k.k_id) GROUP BY k_id, nachname
4
```



Script Output x Query Result x Explain Plan x

SQL | 0.053 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			11
HASH		GROUP BY	11
NESTED LOOPS			10
NESTED LOOPS			7
TABLE ACCESS	BESTELLUNGEN	FULL	3
TABLE ACCESS	KUNDEN	FULL	2
Filter Predicates			
TO_CHAR(B.KUNDEN_ID			
TABLE ACCESS	TEILE	FULL	3
Filter Predicates			
TO_CHAR(B.TEILE_ID) LIKE			

Optimierung bei Oracle: Explain plan



ORACLE bietet die Möglichkeit, sich mit **EXPLAIN PLAN** alle Schritte bei der Ausführung eines SQL-Statements anzeigen zu lassen. Damit wird gezeigt, welche Strategie der ORACLE-Optimizer zur Abarbeitung eines SELECT, INSERT usw. wählt. Literatur: ORACLE RDBMS , Performance Tuning Guide

Einige mögliche Operationen:

MERGE JOIN Die Ergebnismenge wird durch die Verbindung zweier sortierter Mengen gebildet

NESTED LOOPS Die Ergebnismenge wird durch die Verbindung zweier Unterabfragen gebildet. Jede Rückgabezeile der ersten Unteroperation bringt auch die zweite zur Ausführung.

TABLE ACCESS FULL Zugriff auf die gesamte Tabelle

TABLE ACCESS BY ROW ID Zugriff über ROW ID, nicht seq.

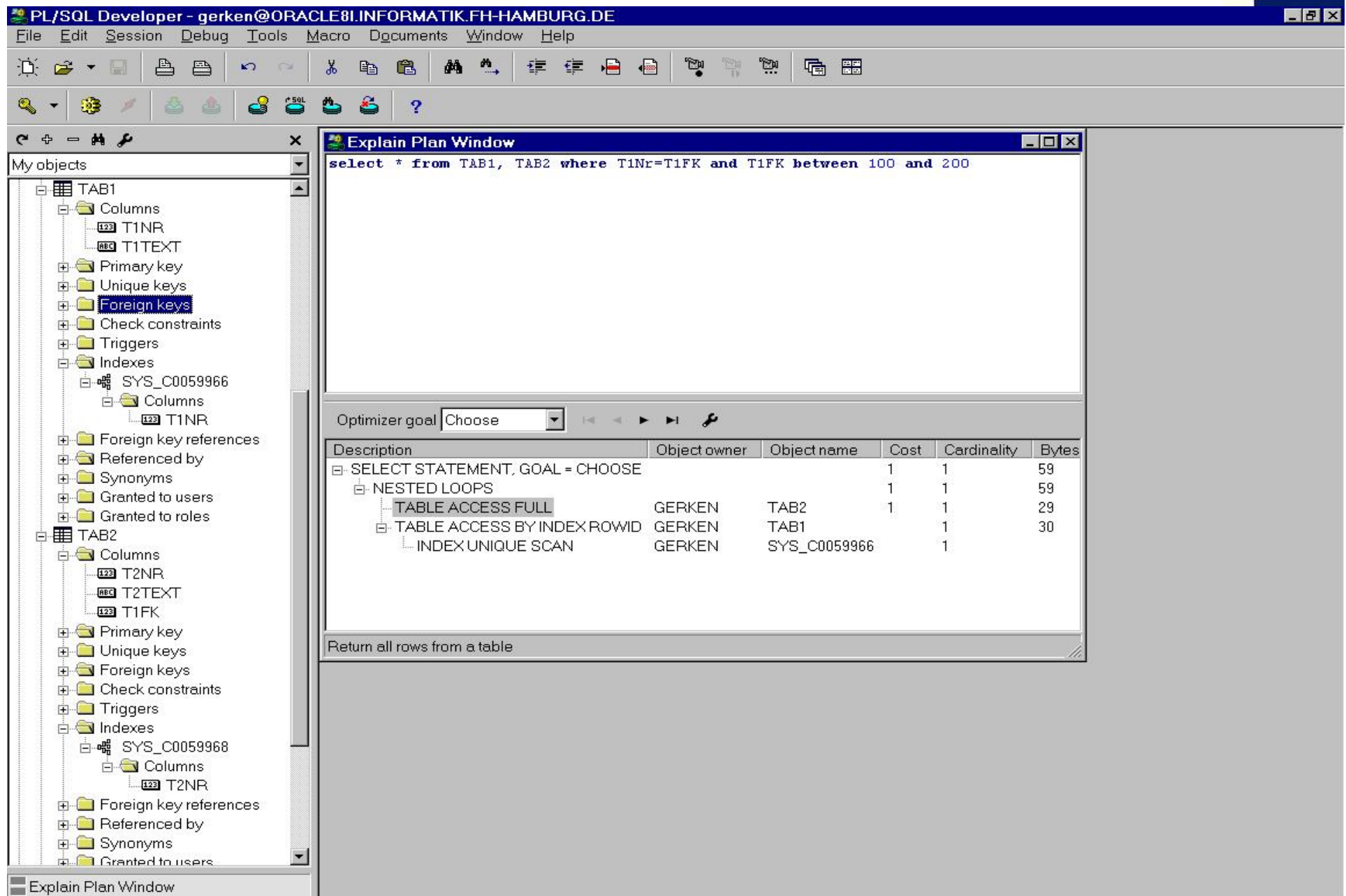
INDEX Indexsuche (UNIQUE oder über einen Bereich)

SORT JOIN Sortierung für einen MERGE JOIN

FILTER Restriktion auf die Zeilen, die von einer Tabelle zurückgegeben werden.

PROJECTION Rückgabe einer Untermenge von Spalten

Explain plan im PL/SQL-Developer



PL/SQL Developer - gerken@ORACLE8I.INFORMATIK.FH-HAMBURG.DE

File Edit Session Debug Tools Macro Documents Window Help

My objects

- TAB1
 - Columns
 - T1NR
 - T1TEXT
 - Primary key
 - Unique keys
 - Foreign keys
 - Check constraints
 - Triggers
 - Indexes
 - SYS_C0059966
 - Columns
 - T1NR
 - Foreign key references
 - Referenced by
 - Synonyms
 - Granted to users
 - Granted to roles
- TAB2
 - Columns
 - T2NR
 - T2TEXT
 - T1FK
 - Primary key
 - Unique keys
 - Foreign keys
 - Check constraints
 - Triggers
 - Indexes
 - SYS_C0059968
 - Columns
 - T2NR
 - Foreign key references
 - Referenced by
 - Synonyms
 - Granted to users

Explain Plan Window

select * from TAB1, TAB2 where T1Nr=T1FK and T1FK between 100 and 200

Optimizer goal Choose

Description	Object owner	Object name	Cost	Cardinality	Bytes
SELECT STATEMENT, GOAL = CHOOSE			1	1	59
NESTED LOOPS			1	1	59
TABLE ACCESS FULL	GERKEN	TAB2	1	1	29
TABLE ACCESS BY INDEX ROWID	GERKEN	TAB1		1	30
INDEX UNIQUE SCAN	GERKEN	SYS_C0059966		1	

Return all rows from a table

Beispiele



Vgl. die Tabelle Adressen(Name, Land, PLZ, Ort, Sex) Index auf Name, Sex

explain plan for

select Name from Adressen where Name like 'Ge%' and Sex = 'M';

<u>OPERATION</u>	<u>OPTIONS</u>	<u>OBJECT_NAME</u>
SELECT STATEMENT		
TABLE ACCESS	BY INDEX ROWID	ADRESSEN
INDEX	RANGE SCAN	ADR_SEX

explain plan for **select Name from Adressen where Name like 'Ge%' and Sex || '' = 'M';**

<u>OPERATION</u>	<u>OPTIONS</u>	<u>OBJECT_NAME</u>
SELECT STATEMENT		
TABLE ACCESS	BY INDEX ROWID	ADRESSEN
INDEX	RANGE SCAN	ADR_NAME

Beispiele



```
create table KONDITIONEN (PARTNER_ID number not null,  
                           TEIL_ID      number not null,  
                           LISTENPREIS number(7,2) null,  
constraint FK_KOND_LIEF foreign key (PARTNER_ID) references LIEFERANTEN,  
constraint FK_KOND_TEILE foreign key (TEIL_ID) references TEILE ,  
constraint PK_KONDITIONEN primary key (PARTNER_ID, TEIL_ID)      );
```

explain plan for (select * from KONDITIONEN where PARTNER_ID=101);

<u>OPERATION</u>	<u>OPTIONS</u>	<u>OBJECT_NAME</u>
SELECT STATEMENT		
TABLE ACCESS	BY INDEX ROWID	KONDITIONEN
INDEX	RANGE SCAN	PK_KONDITIONEN

explain plan for (select * from KONDITIONEN where TEIL_ID=1);

<u>OPERATION</u>	<u>OPTIONS</u>	<u>OBJECT_NAME</u>
SELECT STATEMENT		
TABLE ACCESS	BY INDEX ROWID	KONDITIONEN
INDEX	SCIP SCAN	PK_KONDITIONEN

Beispiel



Gegeben seien die Relationen Rechnung(RgNr, ArtNr, KdNr, ...)
 Artikel(ArtNr, Bezeichnung, Preis, ...)
 Kunde(KdNr, Name, ...)

SELECT Rechnung.RgNr, Bezeichnung, Preis FROM Artikel, Rechnung
WHERE Artikel.ArtNr = Rechnung.ArtNr AND Rechnung.KdNR =
 (SELECT KdNr FROM Kunde WHERE Name='Horn')

FILTER

MERGE JOIN

SORT JOIN

TABLE ACCESS FULL Rechnung

SORT JOIN

TABLE ACCESS FULL Artikel

TABLE ACCESS FULL KUNDE

Ausführungsgraph



FILTER

MERGE JOIN

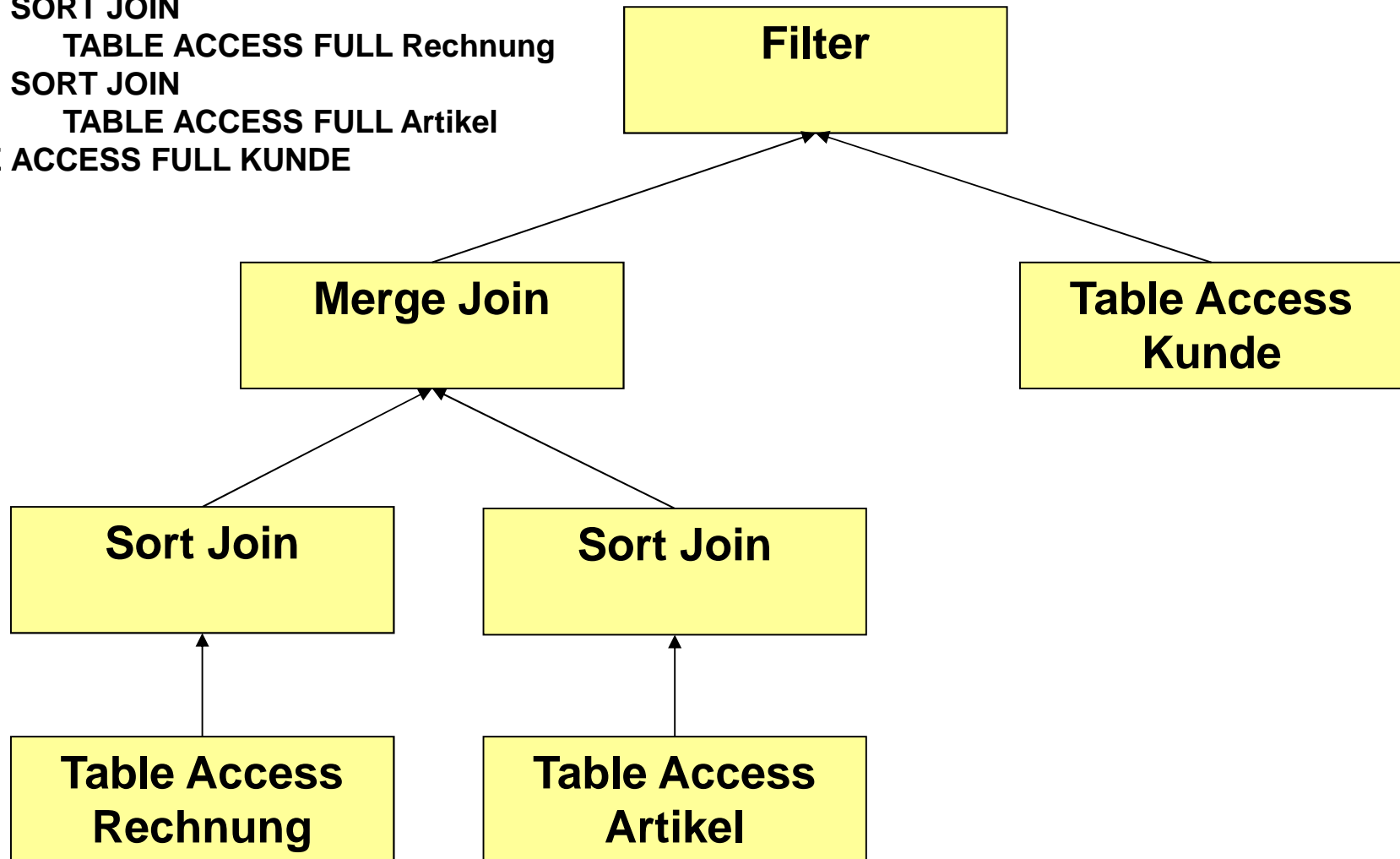
SORT JOIN

TABLE ACCESS FULL Rechnung

SORT JOIN

TABLE ACCESS FULL Artikel

TABLE ACCESS FULL KUNDE



Zur Vertiefung



1. Weitere Beispiele in meinem PUB-Verzeichnis
Optimierung_von_Sql-Befehlen.pdf (Studienarbeit)
2. Mit PL/SQL-Developer ausprobieren
3. Literatur