

Datenbanken und SQL

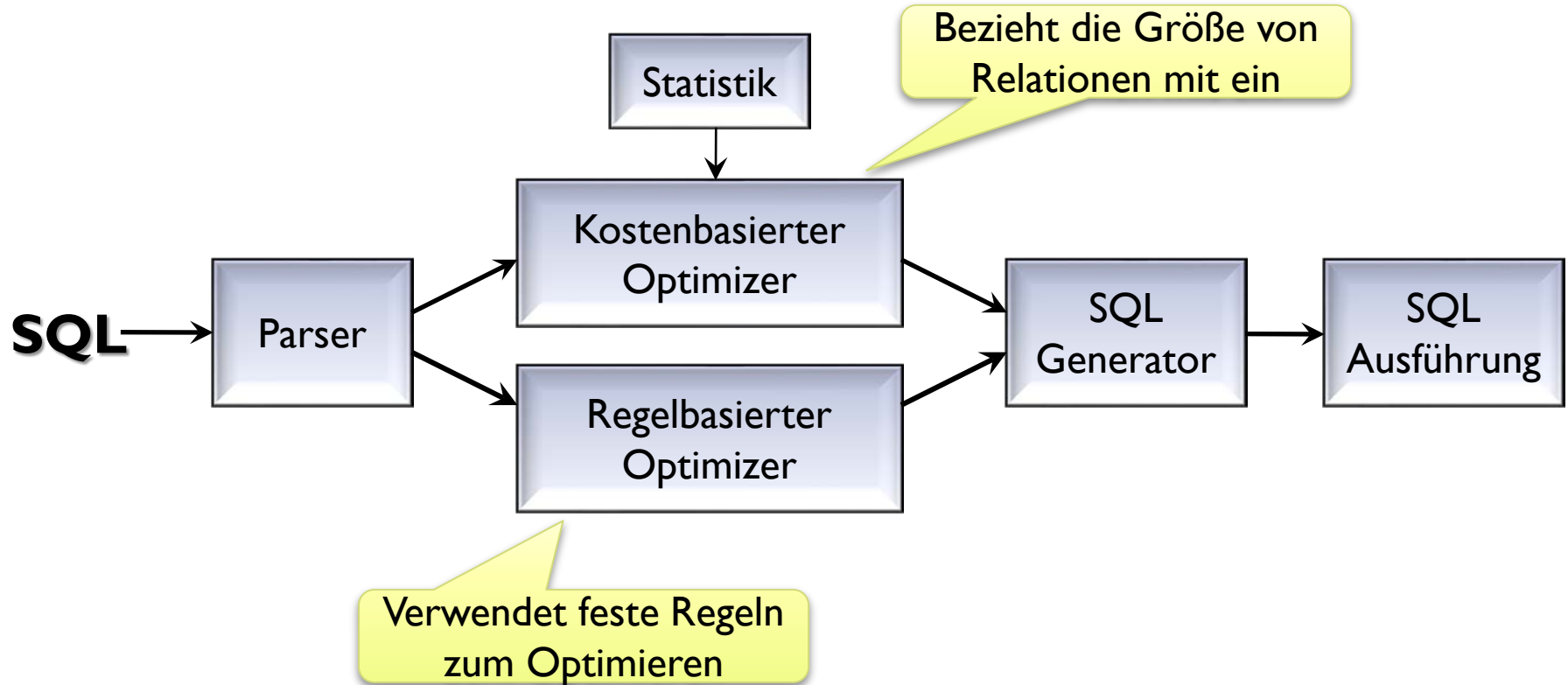
Kapitel 7

Performance in Datenbanken

Performance in Datenbanken

- ▶ **Datenbankenoptimizer und Ausführungsplan**
- ▶ **Index**
- ▶ **Partitionierung**
- ▶ **Materialisierte Sichten**
- ▶ **Optimierungen im Select-Befehl**
- ▶ **Stored Procedures**
- ▶ **Weitere Optimierungen**

Abfrageoptimierung



Optimizer

▶ Regelbasierte Optimierung

- ▶ Optimierung nach vorgegebenen Regeln. Beispiele:
 - ▶ Verwende einen Index, falls vorhanden
 - ▶ Führe erst Projektion durch, dann einen Verbund
 - ▶ Führe erst Restriktion durch, dann einen Verbund
 - ▶ Verwende Merge Join, wenn Relationen bereits sortiert sind

▶ Kostenbasierte Optimierung

- ▶ Optimierung zusätzlich in Abhängigkeit von den Kosten
- ▶ Berücksichtigung der Größe (und des Inhalt) der Relationen
 - ▶ Beispiel: Verwende im Verbund erst die kleinere Relation

Beispiel zur Optimierung

```
Select Auftrnr, Artnr, Anzahl, Gesamtpreis  
From Auftrag Natural Inner Join Auftragsposten  
Where Kundnr = 3;
```

Im Befehl: Erst der Join,
dann die Restriktion

Explain-Plan x

SQL | 0,015 Sekunden

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
HASH JOIN		
Zugriffsprädikate AUFTRAG.AUFTRNR=AUFTRAGSPOSTEN.AUFTRNR		
TABLE ACCESS	AUFTRAG	FULL
Filterprädikate AUFTRAG.KUNDNR=3		
TABLE ACCESS	AUFTRAGSPOSTEN	FULL

Tatsächliche Ausführung: Erst
die Restriktion, dann der Join

Regelbasiert versus kostenbasiert

▶ Regelbasierter Optimizer

- + Relativ einfach implementierbar
- Ungenau, da nicht an die reale Tabelle angepasst

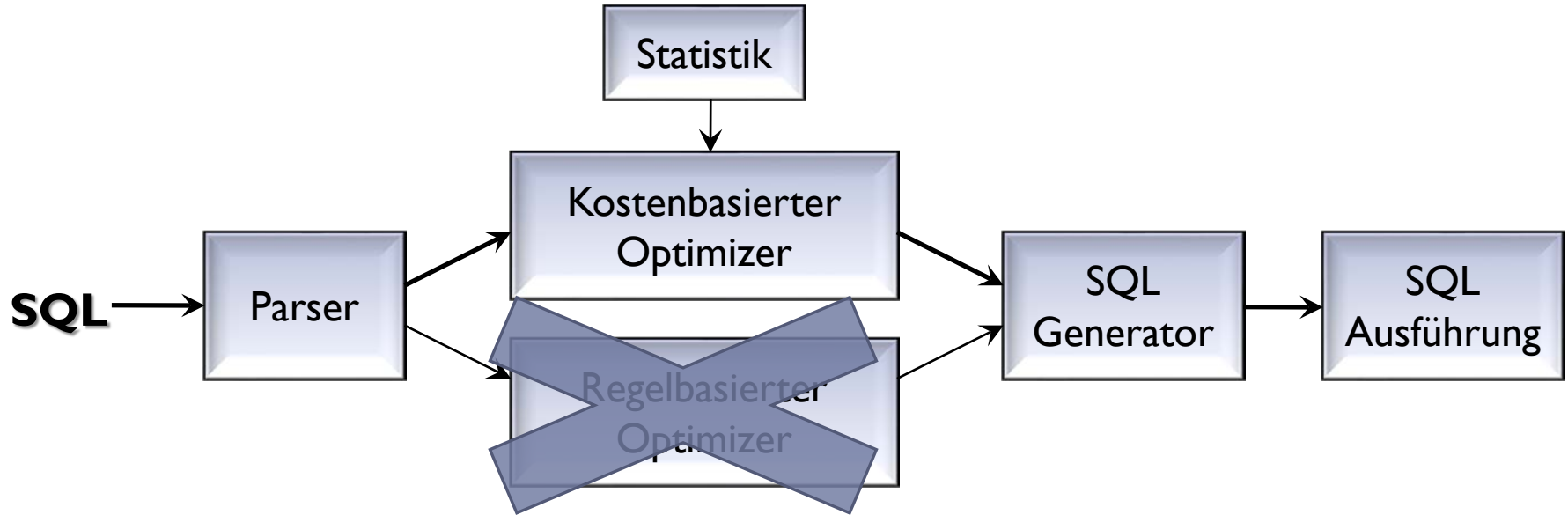
▶ Kostenbasierter Optimizer

- + Berücksichtigt den Aufbau einer Tabelle
- + Liefert deshalb sehr genaue Ergebnisse
- Benötigt zusätzliche Statistiken
- Relativ komplex und umfangreich

▶ Fazit:

- ▶ Kostenbasierter Optimizer ist der Standard

Abfrageoptimierung



Kostenbasierter Optimizer: Aufgaben

- ▶ Analyse des Select-Befehls
- ▶ Überprüfen der Umgebung nach
 - ▶ vorhandenen Indexen
 - ▶ Aufteilung von Relationen in Partitionen
 - ▶ internen Strukturen (z.B. Aufteilung auf Festplatten)
- ▶ Auswertung der Statistiken
- ▶ Optimierung des Select-Befehls
 - ▶ unter Verwendung der obigen Gesichtspunkte
- ▶ Weiterleitung an den SQL-Generator

Statistiken

- ▶ Jede Datenbank sammelt umfangreiche Statistiken
- ▶ Notwendig für den kostenbasierten Optimizer
- ▶ Wichtige Statistiken:
 - ▶ Anzahl der Zeilen jeder Relation
 - ▶ Anzahl unterschiedlicher Einträge je Attribut und Relation
 - ▶ Zusätzlich: Histogramme zu jedem Attribut
- ▶ In der Praxis wichtig:
 - ▶ Selektivität eines Attributs → nächste Folie

Selektivität

▶ Definition (Selektivität eines Attributs):

Die Selektivität eines Attributs **sel** ist der Kehrwert zur Anzahl der unterschiedlichen Attributswerte.

▶ Beispiele:

▶ Primärschlüssel: **sel** = $1 / \text{Anzahl der Zeilen}$

▶ Geschlecht (m/f): **sel** = 0,5

▶ Wochentage: **sel** = 0,14

▶ Die Selektivität wird benötigt:

▶ bei Restriktionen: zur Abschätzung der Größe der Restrelation

▶ ebenso bei Gruppierungen

Selektivität (2)

- ▶ Die Selektivität **sel** ist ein Mittelwert
 - ▶ Sind die Attributswerte ungleich verteilt, so sollten zusätzlich Histogramme verwendet werden
- ▶ Die Selektivität mehrerer Attribute kann einfach aus den Einzelattributen berechnet werden. Es gilt:

$$\text{sel}(\text{Attr1 AND Attr2}) = \text{sel}(\text{Attr1}) * \text{sel}(\text{Attr2})$$

$$\text{sel}(\text{Attr1 OR Attr2}) = \text{sel}(\text{Attr1}) + \text{sel}(\text{Attr2}) - \text{sel}(\text{Attr1}) * \text{sel}(\text{Attr2})$$

$$\text{sel}(\text{NOT Attr1}) = 1 - \text{sel}(\text{Attr1})$$

- ▶ Diese Gleichungen gelten exakt nur bei Gleichverteilungen

Optimierung in Oracle

Systemtabellen	Wichtige Attribute
USER_TABLES	Table_Name, Num_Rows, Avg_Row_Len
USER_TAB_STATISTICS	Table_Name, Num_Rows, Avg_Row_Len
USER_TAB_COLUMNS	Table_Name, Column_Name, Num_Distinct, Density, Num_Nulls, Avg_Col_Len, Histogram
USER_TAB_COL_STATISTICS	Table_Name, Column_Name, Num_Distinct, Density, Num_Nulls, Avg_Col_Len, Histogram

Anzahl Zeilen

entspricht
Selektivität

für manche Anfragen
sehr wichtig

Anzahl unterschiedliche
Einträge

Statistiken in Oracle

- ▶ Automatische Aktualisierung der Statistiken mit:

DBMS_AUTO_TASK_ADMIN.ENABLE

Administratorrechte!

- ▶ Manuelle Aktualisierung der Statistiken:

- ▶ Statistikpaket DBMS_STATS

- ▶ Beispiele:

Aktualisierung
einzelner Tabellen

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS( 'Schema', 'Tabelle' );
```

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS( 'Schema' );
```

Aktualisierung im
gesamten Schema

Achtung!
Performance!

Ausführungsplan in Oracle

Arbeitsblatt Query Builder

```
select nr, name, anr, bezeichnung
from lieferant inner join lieferung on nr=liefnr
natural inner join artikel;
```

Join von drei Relationen

Skriptaussgabe x Abfrageergebnis x Explain-Plan x

SQL | 0 Sekunden

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			9
SORT		ORDER BY	9
HASH JOIN			8
Access Predicates			
LIEFERUNG.ANR=ARTIKEL.ANR			
MERGE JOIN			4
TABLE ACCESS	LIEFERANT	BY INDEX ROWID	2
INDEX	PK_LIEFERANT	FULL SCAN	1
SORT		JOIN	2
Access Predicates			
LIEFNR=NR			
Filter Predicates			
LIEFNR=NR			
INDEX	PK_LIEFERUNG	FULL SCAN	1
TABLE ACCESS	ARTIKEL	FULL	3

... dann Hash Join mit Artikel

Nur Index-Zugriff

Merge Join zwischen Lieferant und Lieferung ...

Nur Index-Zugriff

Vollzugriff auf Artikel

Optimierung in SQL Server

- ▶ SQL Server erzeugt zu jedem Schlüssel einen Index
- ▶ SQL Server führt Statistiken zu allen Indexen
- ▶ Ansehen von Statistiken:
 - ▶ `DBCC SHOW_STATISTICS( Relation, Index );`
 - ▶ oder komfortabel mit SQL Server Management Studio
- ▶ Statistiken anlegen, aktualisieren:

10% der Daten sind
Basis für Hochrechnung

```
CREATE STATISTICS Statistikname ON Tabelle (Spalte) WITH FULLSCAN;  
CREATE STATISTICS Statistikname ON Tabelle (Spalte) WITH SAMPLE 10 PERCENT;  
ALTER DATABASE Datenbank SET AUTO_CREATE_STATISTICS ON;  
ALTER DATABASE Datenbank SET AUTO_UPDATE_STATISTICS ON;
```

Automatisches
Erzeugen bzw. Ändern

Beispiel einer Statistik (im SSMS)

Tabellenname:

Statistikname:

Statistiken für INDEX 'PK_Personal'.

Name	Updated
PK_Personal	Feb 5 2013 9:00PM

= 1/9, da 9 Einträge
(Selektivität)

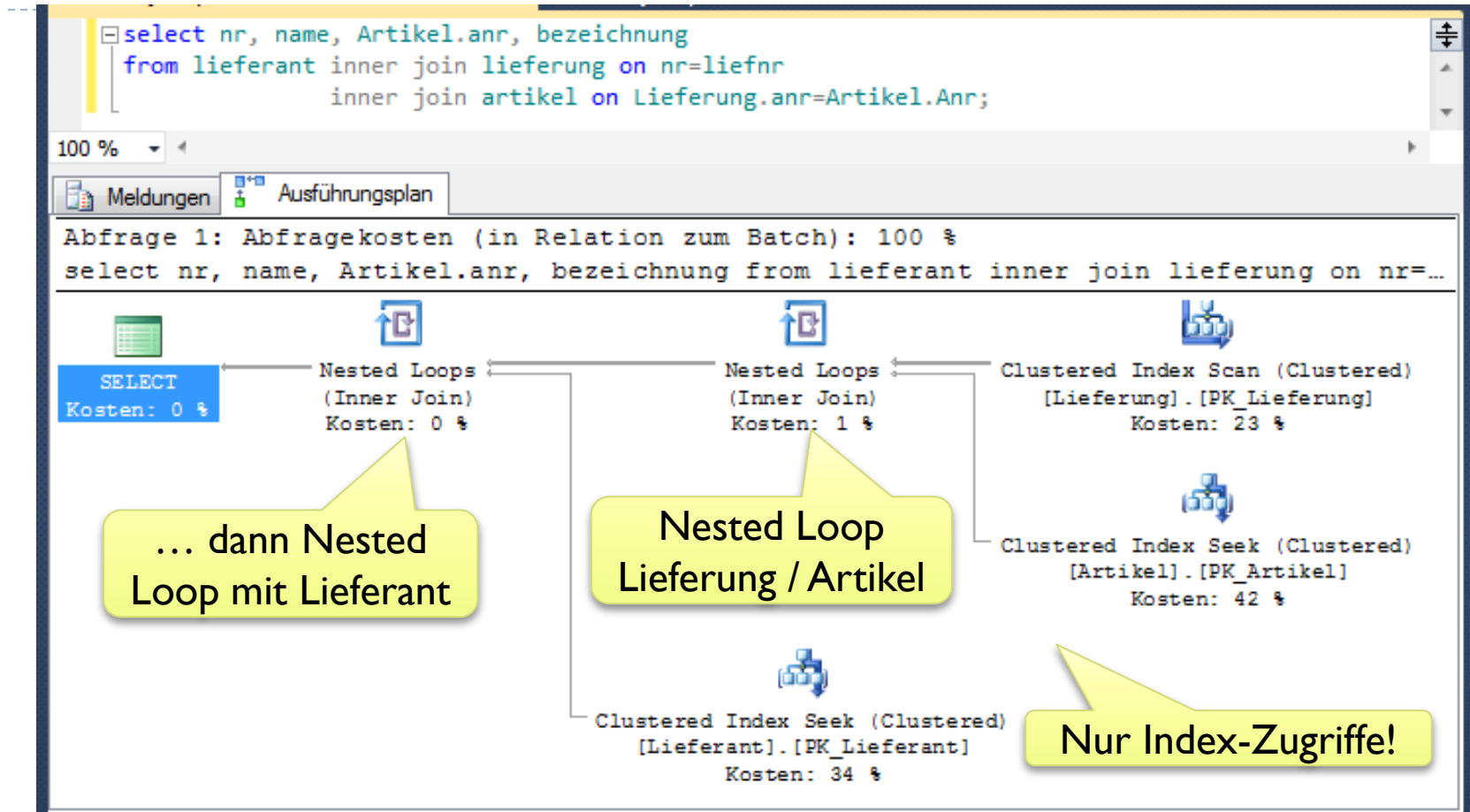
All Density	Average Length
0.1111111	4

Histogram Steps

RANGE_HI_KEY	RANGE_ROWS
1	0
3	1
5	1
7	1
9	1

Einfaches Histogramm

Ausführungsplan in SQL Server



Optimierung in MySQL

- ▶ Statistiken werden automatisch angelegt für
 - ▶ Primärschlüssel
 - ▶ Alternative Schlüssel
 - ▶ Fremdschlüssel
- ▶ Statistiken stehen in:
 - ▶ `Information_Schema.Statistics`
- ▶ Statistiken enthalten keine Selektivität und keine Histogramme

Index

- ▶ **Definition (Index):**

Ein Index in einer Datenbank ist eine Struktur zur Beschleunigung von Suchvorgängen, in der Daten auf- oder absteigend sortiert angelegt werden.

- ▶ Ein Index in einer relationalen Datenbank wird in der Regel intern als eine eigenständige sortierte Tabelle angelegt, auf den mittels eines B*-Baums zugegriffen wird.

Index in SQL-1

Falls eindeutig

CREATE [UNIQUE] INDEX Name ON

Index über mehrere
Spalten möglich

Tabellenname ({ Spalte [ASC | DESC] } [, ...])

aufsteigend
(Standard)

absteigend

DROP INDEX Name

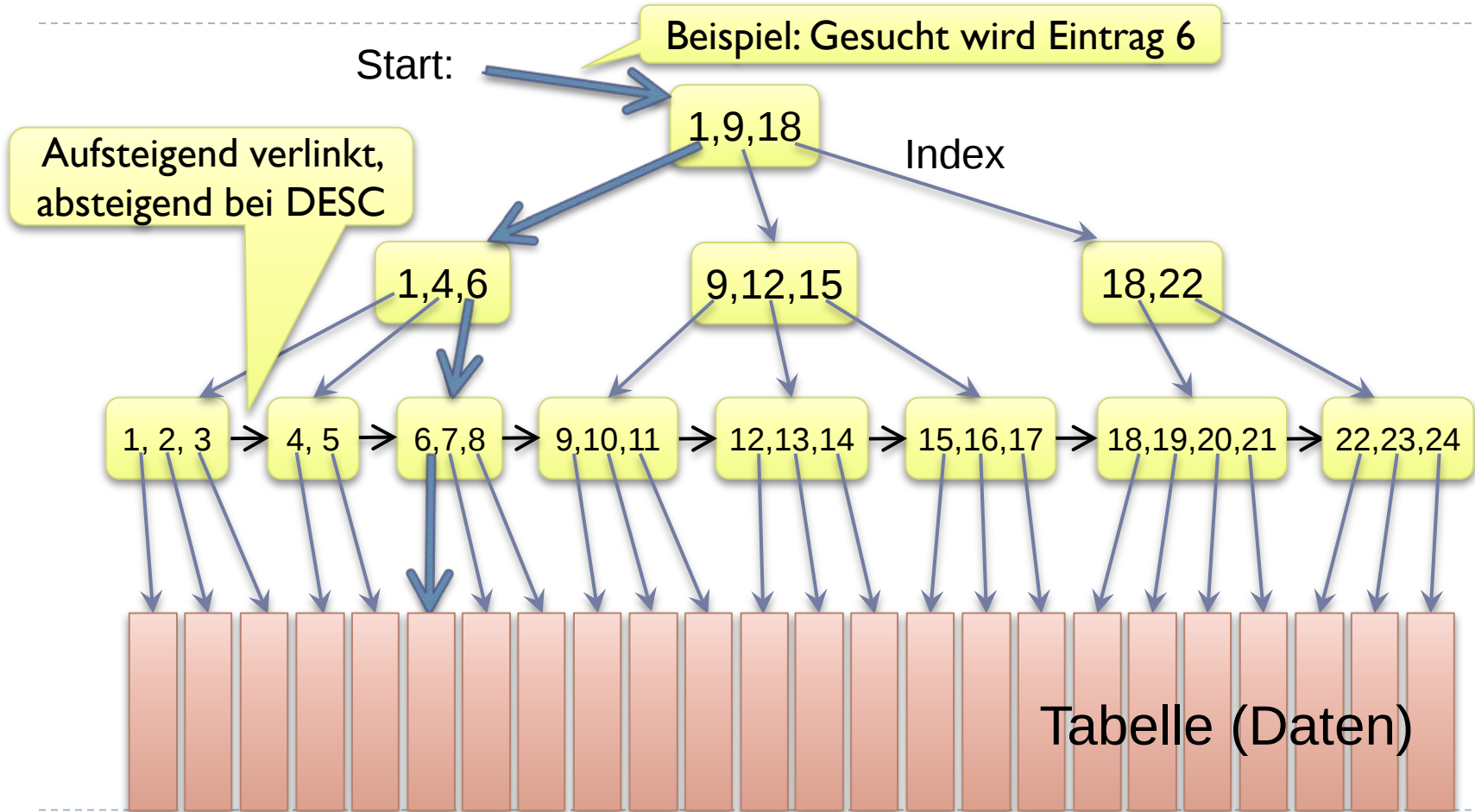
Seit SQL-2 nicht mehr normiert, aber immer noch üblich!

Systemtabellen und Indexe

Hersteller	Systemtabelle / Befehl	Inhalt
Oracle	USER_INDEXES	alle Indexe der Benutzerrelationen
Oracle	USER_IND_COLUMNS	alle Attribute mit gesetzten Indexen
SQL Server	SYS.INDEXES	alle Indexe
MySQL	SHOW INDEX FROM Tabelle	alle Indexe der Tabelle

Achtung:
nicht in INFORMATION_SCHEMA,
da Indexe nicht normiert!

Realisierung von Indexen: B*-Baum



Beispiel zu Indexen: Relation Umfrage

Name	Wert
NUM_ROWS	800000
BLOCKS	6409
AVG_ROW_LEN	51
SAMPLE_SIZE	800000
LAST_ANALYZED	08.06.13
LAST_ANALYZED_SINCE	08.06.13

800000 Einträge

Spaltenstatistik



Aktualisieren: 0

Selektivität

OWNER	TABLE_NAME	COLUMN_NAME	NUM_DISTINCT	DENSITY	NUM_NULLS
BIKE	UMFRAGE	NR	800000	0,00000125	0
BIKE	UMFRAGE	GESCHLECHT	2	0,5	0
BIKE	UMFRAGE	FAMILIENSTAND	4	0,25	160000
BIKE	UMFRAGE	GEBURTJSJAHR	50	0,02	0
BIKE	UMFRAGE	AUTOMARKE	1000	0,001	0
BIKE	UMFRAGE	WOHNORT	49548	0,0000201824493...	0

50000 Orte

Zugriff auf Relation Umfrage

```
Select * From Umfrage  
Where Wohnort = 'wohnort22222';
```

Zugriff auf wohnort22222

Explain-Plan x

SQL | 0 Sekunden

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			1752
TABLE ACCESS	UMFRAGE	FULL	1752
Filter Predicates			
WOHNORT='wohnort22222'			

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			20
TABLE ACCESS	UMFRAGE	BY INDEX R...	20
INDEX	IUMFRAGE...	RANGE SCAN	3
Access Predicates			
WOHNORT='wohnort22222'			

Ohne Index:
Kosten 1752

Mit Index:
Kosten 20

Range Scan des Index

Um Faktor 88
schneller!

Ergebnis

- ▶ Das Setzen und Entfernen des Index erfolgte mit:

```
CREATE INDEX IUmfrageWohnort ON Umfrage(Wohnort);  
DROP INDEX IUmfrageWohnort;
```

- ▶ Das Erzeugen eines Index erfordert etwas Zeit und benötigt Speicherplatz
- ▶ Der lesende Zugriff wird erheblich beschleunigt
- ▶ Der schreibende Zugriff wird langsamer, da Index immer mit aktualisiert werden muss
- ▶ Folgerung: So viele Indexe wie nötig, so wenige wie möglich

Partitionierung

- ▶ **Nachteil sehr großer Relationen**
 - ▶ Keine Parallelzugriffe, da nur ein Medium
 - ▶ Bei Suche ohne Index: Komplette Relation durchsuchen
- ▶ **Zerlegen in viele kleine Teilrelationen**
 - ▶ **Vorteil:**
 - ▶ Obige Nachteile fallen weg
 - ▶ **Nachteil:**
 - ▶ Anwender muss Strukturen kennen
 - ▶ Neue Teile sind den Anwendungsprogrammen nicht bekannt, daher inflexibel
- ▶ **Lösung: Partitionen**

Definition: Partitionierung

▶ Definition:

- ▶ Unter einer Partitionierung verstehen wir eine horizontale, vollständige und transparente Aufteilung einer Relation in disjunkte Teilrelationen.
- ▶ Diese Teilrelationen bezeichnen wir als Partitionen.

▶ Begriffe:

- ▶ Horizontal: zeilenweise (nicht spaltenweise)
- ▶ Vollständig: 
- ▶ Transparent: Nicht sichtbar für den Anwender
- ▶ Disjunkt: Keine redundante Aufteilung

Partitionierung am Beispiel

Gegeben: Relation Produktion,
partitioniert nach Monaten



Zugriff



Produktion

Zugriff wird
durchgereicht



Prod_Jan

Prod_Feb

...

Prod_Dez

Anwender kennt nur
die Relation Produktion

In Wirklichkeit:
Monatspartitionen

Unterstützung der Partitionierung

Hersteller	Partitionierungstypen
Oracle	Bereichs-Partitionierung List-Partitionierung Hash-Partitionierung Intervall-Partitionierung Referenz-Partitionierung Virtuelle spaltenbasierte Partitionierung
SQL Server	Bereichs-Partitionierung Index-Partitionierung
MySQL	Bereichs-Partitionierung List-Partitionierung Hash-Partitionierung Schlüssel-Partitionierung

Bereichspartitionierung am Beispiel

► Bereichspartitionierung (Range-Partitioning) in Oracle:

CREATE TABLE Auftrag (Nach außen sichtbar
Auftrnr INT PRIMARY KEY,
Datum DATE NOT NULL,
Kundnr INT NOT NULL REFERENCES Kunde,
Persnr INT REFERENCES Personal) Relation Auftrag wird nach Jahren partitioniert
PARTITION BY RANGE (Datum)
(PARTITION Auftrag2022 VALUES LESS THAN DATE '2023-01-01',
PARTITION Auftrag2023 VALUES LESS THAN DATE '2024-01-01',
PARTITION Auftrag2024 VALUES LESS THAN DATE '2025-01-01',
PARTITION Auftrag2025 VALUES LESS THAN DATE '2026-01-01');

Partitionierungsarten (1)

▶ Bereichspartitionierung:

- ▶ Einteilung in disjunkte Bereiche (meist nach Datum/Zeit)
- ▶ Sehr häufig eingesetzt
- ▶ In allen Datenbanken implementiert

▶ List-Partitionierung

z.B. eine Partition für Deutschland, eine Partition für Österreich und Schweiz gemeinsam usw.

- ▶ Einteilung nach Listen
- ▶ Beispiel: Einteilung nach Verkaufsländern (Liste aller Länder)

▶ Hash-Partitionierung

- ▶ Datenbank übernimmt die Einteilung nach Hash-Codes
- ▶ Nur interessant, wenn es sonst keine sinnvolle Einteilung gibt

Partitionierungsarten (2)

▶ Intervall-Partitionierung

- ▶ Spezielle Bereichspartitionierung
- ▶ Partitionierungsintervalle werden vorgegeben
- ▶ Beispiel: Monatsintervall → Jeden Monat wird automatisch neue Partition erzeugt

▶ Virtuelle spaltenbasierte Partitionierung

- ▶ In Oracle gibt es virtuelle Spalten (aus realen abgeleitet)
- ▶ Partitionierung mit Hilfe dieser virtuellen Spalten

▶ Index-Partitionierung

- ▶ In SQL Server: Index kann mit Relation partitioniert werden

Partitionierungsarten (3)

- ▶ **Schlüsselpartitionierung**
 - ▶ Spezielle Hashpartitionierung
 - ▶ Primär- oder alternativer Schlüssel dienen als Hash-Code
- ▶ **Referenzpartitionierung → nächste Folie**
- ▶ **Viele Kombinationen sind möglich**
 - ▶ Range – List (erst Rangepartitionierung, dann Listpart.)
 - ▶ Range – Hash
 - ▶ List – Hash usw.

Referenzpartitionierung

- ▶ Auftrag ist nach Jahren partitioniert (siehe oben)
- ▶ Auftragsposten enthält kein Datum, soll aber ebenso aufgeteilt werden → Referenzpartitionierung

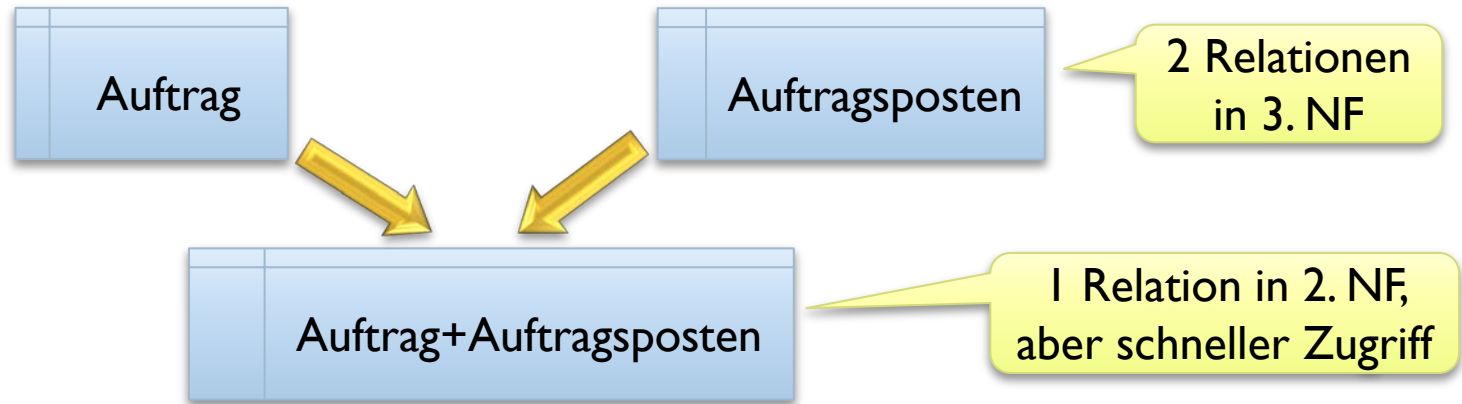
```
CREATE TABLE Auftragsposten (  
    Posnr          INT      PRIMARY KEY  
    Auftrnr        INT      NOT NULL,  
    Artnr          INT      NOT NULL,  
    Gesamtpreis    NUMERIC(8,2),  
    Anzahl         INT,  
    CONSTRAINT Auftrpos_FK FOREIGN KEY(Auftrnr)  
                                     REFERENCES Auftrag    )  
PARTITION BY REFERENCE( Auftrpos_FK )    ;
```

Partitionierung wie Auftrag!

Partitionierung: Zusammenfassung

- ▶ Partitionierung bei sehr großen Relationen wichtig
- ▶ Vorteile:
 - ▶ Transparenz: nicht sichtbar für Anwender
 - ▶ Parallele Verarbeitung möglich
 - ▶ Gezielter Zugriff auf nur eine oder wenige Partitionen statt auf gesamte Relation
- ▶ Folgerung:
 - ▶ Teils erhebliche Performancesteigerung

Materialisierte Sichten (Problem)



- ▶ Zusammenfassung von Auftrag und Auftragsposten?
 - ▶ Kein Join → Schnellere Zugriffe → Bessere Performance
 - ▶ Redundanz → Mehr Speicher → Gefahr von Inkonsistenz

Was tun?

Relation AuftragKomplett

Auftrag JOIN Auftragsposten

Posnr	Auftrnr	Datum	Kundnr	Persnr	Artnr	Anzahl	Gesamtpreis
101	1	04.01.2025	1	2	200002	2	800,00
201	2	06.01.2025	3	5	100002	3	1.950,00
202	2	06.01.2025	3	5	200001	1	400,00
301	3	07.01.2025	4	2	100001	1	700,00
302	3	07.01.2025	4	2	500002	2	100,00
401	4	18.01.2025	6	5	100001	1	700,00
402	4	18.01.2025	6	5	500001	4	30,00
403	4	18.01.2025	6	5	500008	1	94,00
501	5	03.02.2025	1	2	500010	1	40,00
502	5	03.02.2025	1	2	500013	1	30,00

Materialisierte Sichten (Idee)

- ▶ Sicht anlegen (Join zwischen Auftrag und Auftragsposten)
- ▶ Sicht physisch speichern!
- ▶ Vorteile:
 - ▶ Schnelle Zugriffe über (physische) Sicht
 - ▶ 3. NF der Basisrelationen bleibt erhalten
- ▶ Nachteile:
 - ▶ Es gibt jetzt Basisrelationen und (physische) Sicht parallel
 - ▶ Also: Redundanz und Gefahr der Inkonsistenz
- ▶ Wunsch:
 - ▶ Datenbank muss Datenabgleich intern übernehmen

Materialisierte Sicht AuftragKomplett

CREATE MATERIALIZED VIEW AuftragKomplett

REFRESH FAST ON COMMIT

Analog zu
Create View

AS SELECT Posnr, AuftrNr, Datum, Kundnr, Persnr,
Artnr, Anzahl, Gesamtpreis

FROM Auftrag NATURAL INNER JOIN Auftragsposten ;

► Refresh Fast / Refresh Complete [On Commit]

Nur Änderungen
nachvollziehen

Alternativ:
Inhalt komplett
neu erzeugen

Beim Commit
aktualisieren

Alternative: Zu bestimmten Zeitpunkten
aktualisieren: START NEXT ...

Materialisierte Sichten: Resümee

- ▶ Bei selten geänderten Relationen hervorragend geeignet
- ▶ Reduziert aufwändige Joins
- ▶ Allerdings:
 - ▶ Hoher interner Aufwand der Aktualisierung
 - ▶ Bei Refresh Fast wird ein Logbuch benötigt:
CREATE MATERIALIZED VIEW LOG ...
- ▶ Entfernen einer materialisierten Sicht:
DROP MATERIALIZED VIEW MV_Name

Optimierung von Select-Befehlen

- ▶ **Optimizer arbeitet nicht immer optimal**
- ▶ **Beispiel:**
 - ▶ Frauenverband mit wenigen männlichen Mitgliedern
 - ▶ Suche aller männlichen Mitglieder
 - ▶ **Optimizer:**
 - ▶ Selektivität für Geschlecht ist 0,5
 - ▶ Index lohnt nicht, da sowieso jedes 2. Mitglied gesucht wird
 - ▶ **Realität:**
 - ▶ Suche der wenigen männlichen Mitglieder über Index wäre effektiv
- ▶ **Folgerung:**
 - ▶ Wir beschäftigen uns ein wenig mit Optimierung

Vorteil des Optimizers

► Optimizer kennt alle Regeln der Relationalen Algebra!

► Wichtig sind insbesondere:

Vertauschung von
Restriktion und Verbund

$$(1) \quad \sigma_{\text{Bedingung}}(R1 \bowtie R2) = \sigma_{\text{Bedingung}}(R1) \bowtie \sigma_{\text{Bedingung}}(R2)$$

Spezialfall von (1)

$$(2) \quad \sigma_{\text{Bedingung_an_R2}}(R1 \bowtie R2) = R1 \bowtie \sigma_{\text{Bedingung_an_R2}}(R2)$$

Vertauschung von Projektion und Restriktion

$$(3) \quad \pi_{\text{Auswahl}}(\sigma_{\text{Bedingung}}(R)) = \sigma_{\text{Bedingung}}(\pi_{\text{Auswahl}}(R))$$

Vertauschung von Projektion und Verbund

$$(4) \quad \pi_{\text{Auswahl}}(R1 \bowtie R2) = \pi_{\text{Auswahl}}(R1) \bowtie \pi_{\text{Auswahl}}(R2)$$

Achtung: Projektion muss
verbindende Attribute enthalten!

Wichtige Regeln (1)

- ▶ Ein Verbund (und ein Produkt) zweier Relationen ist aufwändig
- ▶ Ziel: Beide Relationen vorher verkleinern
- ▶ Regel:
 - ▶ Erst Restriktion und Projektion
 - ▶ Dann Gruppierung
 - ▶ Und dann erst Verbund (Produkt)
- ▶ Der Optimizer kennt diese Regeln

Restriktion ist besonders effektiv, da dann meist weniger Daten eingelesen werden müssen

Beispiel zur Optimierung (Wiederholung)

```
Select Auftrnr, Artnr, Anzahl, Gesamtpreis  
From Auftrag Natural Inner Join Auftragsposten  
Where Kundnr = 3;
```

Im Befehl: Erst der Join,
dann die Restriktion

Explain-Plan x

SQL | 0,015 Sekunden

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
HASH JOIN		
Zugriffsprädikate AUFTRAG.AUFTRNR=AUFTRAGSPOSTEN.AUFTRNR		
TABLE ACCESS	AUFTRAG	FULL
Filterprädikate AUFTRAG.KUNDNR=3		
TABLE ACCESS	AUFTRAGSPOSTEN	FULL

Tatsächliche Ausführung: Erst
die Restriktion, dann der Join

Sicht VAuftrag (Wiederholung)

AuftrNr	Datum	Kundname	Persname	Summe
1	04.01.2025	Fahrrad Shop	Anna Kraus	800
2	06.01.2025	Maier Ingrid	Johanna Köster	2350
3	07.01.2025	Rafa – Seger KG	Anna Kraus	800
4	18.01.2025	Fahrräder Hammerl	Johanna Köster	824
5	06.02.2025	Fahrrad Shop	Anna Kraus	70

► Verbund aus Auftrag, Kunde, Personal und Auftragsposten

Beispiel zu den Grenzen des Optimizers

▶ Select-Befehl zur Sicht VAuftrag:

```
SELECT  AuftrNr, Datum, Kunde.Name, Personal.Name,  
        SUM (Gesamtpreis)  
FROM    Auftrag JOIN Kunde ON Kunde.Nr = Auftrag.Kundnr  
        JOIN Personal USING (Persnr)  
        JOIN Auftragsposten USING (Auftrnr)  
GROUP BY Auftrnr, Datum, Kunde.Name, Personal.Name ;
```

▶ Schwäche:

- ▶ Group By kommt nach JOIN
- ▶ Group By bezieht sich aber nur auf Auftragsposten

Woher soll das der
Optimizer wissen?

Ausführungsplan

Zusätzlicher Hash
für Group By

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			13
HASH		GROUP BY	13
HASH JOIN			12
Access Predicates			
AUFTRAG.PERSNR=PERSONAL.PERSNR			
HASH JOIN			9
Access Predicates			
AUFTRAG.AUFRNR=ITEM_1			
MERGE JOIN			6
TABLE ACCESS	KUNDE	BY INDEX ROWID	2
INDEX	PK_KUNDE	FULL SCAN	1
SORT		JOIN	4
Access Predicates			
AUFTRAG.KUNDNR=KUNDE.NR			
Filter Predicates			
AUFTRAG.KUNDNR=KUNDE.NR			
TABLE ACCESS	AUFTRAG	FULL	3
VIEW	VW_GRP_13		2
HASH			2
TABLE ACCESS	AUFTRAGSPOSTEN	BY INDEX ROWID	2
INDEX	AK_AUFTRAGSPOSTEN	FULL SCAN	1
TABLE ACCESS	PERSONAL	FULL	3

Hash Join mit Personal

Hash Join mit Auftragsposten

Merge Join zwischen
Kunde und Auftrag

Index!

Sortieren wegen Group
By und Merge Join

Full Scan!

Hash mit Auftragsposten

Index!

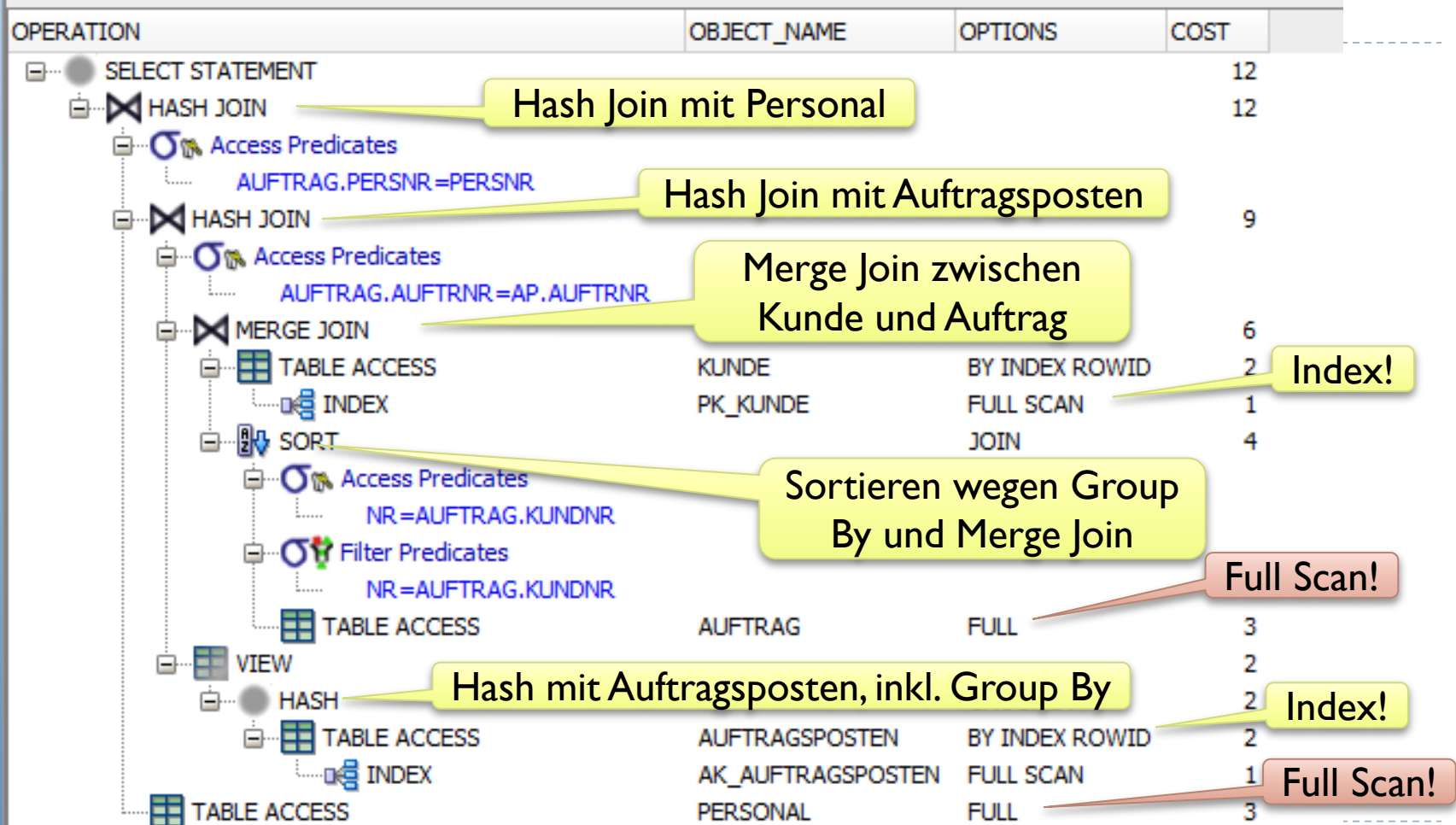
Full Scan!

Gruppierung vor Verbund

► Der Select-Befehl lautet jetzt:

```
SELECT  AuftrNr, Datum, K.Name, P.Name, AP.Summe
FROM    Auftrag JOIN
        (Select Nr, Name From Kunde) K ON K.Nr = Auftrag.Kundnr
        JOIN (Select Persnr, Name From Personal) P USING (Persnr)
        JOIN (  Select Auftrnr, Sum(Gesamtpreis) As Summe
                From Auftragsposten
                Group By Auftrnr      ) AP      USING (Auftrnr)
```

Ausführungsplan nach Vertauschung



Wichtige Regeln (2)

- ▶ Sortieren ist in großen Relationen extrem aufwändig
- ▶ Sortieren steckt indirekt in vielen Operationen
- ▶ Wir versuchen daher einige Operationen zu vermeiden:

- ▶ Kein Union (stattdessen: Union All)

Union entfernt gleiche
Einträge, aufwändig!
Union All tut dies nicht!

- ▶ Order By nur für kleine Teilrelationen

Sortiert!
Alternative: Limit, Rownum

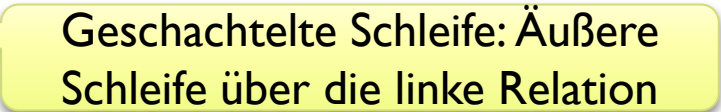
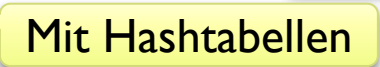
- ▶ Select Distinct nur für kleine Teilrelationen

Entfernt gleiche
Einträge, aufwändig!

- ▶ Group By, wenn möglich, vermeiden

Gruppieren heißt auch:
Gleiche Einträge suchen!

Der Verbund (JOIN)

- ▶ Der Verbund wird sehr oft benötigt
- ▶ Der Verbund ist aufwändig
- ▶ Regel:
 - ▶ Die kleinere Relation steht links vom Join-Operator
 - ▶ Eine vorherige Restriktion wird dabei berücksichtigt!
- ▶ Wichtige (interne) Verbund-Operationen:
 - ▶ Nested Loop Join 
 - ▶ Hash Join 
 - ▶ Merge Join 

Nested Loop Join ($R1 \bowtie R2$)

- ▶ **Schleife (grob):**

- foreach (linke Relation R1 as row1)

- foreach (rechte Relation R2 as row2)

- vergleiche row1 mit row2

- ▶ **Optimierung:**

- ▶ Vergleiche datenblockweise, nicht zeilenweise

- ▶ Dies optimiert die Einlesevorgänge

- ▶ **Aufwand:**

- ▶ R1 wird komplett eingelesen

- ▶ Ist Hauptspeicher knapp, so wird R2 bis zu $|R1|$ -mal gelesen

- ▶ In Summe: $|R1| + |R1| * |R2|$

($|R|$ = Anzahl der Blöcke von R)

Wenn R1 kleiner als R2, dann $|R1| < |R2|$

Also: kleinere Relation steht links!

Hash Join ($R1 \bowtie R2$)

- ▶ **Aufbau einer Hash-Tabelle zu $R1$ und $R2$**
- ▶ **Aufwand**
 - ▶ Lesen von $R1$ und $R2$ wegen Hash-Tabelle
 - ▶ Lesen von $R1$ und $R2$ wegen des Verbindens
 - ▶ Lesen von $R1$ und $R2$ wegen abschließenden Mischens
- ▶ **Bei extrem knappem Speicher gilt für den max.Aufwand:**
 $3 * (|R1| + |R2|)$ ($|R|$ = Anzahl der Blöcke von R)
- ▶ **Ohne Beweis:**
 - ▶ Bei knappem Speicher hat Hash große Vorteile, wenn die linke Relation kleiner ist!

Merge Join ($R1 \bowtie R2$)

► Idee:

- Sortieren von R1
- Sortieren von R2
- Zusammenmischen der beiden Relationen

► Aufwand:

- Das Sortieren erfordert $c \cdot n \cdot \log(n)$ Schritte, n =Anzahl, c =const

► Problem:

- Bei knappem Speicher können sortierte Relationen nicht im Arbeitsspeicher gehalten werden.
- Daher:Aufwand vergleichbar mit Hash und Nested Loop

Vergleich der drei Joins

- ▶ **Je nach Anwendung kann jeder Join der beste sein**
- ▶ **Grob gilt:**
 - ▶ Merge Join kommt mit wenig Arbeitsspeicher aus
 - ▶ Nested Loop Join benötigt viel Arbeitsspeicher, ist dann schnell
 - ▶ Hash Join ist optimal, wenn eine Relation relativ klein
- ▶ **Die kleinere Relation als erste Relation hat Vorteile**
 - ▶ bei Nested Loop Join
 - ▶ bei Hash Join

Index und Select-Befehl

- ▶ Index zu einem Attribut kann nur verwendet werden, wenn direkt dieses Attribut im Select verwendet wird!
- ▶ Negatives Beispiel:
 - ▶ Suche nach einem Namen, der indiziert ist
 - ▶ Verwendet wird: `... Where Trim(Name) Like ...`
 - ▶ Index kann nicht mehr verwendet werden, da auch führende Leerzeichen entfernt werden!
 - ▶ Hervorragende Alternative:
 - ▶ `... Where RTrim(Name) Like ...`

Das noch verbleibende rechtsseitige Trimmen stört den Index nicht

Index und Oracle

- ▶ Standard: Setzen von Indexen auf Attribute
- ▶ Oracle: Zusätzlich Setzen von Indexen auf Ausdrücke!
- ▶ Beispiel:

```
CREATE INDEX IUmfrageWohnort  
ON Umfrage(Upper(Wohnort)) ;
```

- ▶ Der Index wird auf die Großbuchstaben angewendet
- ▶ Nur bei Verwendung von Upper(Wohnort) ist die Suche jetzt schnell

Vergleich zwischen In und Exists

▶ Trugschluss:

- ▶ Der Exists-Operator ist in der Regel nicht langsamer als der In-Operator!
- ▶ Das Gleiche gilt für Not Exists und Not In

▶ Beispiel:

- ▶ Ausgabe aller Mitarbeiter, die weniger als Mitarbeiter 3 verdienen, siehe Kapitel 5
 - ▶ Lösung mit Vergleichsoperator
 - ▶ Lösung mit Verbund
 - ▶ Lösung mit Exists-Operator

Ausführungsplan mit Vergleich

SQL | 0 Sekunden

Abfrageergebnis x Explain-Plan x

... dann FullScan von Personal

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			4
TABLE ACCESS	PERSONAL	FULL	3
Filter Predicates GEHALT < (SELECT GEHALT FROM PERSONAL WHERE PERSNR = 3)			
TABLE ACCESS	PERSONAL	BY INDEX ROWID	1
INDEX PK_PERSONAL	PK_PERSONAL	UNIQUE SCAN	0
Access Predicates PERSNR=3			

... dann Filter ...

Personal mittels Index scannen ...

Ausführungsplan mit Exists

```
SELECT *  
FROM Personal P1  
WHERE EXISTS ( SELECT * FROM Personal  
                WHERE Persnr = 3 AND P1.Gehalt < Gehalt ) ;
```

Abfrageergebnis x Explain-Plan x

SQL | 0,015 Sekunden

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			4
NESTED LOOPS			4
TABLE ACCESS	PERSONAL	BY INDEX ROWID	1
Filter Predicates			
GEHALT>500			
INDEX	PK_PERSONAL	UNIQUE SCAN	0
Access Predicates			
PERSNR=3			
TABLE ACCESS	PERSONAL	FULL	3
Filter Predicates			
AND			
P1.GEHALT<6000			
P1.GEHALT<GEHALT			

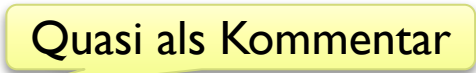
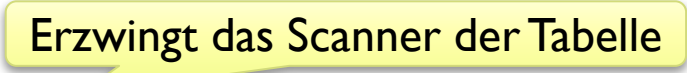
... dann Join

... dann weiterer Filter ...

Personal mittels Index scannen und filtern...

... dann FullScan von Personal ...

Hinweise (HINTS)

- ▶ Nicht immer kennt der Optimizer die Daten optimal
- ▶ Dem Optimizer können daher Hinweise gegeben werden
- ▶ Beispiel:
 - ▶ Wir wollen, dass bei der Abfrage auf den Wohnort in der Relation Umfrage kein Index verwendet wird
 - ▶ Lösung in Oracle:  Quasi als Kommentar
Select /*+ NO_INDEX(Umfrage) */ * From Umfrage Where ...
 - ▶ Lösung in SQL Server:  Erzwingt das Scanner der Tabelle
Select * From Umfrage With (FORCESCAN) Where ...
 - ▶ Lösung in MySQL:
Select * From Umfrage Ignore Index(IUmfrageWohnort) Where ...

Stored Procedures

- ▶ **Eine Stored Procedure**

- ▶ ist eine Prozedur (procedure)
- ▶ wird in der Datenbank abgespeichert (stored)

- ▶ **Eine Stored Procedure enthält**

- ▶ SQL Befehle

z.B. Select, Insert, Update,
Delete, Create Index

- ▶ Variablen-Deklarationen

z.B.: DECLARE nummer INT;

- ▶ Anweisungen

z.B.: SET nummer = 3;

- ▶ **Stored Procedures sind normiert mit**

- ▶ SQL 3 (SQL 1999)
- ▶ SQL 2003

Stored Procedure versus Trigger

- ▶ **Gemeinsam:**

- ▶ Prozeduren
- ▶ In Datenbank gespeichert

- ▶ **Unterschiede:**

- ▶ Trigger werden bei Ereignissen automatisch aufgerufen
- ▶ Stored Procedures werden explizit aufgerufen

- ▶ **Syntax:**

- ▶ **CREATE TRIGGER**
- ▶ **CREATE PROCEDURE**

Beispiel

▶ Prozedur RABATT:

▶ Parameter:

- ▶ ARTNR: Artikelnummer, auf den Rabatt gewährt wird
- ▶ NACHLASS: Der Preisnachlass in Euro für den Artikel

▶ Idee:

- ▶ Wenn der Nachlass unter 50% des eingetragenen Preises liegt, so wird der angegebene Nachlass vom Preis des angegebenen Artikels abgezogen. Der neue Preis wird in der Relation eingetragen.
- ▶ Sonst wird genau ein Nachlass von 50% gewährt. Der neue Preis wird entsprechend eingetragen.

▶ Realisierung:

- ▶ CREATE PROCEDURE Rabatt

Prozedur RABATT in Oracle

nicht NUMERIC(8,2)

```
CREATE OR REPLACE PROCEDURE Rabatt ( artnr INT , nachlass NUMERIC ) AS
```

```
  altPreis NUMERIC(8,2);
```

kein: DECLARE ... (gemäß Norm)

```
  neuPreis NUMERIC(8,2);
```

```
BEGIN
```

speichert Attribut Preis in Variable altPreis

```
  SELECT Preis      INTO altPreis
```

```
  FROM Artikel      WHERE anr = artnr;
```

```
  neuPreis := altPreis - nachlass;
```

kein: SET var = ... (gemäß Norm)

```
  IF neuPreis < 0.5 * altPreis THEN neuPreis := 0.5 * altPreis; END IF;
```

IF: normkonform!

```
  UPDATE Artikel
```

```
  SET Preis = neuPreis, Netto = Netto * neuPreis / altPreis, Steuer = Steuer * neuPreis / altPreis
```

```
  WHERE Anr = artnr ;
```

```
EXCEPTION WHEN OTHERS THEN
```

Ausnahmebehandlung in Oracle
WHEN OTHERS: alle sonstigen Ausnahmen

```
  DBMS_OUTPUT.PUT_LINE ('Fehler beim Ausfuehren der Prozedur Rabatt');
```

```
END;
```

Begrenzer in Oracle

gibt Zeile aus

```
/
```



Prozedur RABATT in SQL Server

Keine Klammern!

```
CREATE PROCEDURE Rabatt @artnr INT, @nachlass NUMERIC(8,2) AS  
DECLARE @altPreis NUMERIC(8,2), @neuPreis NUMERIC(8,2);  
BEGIN TRY
```

Variablen beginnen mit @

```
    SELECT @altPreis = Preis  
    FROM Artikel WHERE anr = @artnr;
```

speichert Attribut Preis in Variable altPreis

```
    SET @neuPreis := @altPreis - @nachlass;
```

```
    IF @neuPreis < 0.5 * @altPreis SET @neuPreis := 0.5 * @altPreis;
```

IF: nicht normkonform!

```
    UPDATE Artikel
```

```
    SET Preis = @neuPreis, Netto = Netto * @neuPreis / @altPreis,  
        Steuer = Steuer * @neuPreis / @altPreis
```

```
    WHERE Anr = @artnr ;
```

```
END TRY
```

TRY - CATCH

```
BEGIN CATCH
```

```
    SELECT 'Fehler beim Ausfuehren der Prozedur Rabatt';
```

```
END CATCH
```

```
GO
```

Begrenzer in SQL SERVER

Prozedur RABATT in MySQL

Definition eines Begrenzers

DELIMITER //

CREATE PROCEDURE Rabatt (artnr INT , nachlass NUMERIC(8,2))

BEGIN

DECLARE altPreis NUMERIC(8,2);

DECLARE-Teil

DECLARE neuPreis NUMERIC(8,2);

SELECT Preis INTO altPreis
FROM Artikel WHERE anr = artnr;

speichert Attribut Preis in Variable altPreis

SET neuPreis = altPreis - nachlass;

normgemäß

IF neuPreis < 0.5 * altPreis THEN SET neuPreis = 0.5 * altPreis; END IF;

UPDATE Artikel

SET Preis = neuPreis, Netto = Netto * neuPreis / altPreis, Steuer = Steuer * neuPreis / altPreis
WHERE Anr = artnr ;

END;

//

Aufruf von Prozeduren

- ▶ Aufruf erfolgt in Oberfläche (Oracle, SQL Server)
- ▶ Aufruf erfolgt in Kommandozeile:
 - ▶ Oracle: `execute rabatt ( 100002, 50 )`
 - ▶ SQL Server: `execute rabatt 100002, 50`
 - ▶ MySQL: `call rabatt ( 100002, 50 )`
- ▶ Damit wird Artikel 100002 um 50 Euro günstiger

Optimierung: Transaktionsbetrieb

- ▶ In MySQL

- ▶ Abschalten von Autocommit

- ▶ SET Autocommit = 0

- ▶ oder

- ▶ START TRANSACTION ... COMMIT

Optimierung: bindParam

- ▶ Oft werden Befehle mit modifizierten Parametern mehrfach aufgerufen
- ▶ Dann: Arbeiten mit bindParam (prepare + execute, statt query)

```
$sql = "Select Auftrnr, Artnr, Anzahl, Gesamtpreis  
      From Auftragsposten Where Auftrnr = :anr" ;
```

Variable

```
$stmt = $conn->prepare($sql);
```

Nur einmal geparkt und aufbereitet!

```
$stmt->bindParam( ':anr', 1 ) ;
```

Ersetzt :anr durch den Wert 1

```
$stmt->execute( ) ;
```

```
$stmt->bindParam( ':anr', 2 ) ;
```

Ersetzt :anr durch den Wert 2

```
$stmt->execute( ) ;
```

```
$stmt->bindParam( ':anr', 4 ) ;
```

Ersetzt :anr durch den Wert 4

```
$stmt->execute( ) ;
```

Mehrfach ausgeführt

Zusammenfassung

- ▶ Eine Datenbank führt selbst Optimierungen durch
- ▶ Eine Datenbank stellt viele Werkzeuge zur Optimierung zur Verfügung:
 - ▶ Indexe
 - ▶ Partitionierungen
 - ▶ Materialisierte Sichten
 - ▶ Stored Procedures
- ▶ Der Anwender und der Systemverwalter setzen diese Werkzeuge gezielt ein
- ▶ Der Anwender optimiert zusätzlich