Data Warehousing

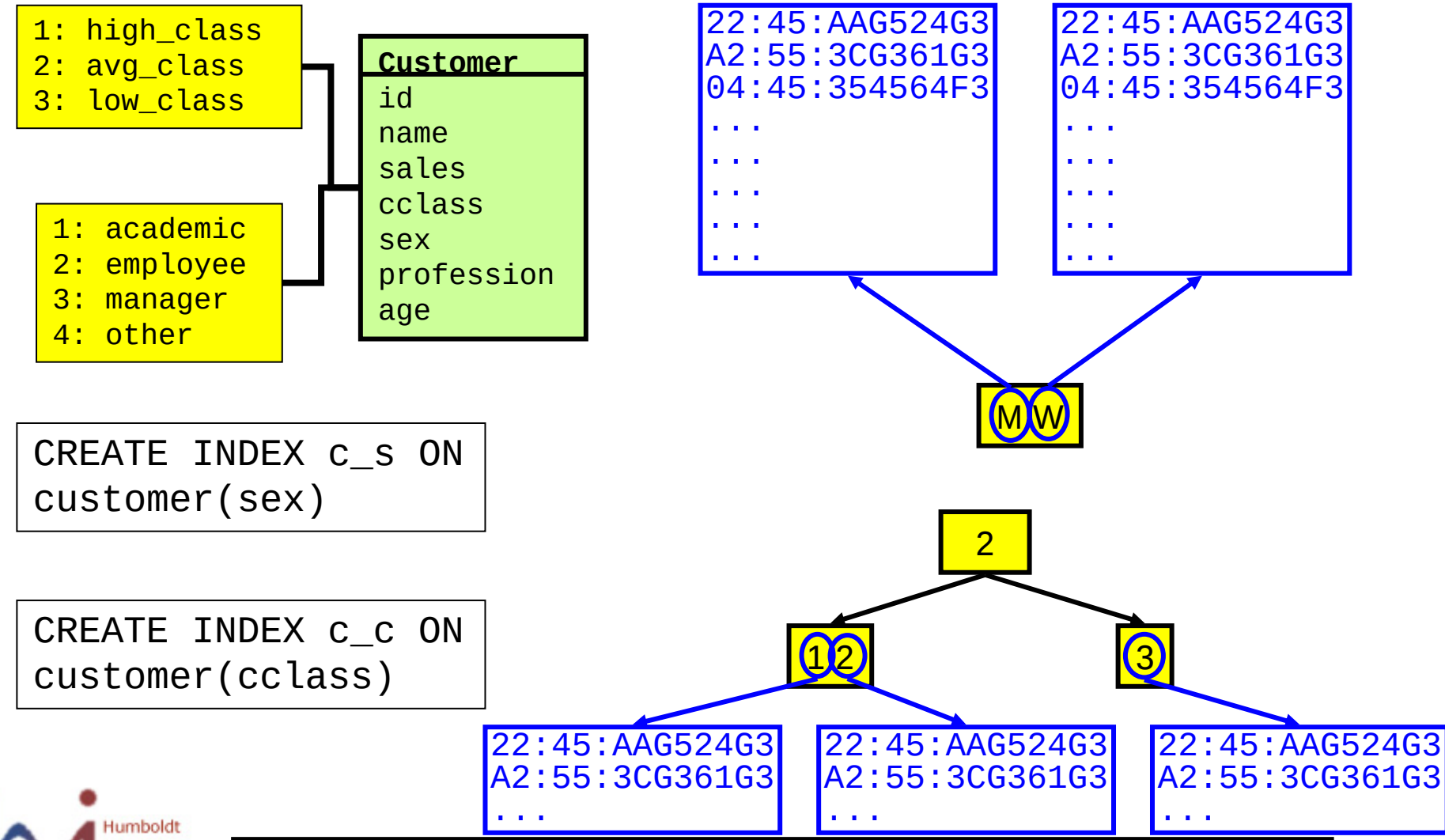
STAR Joins

Partitionierung

Ulf Leser

Wissensmanagement in der Bioinformatik

Degenerierte B*-Bäume



Falsch geordnete B*-Bäume

- Zusammengesetzter Index
  - Indexierung der Konkatenation mehrerer Attribute
  - Selektivität des zusammengesetzten Index = Produkt der Selektivitäten der Einzelindizes (bei statistischer Unabhängigkeit der Attributwerte)
- Problem: Ordnungsabhängig

```
CREATE INDEX c_scp ON
customer(sex, cclass,
profession)
```

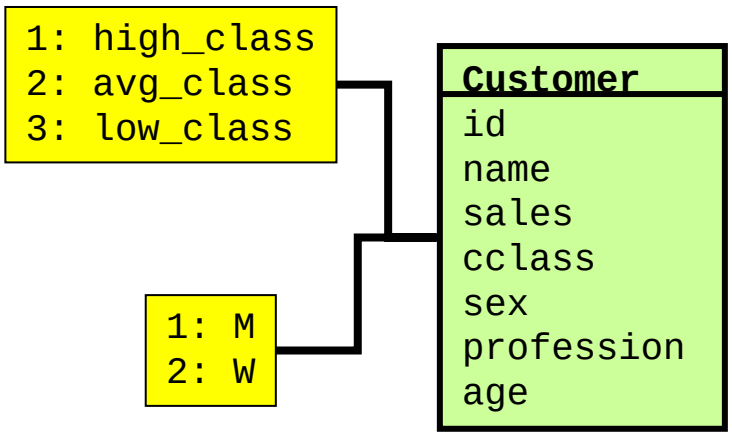
```
SELECT ...
FROM ...
WHERE   sex=‚m‘ AND
        cclass=1 AND
        prof=‚other‘
```

```
SELECT ...
FROM ...
WHERE   cclass=1 AND
        prof=‚other‘ AND
        sex=‚m‘
```

```
SELECT ...
FROM ...
WHERE   cclass=1 AND
        prof=‚other‘
```

Bitmapindex Grundaufbau

- Tabelle T mit Attribut A; |T|=n, |A|=a
- Repräsentation eines Attributwerts: Bitarray der Länge n
- Repräsentation aller Attributwerte: Bitmatrix mit n*a Bits



Tabelle

```
22:45,1,Meier,20.000,2,M,...
A2:55,2,Müller,15.000,3,W,...
33:D1,3,Schmidt,25.000,1,M,...
1A:0E,4,Dehnert,22.000,2,M,...
...
```

Bitmapindex cclass

```
1:0010...
2:1001...
3:0100...
```

Bitmapindex sex

```
M:1011...
W:0100...
```

Verwendung von Bitmapindexen

- Wann lohnen sich Bitmap-Indexe?
  - Attribute geringer Kardinalität
  - Statische Daten
    - Wenig INSERT / DELETE
  - Häufige Punktanfragen
  - Unvorhersehbare Kombinationen von Bedingungen
    - Ad-Hoc Anfragen
- Vorteile
  - Geringer Speicherbedarf
  - Effiziente Berechnung zusammen gesetzer Prädikate
  - Unabhängig von Reihenfolge
- Nachteile
  - Teuer Unterhalt – gut für statische Daten
  - Nur für Attribute geringer Kardinalität

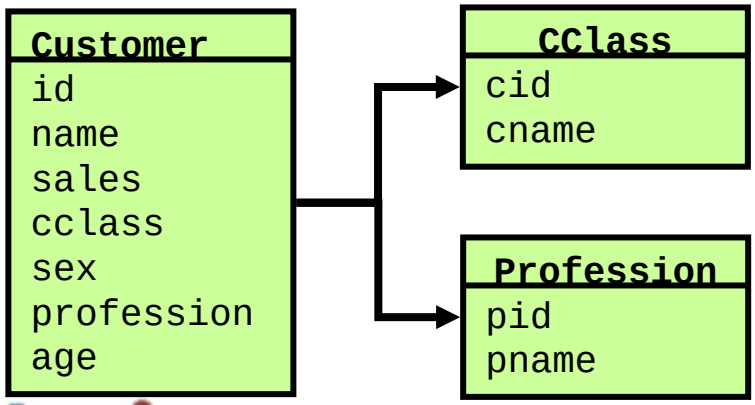
Kompression: RLE Beispiel

- Beispiel: Run-Length Encoding (RLE)
  - N=1.000.000, a=100, RID: 4 Byte
  - Pro Bitarray ist nur einer von 100 Werten eine „1“
  - Explizites Speichern der Positionen p_i mit 1: $p_1,p_2,p_3,p_4,\ldots$
  - Bitmap ohne RLE: 1.000.000 * 100/8 = 12,5 MB
  - Bitmap mit RLE
    - 1.000.000 ist durch 20 Bit adressierbar

      1.000.000 * (20/8) /100*100 = 2.5MB
- Vorsicht vor Sperren bei komprimierten Bitmaps
  - Sehr viele Bits in einer Page
  - Sperrung der Page kann zig-tausend Tupel sperren
  - Geeignet nur für Read-Only Umgebungen
- Nachteil RLE: Teure Dekomprimierung
  - Sonst keine Bitoperationen möglich

Bitmapped Join Index

- Unsere bisherigen Beispiele für Bitmapindexe waren nur zum Teil realistisch
  - Bedingungen nicht an FK, sondern an „semantische“ Attribute der Dimensionstabellen gerichtet
  - Bitmaps auf FK nützen dann nichts, sie verhindern nicht den Join zu Dimensionstabellen (aber: STAR JOIN)
- Lösung: Bitmap Join Index

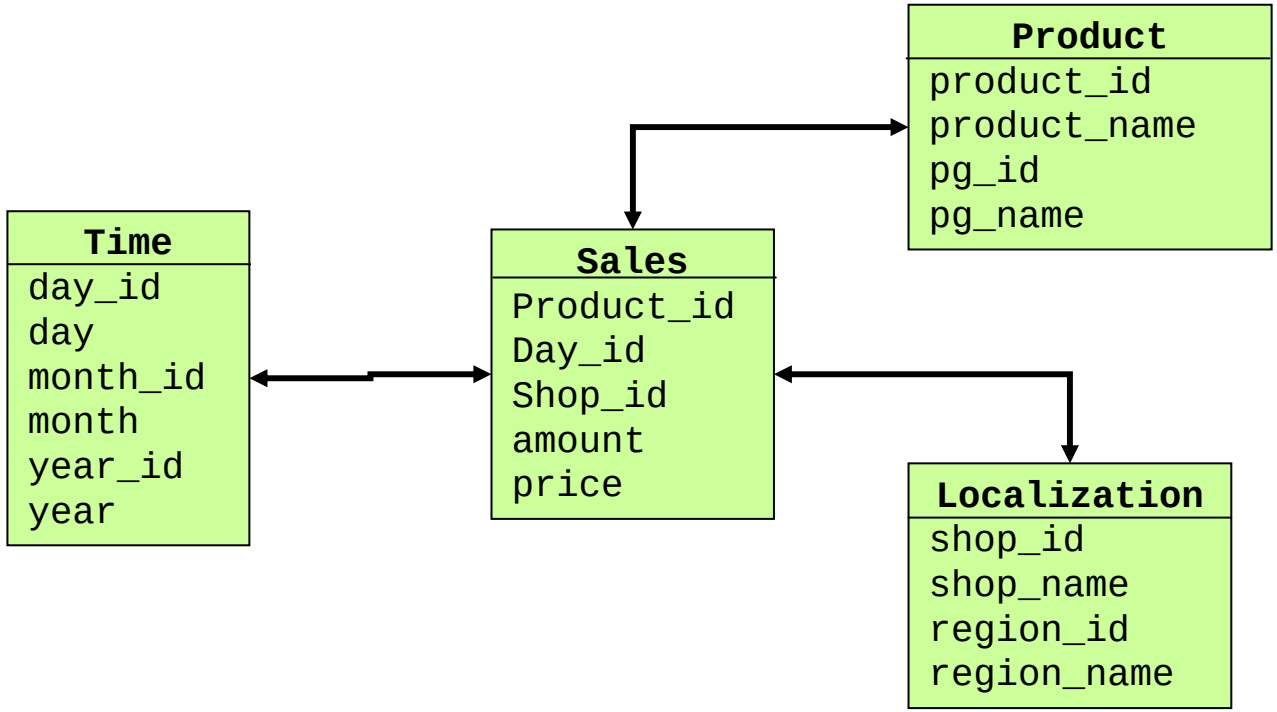


- Bitmaps erstellen für Tupel in Customer für alle Werte in `cclass.cname` und in `profession.pname`
- Nur auf der „N“ Seite sinnvoll

Inhalt dieser Vorlesung

- STAR Joins
- Partitionierung
- Partitionierung in Oracle

Star Join



- Typische Anfrage gegen Star Schema
  - Aggregation und Gruppierung
  - Bedingungen auf den Werten der Dimensionstabellen
  - Joins zwischen Dimensions- und Faktentabelle

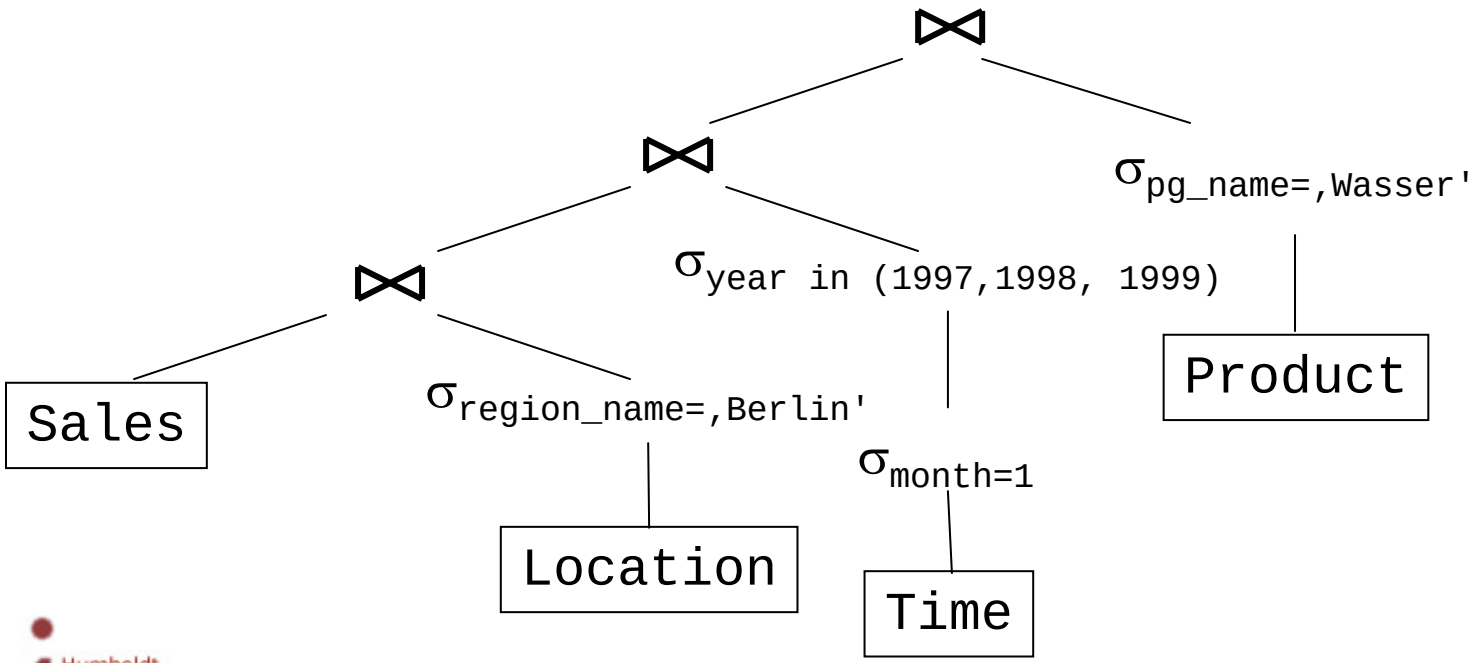
Beispielquery

- Alle Verkäufe von Produkten der Produktgruppe ‚Wasser‘ in Berlin im Januar der Jahre 1997, 1998, 1999, gruppiert nach Jahr

```
SELECT T.year, sum(amount*price)
FROM Sales S, Product P, Time T, Localization L
WHERE   P.pg_name=‚Wasser‘ AND
        P.product_id = S.product_id AND
        T.day_id = S.day_id AND
        T.year in (1998, 1998, 1999) AND
        T.month = ‚1‘ AND
        L.shop_id = S.shop_id AND
        L.region_name=‚Berlin‘
GROUP BY T.year
```

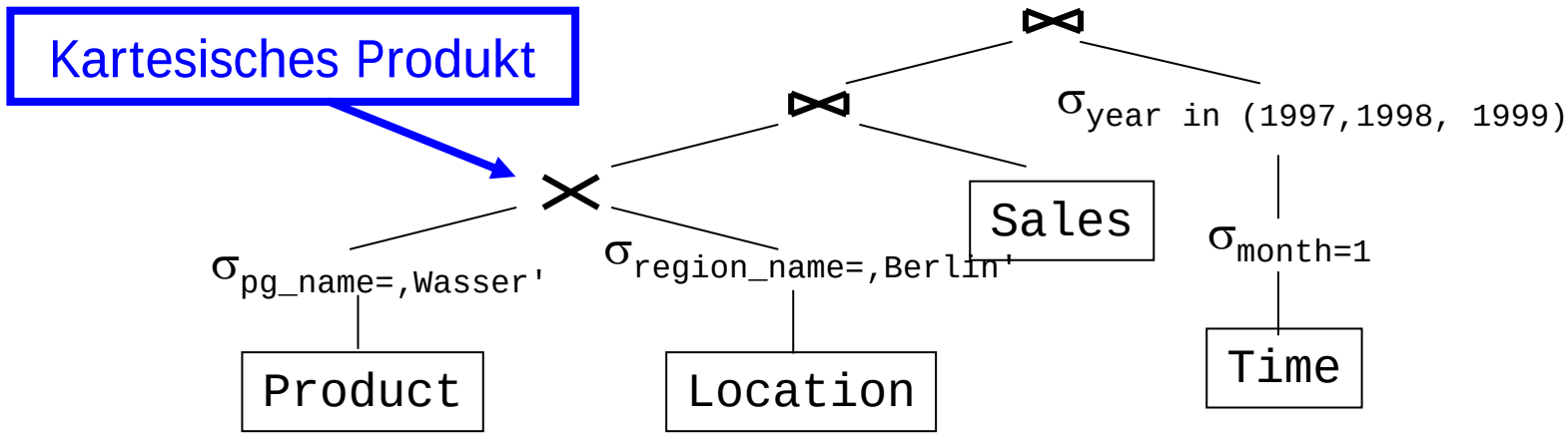
Anfrageplanung

- Anfrage enthält 3 Joins über 4 Tabellen
- Zunächst 4! left-deep join trees
  - Aber: Nicht alle Tabellen sind mit allen gejoined
- Nur 3! beinhalten kein Kreuzprodukt



Heuristiken

- Typisches Vorgehen
  - Auswahl des Planes nach Größe der Zwischenergebnisse
  - Keine Beachtung von Plänen, die kartesisches Produkt enthalten



Abschätzung von Zwischenergebnissen

```
SELECT T.year, sum(amount*price)
FROM Sales S, Product P, Time T, Localization L
WHERE     P.pg_name=‚Wasser‘ AND
          P.product_id = S.product_id AND
          T.day_id = S.day_id AND
          T.year in (1998, 1998, 1999) AND
          T.month = ‚1‘ AND
          L.shop_id = S.shop_id AND
          L.region_name=‚Berlin‘
GROUP BY T.year
```

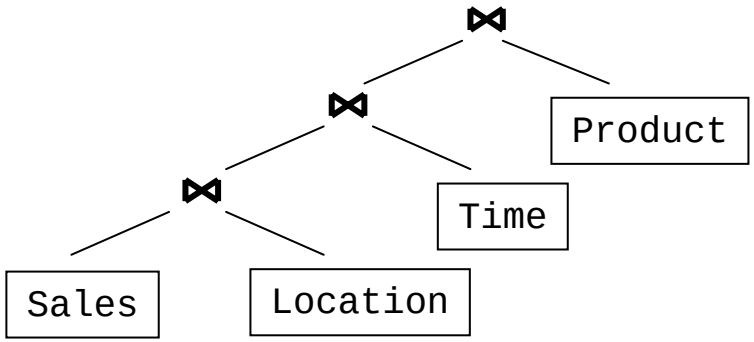
Annahmen

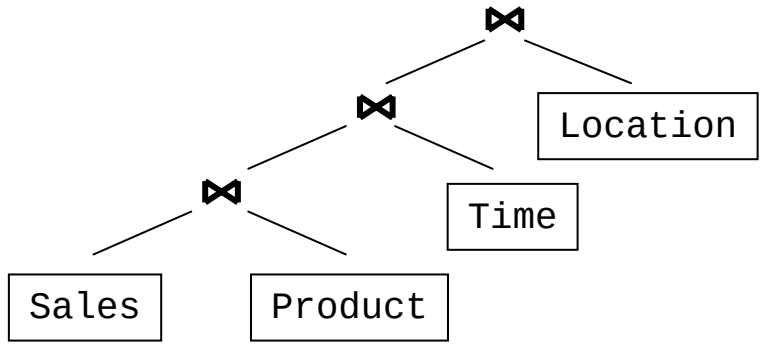
- M= |S| = 100.000.000
- 20 Verkaufstage pro Monat
- Daten von 10 Jahren
- 50 Produktgruppen a 20 Produkten
- 15 Regionen a 100 Shops
- Gleichverteilung aller Verkäufe

Größte des Ergebnis

- Selektivität Zeit
  - 60 Tage:
    (M / (20*12*10)) * 3*20
- Selektivität ‚Wasser‘
  - 20 Produkte
    (M / (20*50)) * 20
- Selektivität ‚Berlin‘
  - 100 Shops
    (M / (15*100)) * 100
- Gesamt
  - 3.333 Tupel
- Selektivität: 0,00003%

Left-deep Pläne

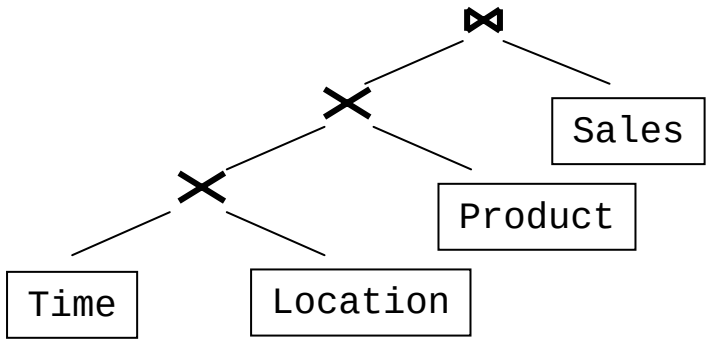




	Zwischen-ergebnis
1. Join (M / 15)	6.666.666
2. Join ($\|J_1\|$*3/120)	166.666
3. Join ($\|J_2\|$/50)	3.333

	Zwischen-ergebnis
1. Join (M / 50)	2.000.000
2. Join ($\|J_1\|$*3/120)	50.000
3. Join ($\|J_2\|$/ 15)	3.333

Plan mit kartesischen Produkten



	Zwischenergebnis
1. Time x Location (3*20 * 100)	6.000
2. … x Product ($\|P_1\|$*20)	120.000
3. … ⋈ Sales	3.333

- Es gibt mehr „Zellen“ als Verkäufe
- Nicht an jedem Tag wird jedes Produkt in jedem Shop verkauft

STAR Join in Oracle (v7)

- STAR Join Strategie in Oracle v7
  - Kartesisches Produkt aller Dimensionstabellen
  - Zugriff auf Faktentabelle über Index
    - Hohe Selektivität für Anfrage wichtig
    - Zusammengesetzter Index auf allen FKs muss vorhanden sein
    - Sonst „nur“ kleinere Zwischenergebnisse, aber trotzdem teurer Scan
- Weiterer Vorteil des kartesischen Produkts
  - „Berichtsform“: Auch leere Würfelzellen sollen ins Ergebnis
  - Werden durch das kartesische Produkt alle gebildet
  - Äquivalent zu Outer-Joins
- Nicht immer gut
  - Daten für 3 Monate, 10 Jahre, 5 Regionen, 10 Produktgruppen
  - Größe des kartesischen Produkts:
    3*20*10 * 5*100 * 10*20 = 60.000.000

STAR Join in Oracle 8i – 9i

- Neue STAR Join Strategie seit Oracle 8i
- Möglichkeit der (komprimierten) Bitmapindexe lässt kartesisches Produkt weniger vorteilhaft erscheinen
- Phasen
  1. Berechnung aller FKs in Faktentabelle gemäß Dimensionsbedingungen einzeln für jede Dimension
  2. Anlegen/laden von Join-Bitmapindexen auf allen FK Attributen der Faktentabelle
  3. Merge (AND) aller Bitmapindexe
  4. Direkter Zugriff auf Faktentabelle über TID
  5. Join nur der selektierten Fakten mit Dimensionstabellen zum Zugriff auf Dimensionswerte
- Zwischenergebnisse sind nur (komprimierte) Bitlisten

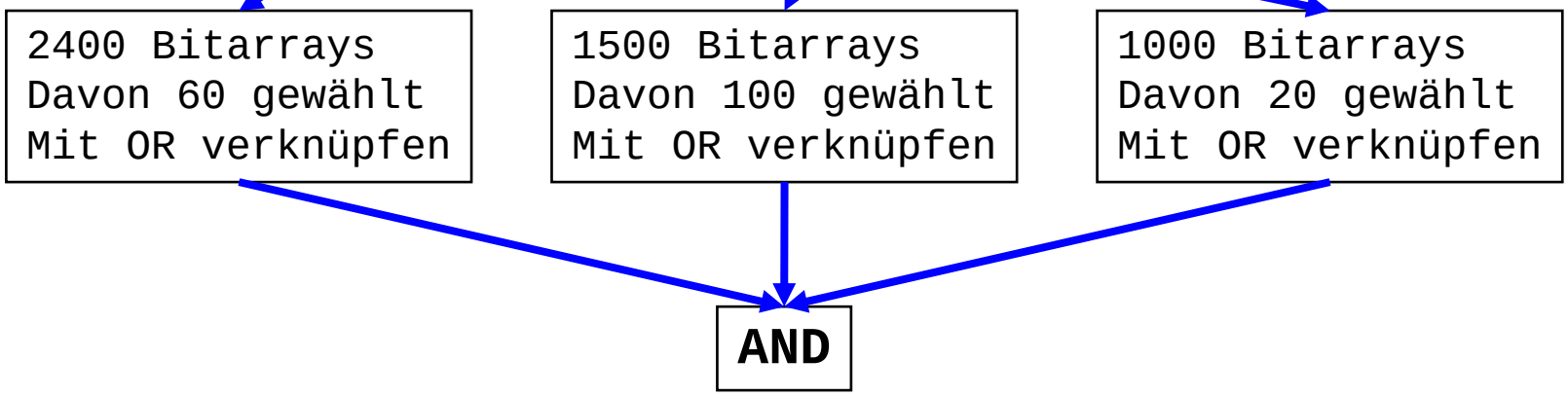
Beispiel – Schritt 1

```
SELECT T.year, sum(amount*price)
FROM Sales S, Product P, Time T, Localization L
WHERE   P.pg_name=‚Wasser‘ AND
        P.product_id = S.product_id AND
        T.day_id = S.day_id AND
        T.year in (1998, 1998, 1999) AND
        T.month = „1“ AND
        L.shop_id = S.shop_id AND
        L.region_name=‚Berlin‘
GROUP BY T.year
```

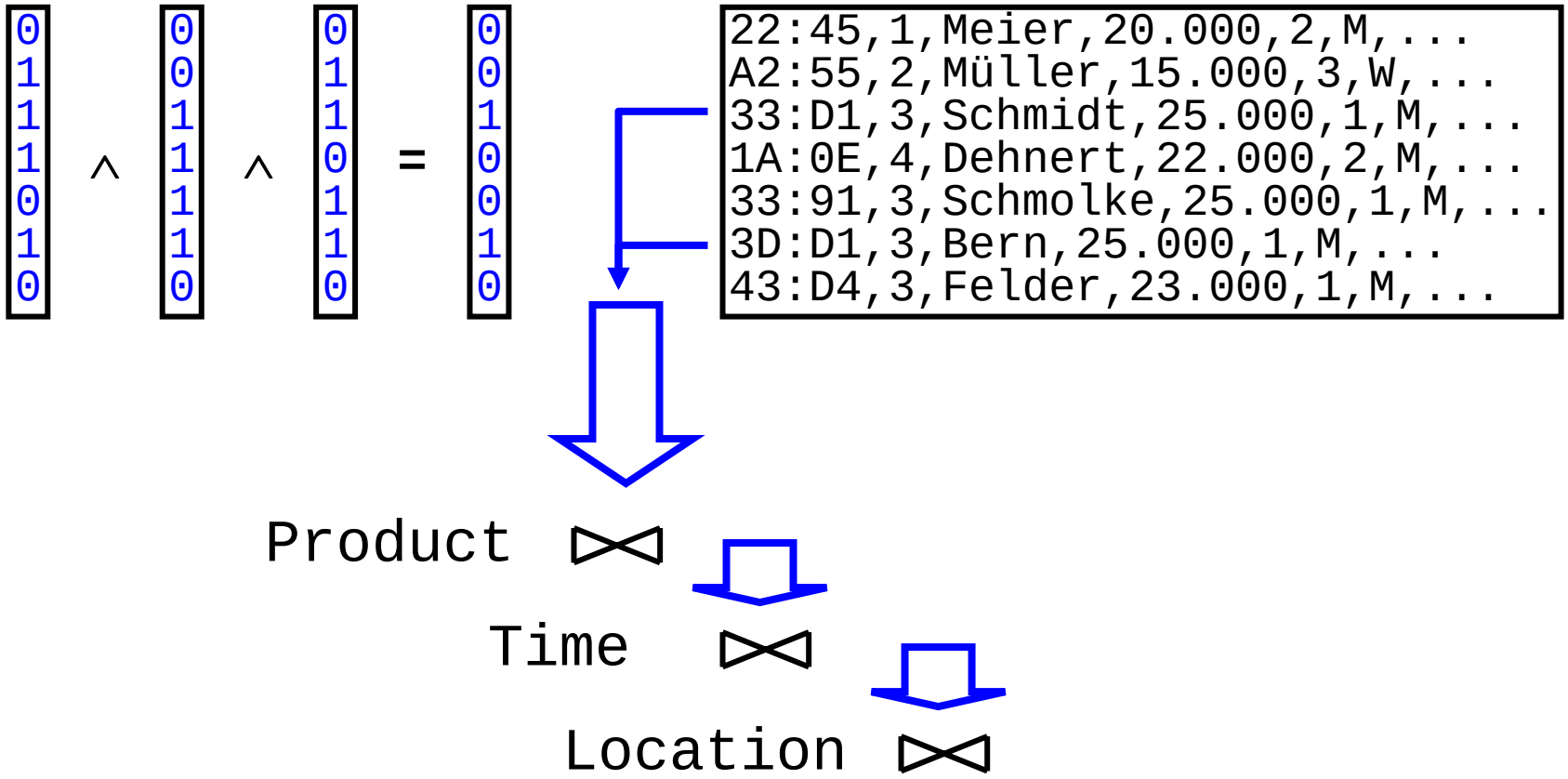

```
SELECT T.year, sum(amount*price)
FROM Sales S, Product P, Time T, Localization L
WHERE   S.day_id IN
        (SELECT day_id FROM time
         WHERE year in (1998, 1998, 1999) AND month=‚1‘)
        AND S.shop_id IN
        (SELECT shop_id FROM localization WHERE region_name=‚Berlin‘)
        AND S.product_id IN
        (SELECT product_id FROM product WHERE pg_name=‚Wasser‘)
GROUP BY T.year
```

Beispiel – Schritt 2 & 3

```
SELECT T.year, sum(amount*single_price)
FROM Sales S, Product P, Time T, Localization L
WHERE   S.day_id IN
        (SELECT day_id FROM time
         WHERE year in (1998, 1998, 1999) AND month=‚1‘)
        AND S.shop_id IN
        (SELECT shop_id FROM localization WHERE region_name=‚Berlin‘)
        AND S.product_id IN
        (SELECT product_id FROM product WHERE pg_name=‚Wasser‘)
GROUP BY T.year
```



Beispiel – Schritt 4 & 5



Gesamtplan

Phase 2

Phase 1

```
SELECT STATEMENT
 SORT GROUP BY
  HASH JOIN
   TABLE ACCESS FULL                          LOCATION
   HASH JOIN
    TABLE ACCESS FULL                         TIME
    HASH JOIN
     TABLE ACCESS FULL                        PRODUCT
     PARTITION RANGE ALL
      TABLE ACCESS BY LOCAL INDEX ROWID       SALES
       BITMAP CONVERSION TO ROWIDS
        BITMAP AND
         BITMAP INDEX SINGLE VALUE            SALES_L_BJIX
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL                 PRODUCT
           BITMAP INDEX RANGE SCAN            SALES_P_BIX
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL                 TIME
           BITMAP INDEX RANGE SCAN            SALES_TIME_BIX
```

Weitere Verfeinerung

- STAR Join schwierig, wenn Bitarrays nicht in Hauptspeicher passen
  - Swappen
  - Performancegewinn geht verloren
- Idee: Bloom-Filter
  - „Unscharfe“ Schritte 2-3
  - Aufbau einer maximal großen Bitmap B
    - Maximal bzgl. Hauptspeichergröße
  - Auswahl einer Dimension D
    - Die mit der geringsten Selektivität
  - Wahl einer Hashfunktion h: TIDs nach B
    - Mehrere TIDs werden auf das selbe Bit in B abgebildet

Ablauf

- Für alle selektierten Tuple t aus D
  - Scan Faktentabelle nach FK-Wert aus t
  - $B_D[h(TID)]=1$
- Iteriere über alle restlichen D_i
  - Berechne $B_{Di}[]$ wie oben
  - $B_D[] = B_D[]$ AND $B_{Di}[]$
- Schließlich
  - $B_D[z]=0$: Alle Tupel mit h(TID)=z garantiert nicht relevant
  - $B_D[z]=1$: Alle Tupel mit h(TID)=z eventuell relevant
- Erneute Auswertung aller Bedingungen nach Joins mit Dimensionstabellen
  - Menge der zu untersuchenden Fakten aber sehr klein

Eigenschaften

- Implementiert in DB2
- Kernidee ist Reduktion der Faktentabelle vor Joins mit Dimensionstabellen
  - Faktentabelle muss trotzdem mehrmals gescanned werden
  - Aber immer nur Zugriff auf FK
  - Kein Zwischenspeichern (z.B. wegen Sortierung) notwendig
- Dynamischer Aufbau von Bitmaps
  - DB2 hat keine persistenten Bitmapindexe
- Voraussetzungen
  - Hohe Selektivität in Dimensionsbedingungen
  - Aufbau einer hinreichend großen Bitmap möglich
  - Wahrscheinlichkeit für „falsche“ 1 steigt proportional mit Verhältnis M / |B|

STAR Joins und Join Indexe

- Verfahren können tw. gut kombiniert werden

Oracle STAR Joins
1. Extraktion der Dim-IDs
2. Auswahl der Bitmap-Arrays
3. Merge (AND) aller Bitmapindexe
4. Direkter Zugriff über RID
5. Joinen der Fakten mit Dim.

Kann bei Vorhandensein von Bitmapped Joinindex wegfallen

- Keine Kombination mit Bloom-Filter
  - Unscharfe Bitmaps müssen jedes Mal berechnet werden

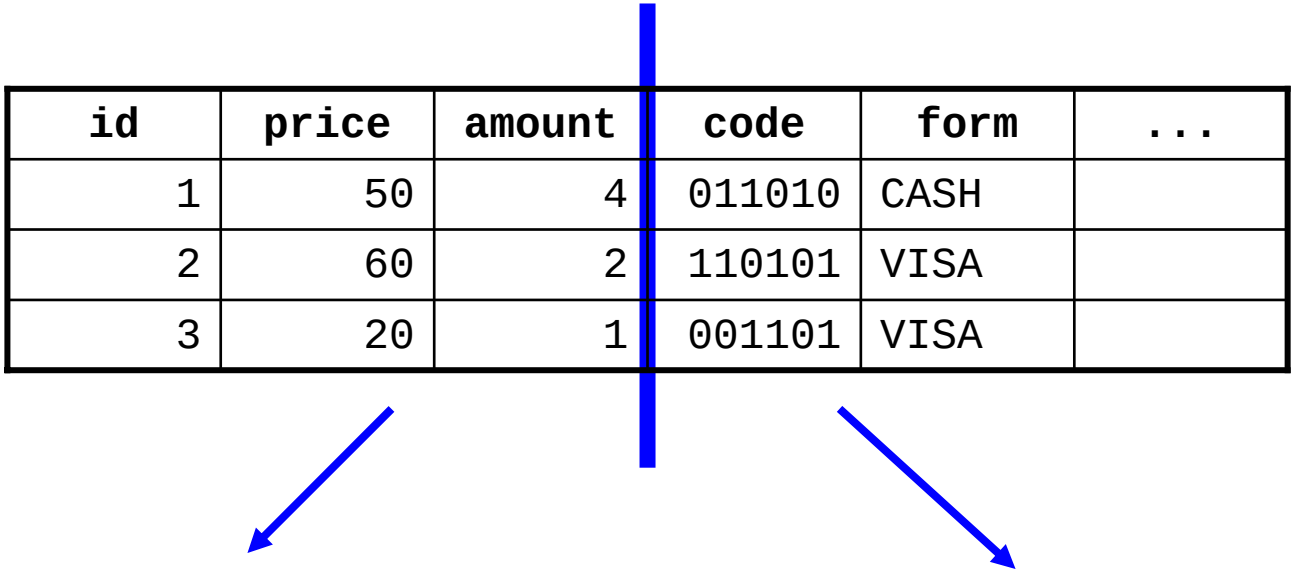
Teil 2. Partitionierung

- Prinzip
- Partitionierungsarten
- Management von Partitionen
- Spezielle Themen
  - Partitionierung und Indexe
  - Partitionierung und Parallelität
  - Partitionierung und Joins

Teil 2. Partitionierung

- Aufteilung der Daten einer Tabelle in Untereinheiten
  - Physische Partitionierung (Allokation): Für den Benutzer transparent
    - Nach Definition der Partitionen
  - Logische Partitionierung: Für den Benutzer nicht transparent
    - Explizites Anlegen mehrerer Tabellen
    - Anfragen müssen entsprechend formuliert werden
- Ursprünglich für verteilte Datenbanken entwickelt
  - Verteilung der Partitionen auf verschiedenen Knoten
  - Vereinfachte Synchronisation
  - Lastverteilung
  - Erfolg unmittelbar abhängig von Lokalität Anfrage – Partition
- ➢ Wichtiges Feature für DWH
  - ➢ Verbessertes Datenmanagement
  - ➢ Parallelisierung
  - ➢ Partition Pruning

Vertikale Partitionierung



id	price	amount	code	form	...
1	50	4	011010	CASH	
2	60	2	110101	VISA	
3	20	1	001101	VISA	

id	price	amount
1	50	4
2	60	2
3	20	1

id	code	form	...
1	011010	CASH	
2	110101	VISA	
3	001101	VISA	

Bewertung

- „Zerstört“ semantische Einheiten – die Tupel
- Zusammenfassen erfordert (teure) Joins
- Geeignete Technik zur „Auslagerung“ ...
  - von wenig benutzter Daten
  - von Attributen, die öfter verändert werden (und deshalb zu Reorganisation / Row Splits führen)
  - von BLOBs, LONGS, etc.
- Transparenz für Benutzer ist möglich
  - Definition eines Views
  - Performanceverschlechterung, wenn Optimierer mit dem View nicht richtig umgehen kann

Beispiel

```
create table t1(
    id number,
    foo number);
alter table t1
    add constraint t1_pk
    primary key (id);
create table t2(
    id number,
    bar varchar(4000),
    blobbel blob);
alter table t1
    add constraint t2_fk
    foreign key (id) references t1(id);
create or replace view tall as
    select t1.id, t1.foo, t2.bar, t2.blobbel
    from t1,t2
    where t1.id=t2.id;
create or replace view tall2 as
    select t1.id, t1.foo, t2.bar, t2.blobbel
    from t1,t2
    where t1.id=t2.id(+);
select id from tall;
select id from tall2;
```

```
#   Operation            Options     Name
0   SELECT STATEMENT
1     NESTED LOOPS
2       TABLE ACCESS     FULL        T2
3       INDEX            UNIQUE SCAN T1_PK
```

```
#    Operation            Options Name
0    SELECT STATEMENT
1      MERGE JOIN          OUTER
2        SORT              JOIN
3          TABLE ACCESS    FULL    T1
4        SORT              JOIN
5          TABLE ACCESS    FULL    T2
```

Horizontale Partitionierung

id	price	amount	code	form	...
1	50	4	011010	CASH	
2	60	2	110101	VISA	
3	20	1	001101	VISA	
4	30	4	110101	CASH	

1	50	4	011010	CASH
2	60	2	110101	VISA

3	20	1	001101	VISA
4	30	4	110101	CASH

1	50	4	011010	CASH
4	30	4	110101	CASH

2	60	2	110101	VISA
3	20	1	001101	VISA

Horizontale Partitionierung

- Wichtige Erweiterung der meisten kommerziellen RDBMS der letzten Jahre

- Hauptvorteile
  - Datenmanagement – Partitionen als eigenständige DDL Objekte
  - Parallele Verarbeitung
  - Scans können Partitionen auslassen

Prinzip

```
CREATE TABLE sales_range
       (salesman_id  NUMBER(5),
        salesman_name VARCHAR2(30),
        sales_amount  NUMBER(10),
        sales_date    DATE)
PARTITION BY RANGE(sales_date)
( PARTITION t21 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY')),
  PARTITION t22 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),
  PARTITION t23 VALUES LESS THAN(TO_DATE('04/01/2000','DD/MM/YYYY')),
  PARTITION t24 VALUES LESS THAN(TO_DATE('05/01/2000','DD/MM/YYYY'))
);
```

- Partitionen sind eigene Datenbankobjekte (Tabellen)
  - Eigene DDL Kommandos
- Müssen explizit angelegt werden
- Einmal angelegt ist ihre Existenz für Benutzer transparent
  - Wie bei Indexen – einmal angelegt, werden Partitionen automatisch aktualisiert und verwendet

Arten von Partitionierung

- Bereichspartitionierung
  - Explizite Angabe der Partitionsbereiche
  - Über einzelne oder zusammengesetzte Attribute
  - Tupel mit entsprechenden Werten liegen in einer Partition
  - Gut für Scans über Attributbereiche (Partitionswahl)
  - Typische Partitionierungsattribute: Zeiträume
    - Gut zur Archivierung historischer Daten
- Hash – Partitionierung
  - Angabe einer Hashfunktion über Attributwerten eines Tupel
    - Zuteilung Tupel auf Partitionen
  - Tupel werden über verschiedene Partitionen verstreut
  - Gut für Parallelität (Parallele Joins, parallele Scans)
    - Wenn man Bedingungen nicht kennt – „zufällige“ Verteilung am sinnvollsten

Arten von Partitionierung

- Listenpartitionierung
  - Explizite Angabe der Werte pro Partition
  - Verhalten ähnlich Bereichspartitionierung
  - Möglich für ungeordnete oder ungleichverteilte Werte
- Zusammengesetzte Partitionierung
  - Kombination verschiedener Partitionierung

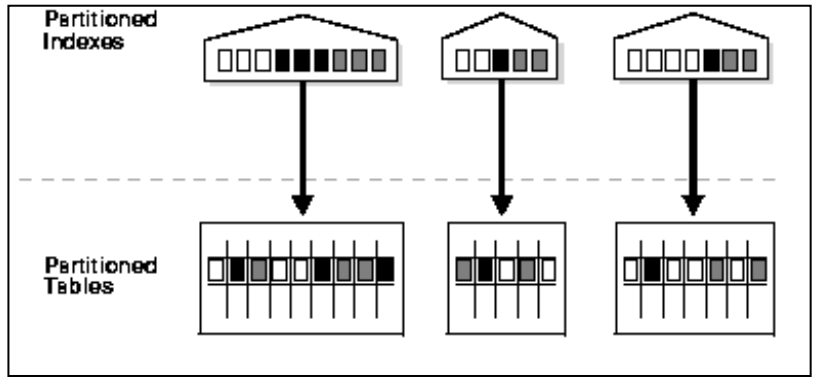
➢ Achtung
  - ➢ Größe der Partitionen wird nicht automatisch ausgeglichen
  - ➢ Schlechtes Partitionierungsattribut: Evt. sogar Performanceverschlechterung, da Parallelisierung ausgehebelt

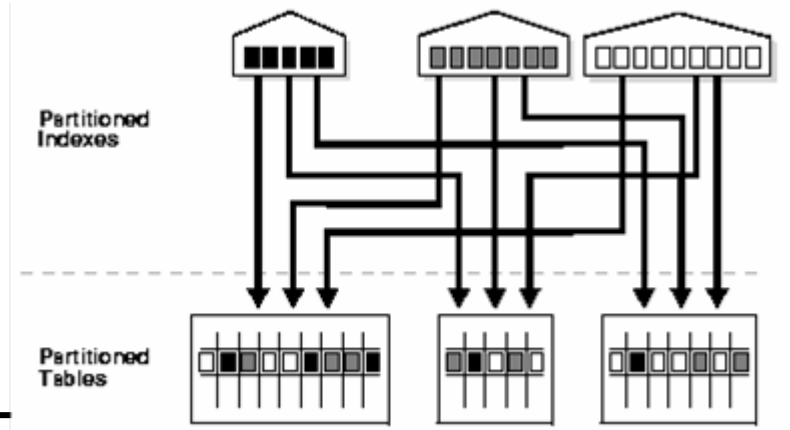
Partitionierung - Datenmanagement

- MERGE – Verschmelzen zweier Partitionen
- ADD – Hinzufügen einer Partition (abh. von P-Art)
- DROP/TRUNCATE – Löschen einer Partition
  - Viel schneller als „`DELETE FROM sales WHERE ...`“
  - Archivierung ohne Beeinträchtigung anderer Partitionen
  - DWH als „Sliding Window“
- Verteilung der Partitionen auf verschiedene Platten
  - Parallelität beim IO Zugriff
- EXCHANGE – Tabelle ↔ Partition
  - Sehr nützlich für ETL
  - Beispiel: Vorverarbeitung der Daten eines Tages in einer eigenen Tabelle
  - Dann hinzufügen zu `sales` mit einem Befehl
  - Keine Reorganisation, kein Indexneubau, ...

Partitionierung und Indexe

- Auch Indexe können partitioniert werden
- Indexierung partitionierter Tabellen
  - „Globale“ Indexe
    - Index unabhängig von Tabelle partitioniert (oder gar nicht)
    - Eigene DDL Kommandos
  - „Lokale“ Indexe
    - Index partitioniert wie Tabelle
    - Bildung eines lokalen Index (Indexpartition) pro Partition
    - Manipulation automatisch mit Tabellenpartition (DROP, ADD, TRUNCATE, ...)





Partitionierung – Pruning

- Bei Anfragen mit Bedingung auf Partitionierungsattribut
- Punktanfragen
  - Direkte Auswahl relevanter Partition
  - Partition ähnlich einer Ebene in B*-Baum
  - ➢ Keine Performanceverbesserung
- Bereichsanfragen
  - Direkte Auswahl relevanter Partitionen
  - Auch Daten liegen partitioniert vor (nicht nur TIDs)
  - Ähnlichkeit zu Index-Organized Tables
  - ➢ Erhebliche Performanceverbesserung
- Unterstützte Prädikate
  - Pruning mit Listen/Bereichspartitionierung nur bei =, <, >, IN
  - Pruning mit Hashpartitionierung nur bei =, IN

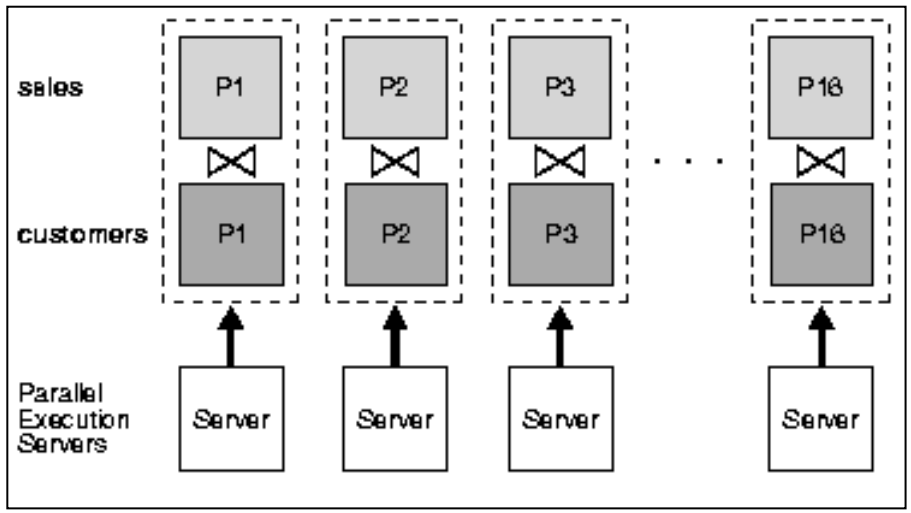
Partitionierung - Parallelität

- Arten von Parallelität
  - Interquery
    - Verteilen von Anfragen auf Prozessoren / Knoten
    - Keine Beschleunigung einzelner Queries
    - Behandlung auf Session-Ebene (Dedicated Server / MTS)
  - Intraquery
    - Aufbrechen der Query in parallel ausgeführte Teilanfragen
    - Option1: Query unverändert auf unterschiedlichen Datenbereichen ausführen (Partitionen)
    - Option 2: Parallelisierung einzelner Operatoren

➢ Partitionierung: Anfrage läuft parallel auf unterschiedlichen Partitionen

Partitionierung und Joins

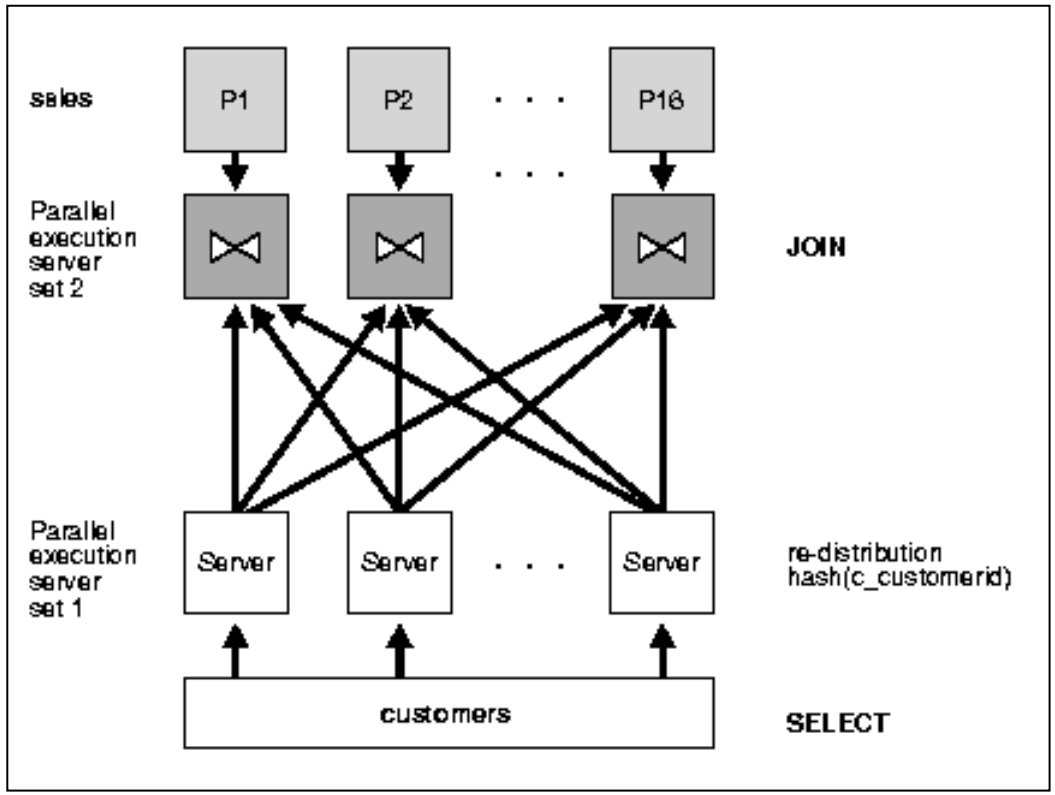
- Join mit zwei auf dem Join-Attribut partitionierten Tabellen



- Effektive Parallelisierung möglich
- Erhebliche Beschleunigung

Partitionierung und Joins

- Join bei nur einer partitionierten Tabelle
  - Dynamische Partitionierung „at runtime“



Partitionierung in Oracle

- Hash Partitionierung
  - Hash Funktion vorgegeben
  - Hash-Partitionsgrößen werden balanciert
  - DDL: Nur Anzahl, kein Split/Merge, nur +-1
- Parallelität
  - Vor 9i: Paralleles DML nur bei Partitionierung
  - 9i: Dynamische Partitionierung
    - Paralleles DML auch auf nicht-partitionierten Tabellen
    - Dynamische Partitionierung muss aber bei jeder Anfrage durchgeführt werden
  - Zusammenspiel mit Real Application Cluster (RAC)

Literatur

- [Leh03]: Kapitel 8.5.2, 8.5.3, 7.1.2
- [BG02]: Kapitel 7.3, 7.4
- Oracle 9i Dokumentation: „Data Warehousing Guide“