Data Warehousing und Data Mining

Materialisierte Sichten II

Aktualisierung und Auswahl

Ulf Leser

Wissensmanagement in der Bioinformatik

Inhalt dieser Vorlesung

- Materialisierte Sichten
- Logische Optimierung mit MV
  - Einschub: Datalog Notation
  - Query Containment
  - Depth-First Algorithmus
  - Ableitbarkeit und Query Rewriting
    - Ableitbarkeit von Bedingungen
    - Ableitbarkeit von Joins
    - Ableitbarkeit von Aggregaten
- Kostenbasierte Optimierung mit MV
- Optimierung mit Aggregaten

Anwendung 2

- Möglichkeit 3: q $\subseteq$ v (aber nicht umgekehrt)
  - Ergebnis von q vollständig enthalten in v
  - Nicht alle Tupel von v sind korrekte Ergebnisse für q, aber v berechnet alle Ergebnisse von q
  - Manche Tupel müssen aus dem Ergebnis von v entfernt werden
- Probleme
  - Vollständigkeit: v enthält alle Tupel – auch die richtigen Attribute?
  - Ableitbarkeit: Wie findet man einen „Filter“ F auf v, so dass nur die richtigen Tupel selektiert werden, also F(v) $\equiv$ q ?

Ableitbarkeit

- Jetzt hat man alle Attribute, aber zu viele Tupel
- Wie findet man die richtigen?

- Ableitbarkeit
  - Wenn $q \subseteq v$, dann gilt: $q \rightarrow h(v)$
    - h: Containment Mapping von v nach q
    - $\rightarrow$ bezeichnet hier die logische Implikation zwischen Formeln in Prädikatenlogik
  - Gesucht: Ausdruck F für den gilt: $q \equiv h(v) \wedge F$
  - Im Allgemeinen unentscheidbar
    - Wegen Unentscheidbarkeit der Prädikatenlogik
  - Aber wir betrachten nur konjunktive Anfragen

Algorithmus

- Annahmen
  - $q \subseteq v$ mit CM $h: v \rightarrow q$
  - Seien $cond(q)=\{B_1,B_2,...B_n\}$ (ohne Joins)
- Algorithmus
  - Für alle Bedingungen B_i mit $h(v) \nrightarrow B_i$
  - Wenn B_i Variablen enthält, deren Urbild bzgl. h in v nicht exportiert ist: Abbruch
    - Unzureichende Bedingung auf einer nicht-exportierten Variable
  - Sonst: $q \equiv h(v) \wedge cond(q)$
- Komplexität des einzelnen Tests
  - $O(n)$ für Bedingungen der Art: $X=5$, $X<5$, $X>5$
  - $O(n^3)$ für Bedingungen der Art: $X=Y$, $X<Y$, $X>Y$

Algorithmus für Joins

- Sei $q \subseteq v$ mit CM $h: v \rightarrow q$
  - h bildet jedes Literal aus v auf mindestens ein Literal aus q ab
- Algorithmus
  - Berechne Literale $L=(l_1,l_2,\ldots)$ aus q, die nicht im Bild von h sind
  - Prüfe alle Variable $V \in l_x$, die als Bild in h enthalten sind
    - D.h. es gibt ein $(X \rightarrow V) \in h$ (X Variable in v)
    - Wenn $X \notin exp(v)$: Abbruch
      - Da nicht kompensierbarer Join
  - Prüfe alle Elemente von $J=\{ (X=Y) \mid (X \rightarrow V) \in h \wedge (Y \rightarrow V) \in h\}$
    - Das sind Joins in q aber nicht in v
    - Wenn $X \notin exp(v)$ oder $Y \notin exp(v)$: Abbruch
- Sonst: $q \equiv h(v) \wedge L \wedge J$

Formaler

- Erinnerung
  Eine Gruppierung G ist aus einer Gruppierung H ableitbar, wenn $H \subseteq G$
- Lemma
  - *Sei v ein View mit Gruppierungsattributen G_v und q eine Query mit Gruppierungsattributen G_q und q und v haben äquivalente SPJ Klauseln*
    - *SPJ: „Select – Project – Join"*
  - *q ist direkt ableitbar aus v, $v \rightarrow q$, gdw.*
    - *$G_q \subset G_v$ und $|G_q| = |G_v| - 1$ oder*
    - *$G_q = G_v \setminus a_x \cup a_y$ mit $a_x \in G_v$, $a_y \notin G_v$ und a_y ist funktional abhängig von a_x*
  - *q ist ableitbar aus v, $v \rightarrow^* q$, gdw.*
    - *$v \rightarrow q$ oder*
    - *Es existieren Views $v_1, \ldots, v_n$ so, dass: $v \rightarrow v_1 \rightarrow \ldots \rightarrow v_n \rightarrow q$*

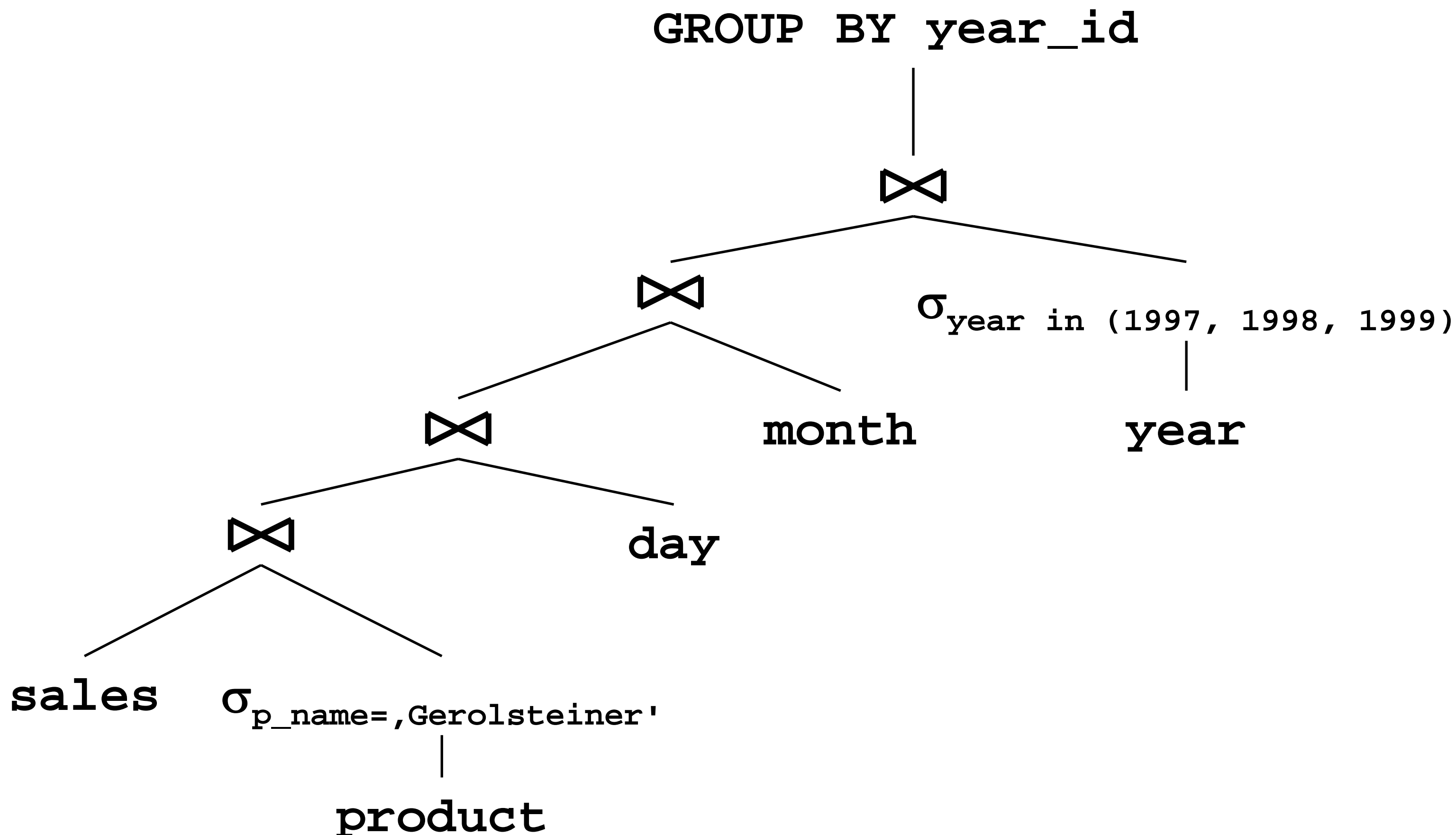
Ziel

- Mögliche nächste Frage
  - Gegeben eine Query q und mehrere materialisierte Views, die zur Ableitung von q dienen können
  - Welcher View soll verwendet werden – wenn überhaupt?
    - Der kleinste
    - Der, bei dem die Kompensationsattribute billig auszuwerten sind
    - ...
- Diese Frage ist sehr eingeschränkt
  - Wir verlangen, dass q vollständig ist v enthalten ist
- Allgemeine Frage
  - Gibt es einen Teil von q, den man mit einem v berechnen kann?
  - Welche Teile? Kostenbasierte Optimierung?

Anfrageoptimierung mit MVs

- Wo passen hier materialisierte Sichten?
- Sketch
  - Jede materialisierte Sicht ist ein potentieller Teilplan
    - Besser: Jede MV plus Kompensationen
  - Bei Bottum-Up Bewertung von Teilplänen auch materialisierte Sichten berücksichtigen
    - Matching: Gibt es für den aktuellen Teilplan einen / mehrere geeignete MVs?
    - Hinzufügen von Kompensationsoperationen notwendig?
  - Bewertung der Kosten (MV + Kompensation)
  - Auswahl des MV als Access Path, wenn er geringere Kosten als andere Teilpläne in Aussicht stellt

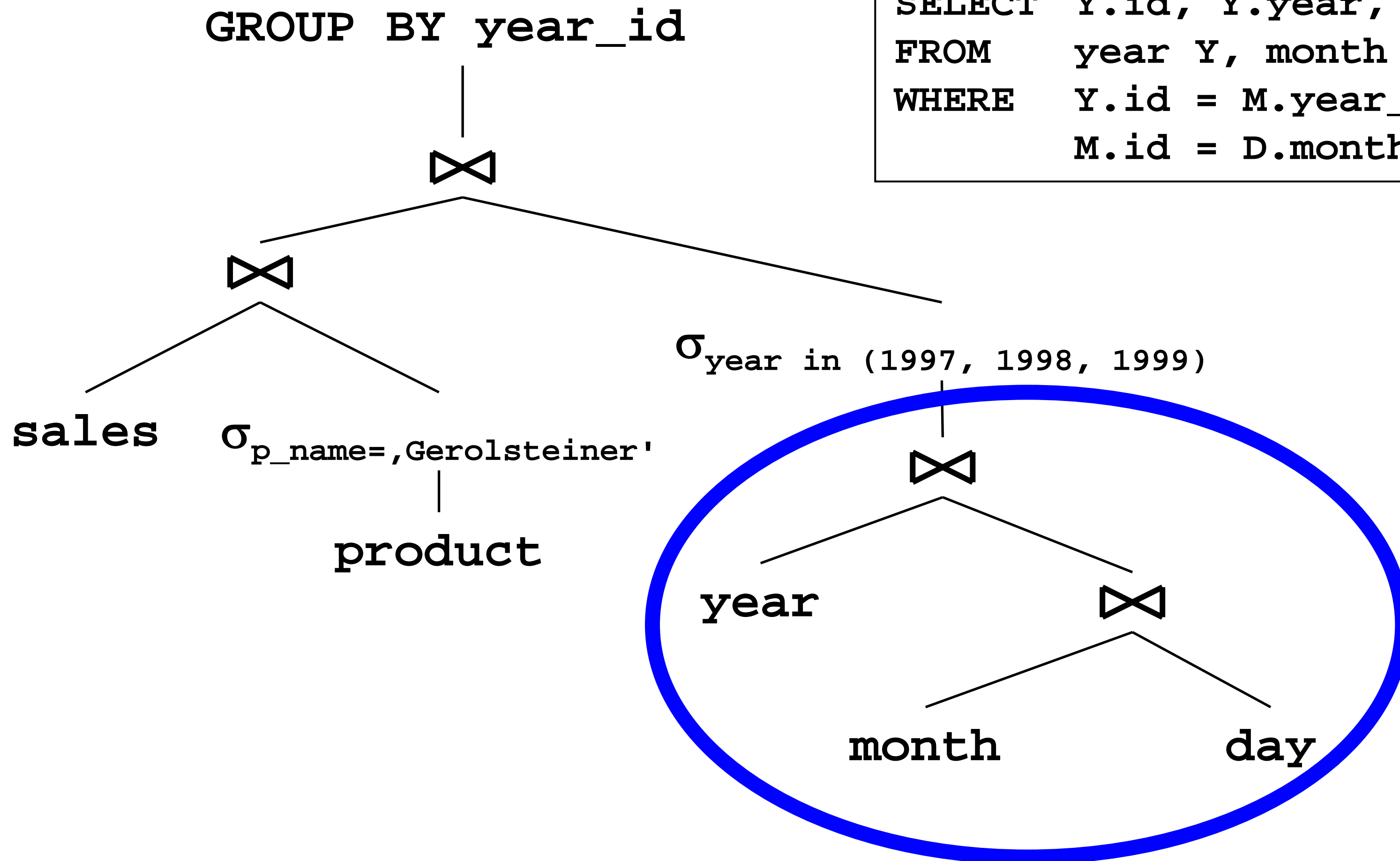
Ausführungsplan




```
CREATE MATERIALIZED VIEW v_time AS
SELECT  Y.id, Y.year, M.id, M.month, D.id, D.day
FROM    year Y, month M, day D
WHERE   Y.id = M.year_id AND
        M.id = D.month_id
```

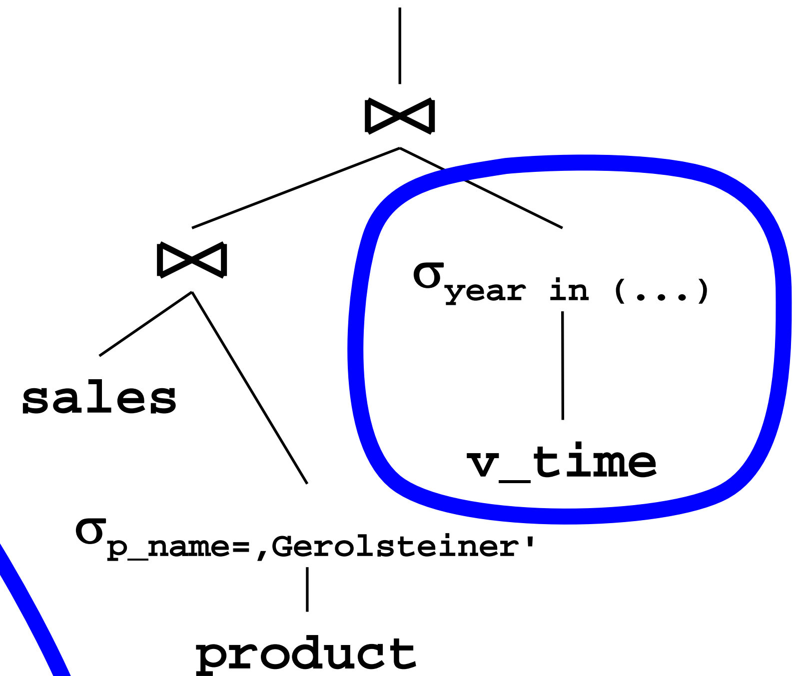
Alternativplan

```
CREATE MV v_time AS
SELECT  Y.id, Y.year, …
FROM    year Y, month M, day D
WHERE   Y.id = M.year_id AND
        M.id = D.month_id
```



Alternativen bewerten

GROUP BY year_id
⋈
⋈
sales
$\sigma_{p_name=,Gerolsteiner'}$
product
⋈
$\sigma_{year\ in\ (1997,\ 1998,\ 1999)}$
year
⋈
month
day



Einschränkungen

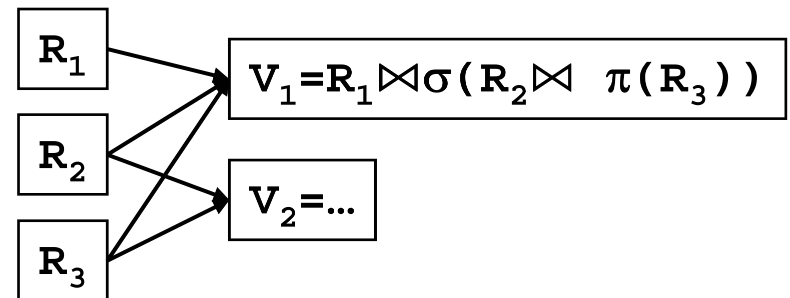
- „Matching" soll sehr schnell gehen
  - Lieber einen MV übersehen, als zu lange für Auswahl brauchen
  - Exponentielle Algorithmen vermeiden
  - Trick: MVs indexieren (insb. nach enthaltenen Relationen)
- I.d.R. werden vom Optimierer nicht alle Pläne aufgezählt
  - Typischerweise nur Left-Deep Joins
  - Gezieltes Suchen nach MVs muss eingebaut werden
- Matching erkennt in echten Systemen nicht alle Matches
  - Abhängig von Datenbanksystem
  - Beispiele für Einschränkungen
    - Keine Funktionen in Bedingungen
    - Keine Negation
    - Keine Self-joins
    - …

Inhalt dieser Vorlesung

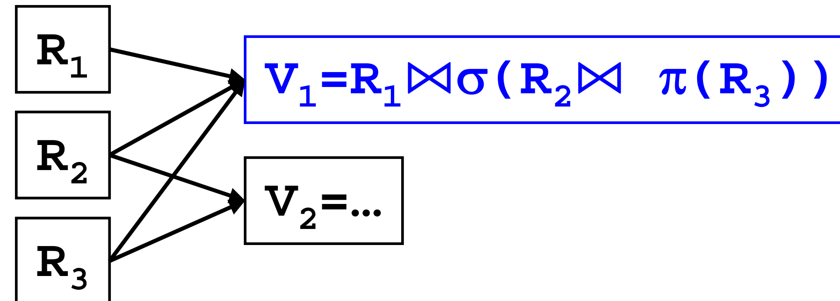
- Aktualisierung materialisierter Sichten
  - MV ohne Aggregate
  - Konsistenz in DWH
  - MV mit Aggregaten
- MV's in Oracle
- Auswahl materialisierter Sichten

Das Problem

- Materialisierte Sichten sind Anfragen auf Basisdaten
- Wenn sich Basisdaten ändern, müssen auch die gespeicherten MV angepasst werden
  - Sonst führt die Verwendung zu falschen Ergebnissen
- Möglichkeit
  - Synchrone Aktualisierung
    - Sowie sich Basisdaten ändern, MV's anpassen
    - Z.B. über Trigger
    - Teuer im laufenden Betrieb
  - Führt schnell zu inkonsistenten Daten
    - Wie?



Inkonsistente Daten

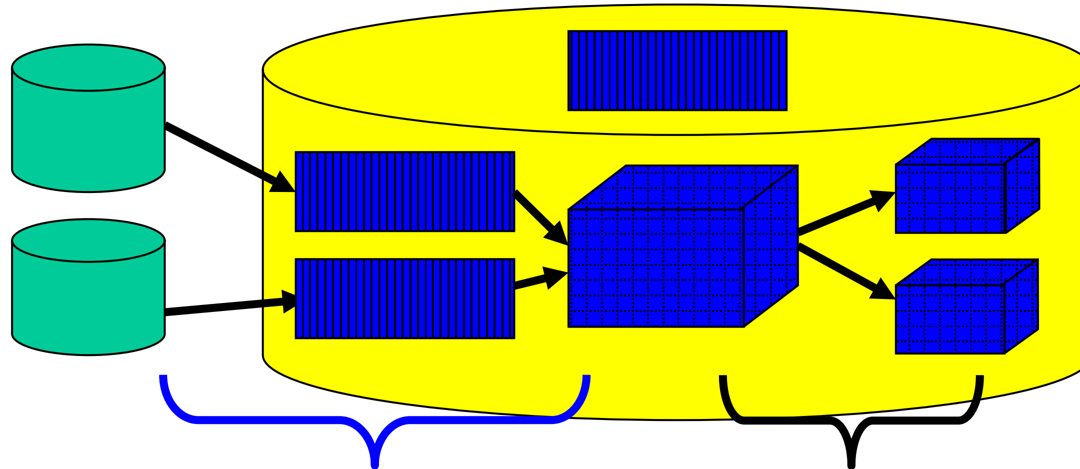


- Seien R_1 und R_2 Daten aus der Kunden- bzw. Verkaufs-DB
- Daten werden asynchron in das DWH eingespielt
  - R_1 und R_2 haben unterschiedliche Aktualisierungszeitpunkte
- Mögliches Problem
  - Daten aus R_2 bis 31.3.2007 werden eingespielt
    - Aber R_1 ist noch auf dem Stand 28.2.2007
  - Sofortige Aktualisierung von V_1 führt potentiell zu „dangling" Verkäufen
    - Verkäufe mit nicht-existenten Kunden
    - Technisch muss man auf alle Fälle die IC ausschalten
  - Inkonsistenter Zustand, bis Daten von R_1 bis zum selben Zeitpunkt wie von R_2 vorhanden sind

Möglichkeiten

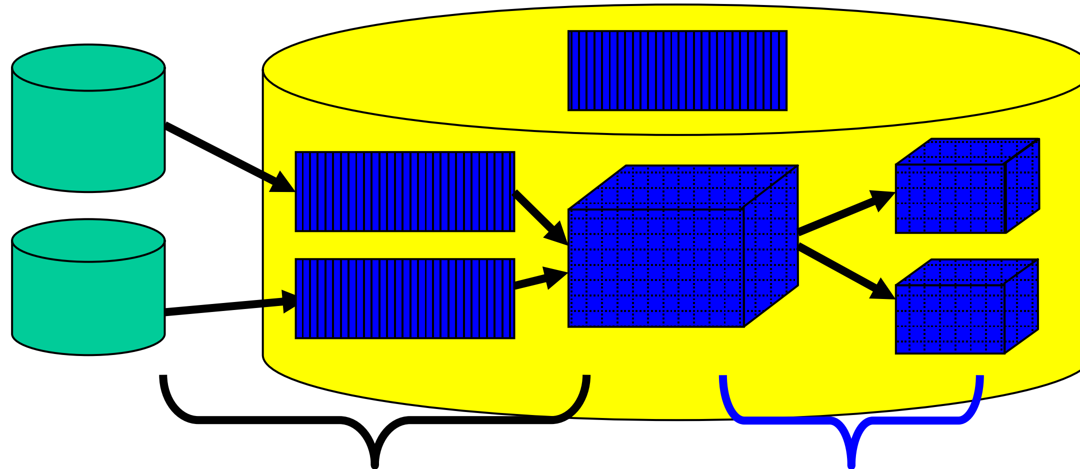
- Regelmäßige komplette Neuberechnung
  - Konsistent der Basisdaten muss extern sichergestellt sein
  - Meistens unnötig teuer
- Aktualisierung auf Anfrage
  - Inkrementelle Aktualisierung, keine Neuberechnung
  - Z.B. immer, wenn Daten aller Basisrelationen bis Zeitpunkt X vorhanden sind
  - Notwendig: Änderungen (Delta) zwischen Zeitpunkt T der letzten Aktualisierung und Zeitpunkt X der aktuellen Aktualisierung müssen ermittelt werden
    - $R^X = R^T \cup \Delta R$
    - Wir schreiben meist: $R' = R \cup \Delta R$

Ermittlung von Änderungen



- Von Quellen zum Cube
  - Änderungen kommen aus Quellen
  - Differential Snapshot Problem
  - Staging Area speichert nur Änderungen
  - Zusätzliche Spalte „change“
    - +: Tupel muss eingefügt werden
    - -: Tupel muss gelöscht werden
    - Updates als delete + insert
  - [Man kann den Cube auch als MV über den Quellen betrachten ...]

Ermittlung von Änderungen



- Von Basisrelationen zu MVs
  - Änderungen werden durch SQL erzeugt
  - Kein Differential Snapshot Problem
    - Da man die alten Versionen der Tabellen nicht aufheben kann / will
  - Besser: Logging aller Änderungen über Trigger
  - Für jede Basistabelle anlegen einer „Schattentabelle" mit zusätzlicher Spalte **`type_of_change (DEL, INS)`**
    - MV Logs

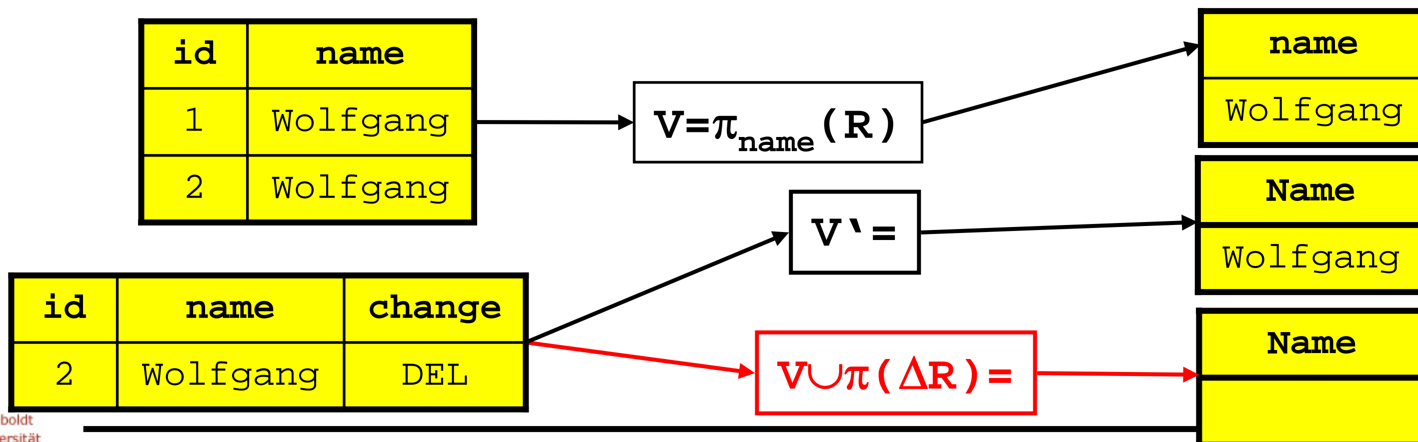
Ausgangslage

- Inkrementelle Aktualisierung von MS's

- Wir betrachten zunächst nur einzelne MVs ohne Aggregation
- Aktualisierung wird vom Benutzer angestoßen
- Alle Deltas der Basisrelationen wurden geloggt

MV: Selektion und Projektion

- MV ohne Join aber mit Selektion und/oder Projektion
- Selektion
  - Ausnutzung der Distributivität der Selektion
  - Sei also $V = \sigma(R)$
  - $V' = \sigma(R') = ?$
  - $V' = \sigma(R') = \sigma(R \cup \Delta R) = \sigma(R) \cup \sigma(\Delta R) = V \cup \sigma(\Delta R)$
  - Sehr schön: Wir müssen nur das Delta betrachten
- Projektion
  - Gilt $V' = \pi(R') = \pi(R \cup \Delta R) = \pi(R) \cup \pi(\Delta R) = V \cup \pi(\Delta R)$?



MV mit Projektionen

- Projektion ist nicht distributiv bei
  - Mengensemantik von (theoretischem) SQL
    - Achtung: Auch in „normalen“ Datenbanken implementiert man das nicht so leicht
    - Einspielen des Logs
      - `DELETE from R`
`WHERE name=‚Wolfgang‘`
    - Löscht alle Tupel – berechnet also das falsche Ergebnis
  - Anfragen mit DISTINCT
- Ausweg
  - Materialisierte Sichten sollten immer einen Schlüssel mitführen
    - Keine Probleme mit Duplikaten mehr
  - Sonst: Zählerspalte Z pro Tupel im MV einführen
    - Bei INS (+) erhöhen
    - Bei DEL (-) erniedrigen
    - Tupel erst löschen, wenn Z=0

MV mit Joins

- Wir beginnen mit nur einem Join

- $V' = R' \bowtie S' = (R \cup \Delta R) \bowtie S'$
  $= (R \bowtie S') \cup (\Delta R \bowtie S')$
  $= (R \bowtie (S \cup \Delta S)) \cup (\Delta R \bowtie S')$
  $= (R \bowtie S) \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S')$
  $= V \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S')$
  $= V \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie (S \cup \Delta S))$
  $= V \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S) \cup (\Delta R \bowtie \Delta S)$

- Problem: Stimmt nicht

Gegenbeispiel

$V \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S) \cup (\Delta R \bowtie \Delta S)$

id	Name	Change
1	A	
2	B	
3	C	
2	A	DEL
3	B	DEL
4	D	INS
5	E	INS

$\bowtie$

id	Age	Change
1	21	
2	22	
4	24	
2	22	DEL
3	23	INS
4	24	DEL
5	25	INS

=

id	Change
1	
2	

$\cup$

id	Change
2	DEL
3	INS

$\cup$

id	Change
2	DEL
4	INS

$\cup$

id	Change
2	DEL
5	INS

=

id	Change
1	
2	DEL
3	INS
4	INS
5	INS

$\neq$

id	Change
1	
5	INS

Berechnung von ΔV

	INS	**DEL**	**Neutral**
INS	INS	Verwerfen	INS
DEL	Verwerfen	DEL	DEL
Neutral	INS	DEL	Verwerfen

- Auswirkungen von Joins von INS und DEL
- Verwerfen heißt: Nichts tun, keine Änderung vollziehen
- Das Problem liegt bei den DEL/INS Kombinationen
  - Tupel 3 im Beispiel: Wird eingefügt durch ($R \bowtie \Delta S$) und nicht gelöscht durch ($\Delta R \bowtie \Delta S$)
  - Problem tritt nur bei den Joins von Deltas auf

MV mit Joins

- Also: Joins von Deltas vermeiden

- $V' = R' \bowtie S' = (R \cup \Delta R) \bowtie S'$

  …

  $= V \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S')$

- Unschön: Wir benötigen den alten Zustand R (bzw. S) zur Berechnung von ΔV
  - Das ist teuer – alle Relationen müssen doppelt gehalten werden
  - (Wunsch: R/S immer aktualisieren und nur die Logs zusätzlich halten)

Beispiel

$V \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S')$

id	Name	Change
1	A	
2	B	
3	C	
2	A	DEL
3	B	DEL
4	D	INS
5	E	INS

$\bowtie$

id	Age	Change
1	21	
2	22	
4	24	
2	22	DEL
3	23	INS
4	24	DEL
5	25	INS

=

id	Change
1	
2	

$\cup$

id	Change
2	DEL
3	INS

$\cup$

id	Change
3	DEL
5	INS

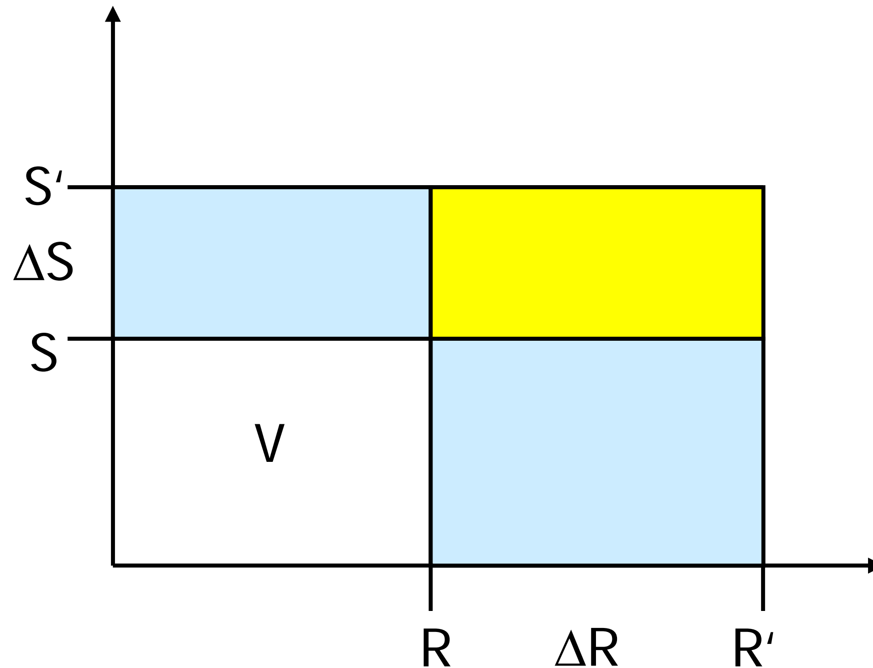
=

id	Change
1	
5	INS

Vorhalten zweier Zustände?

- Möglichkeit 1
  - R wird bei Aktualisierung von V „zurückgerechnet“: **R=R‘-ΔR**
- Alternative : Wir verzichten auf Löschoperationen
  - Das ist in DWH durchaus realistisch
  - Dann gilt wieder das Distributivgesetz
  - Damit
    - $V' = R' \bowtie S' = (R \cup \Delta R) \bowtie S'$
      $= V \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S) \cup (\Delta R \bowtie \Delta S)$
  - Noch kein Gewinn: Wir haben den neuen Zustand und die Deltas
  - Was tun?
    - $V = R \bowtie S = (R'-\Delta R) \bowtie (S'-\Delta S)$
      $= (R' \bowtie S') - (R' \bowtie \Delta S) - (\Delta R \bowtie S') \cup (\Delta R \bowtie \Delta S)$
    - $V' = R' \bowtie S'$
    - $\Delta V = V' - V =$
      $= (R' \bowtie \Delta S) \cup (\Delta R \bowtie S') - (\Delta R \bowtie \Delta S)$

Verdeutlichung



- Vorwärts: $V' = V \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S) \cup (\Delta R \bowtie \Delta S)$
  - Alle Quadrate werden einzeln addiert
- Rückwärts: $\Delta V = (R' \bowtie \Delta S) \cup (\Delta R \bowtie S') - (\Delta R \bowtie \Delta S)$
  - Gelbes Quadrat wird zwei mal addiert und einmal subtrahiert

MV mit vielen Joins (ohne Löschen)

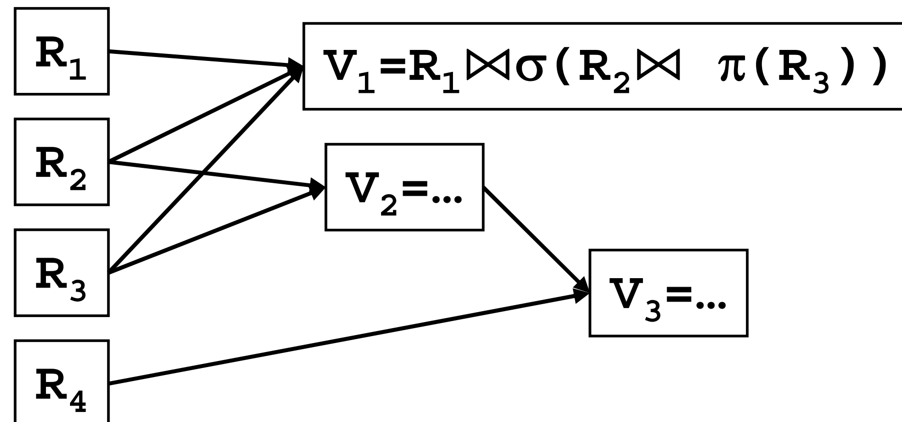
- Das Distributivgesetz gilt genauso
- Zwei Joins: $V = R \bowtie S \bowtie T$
  - $V = R \bowtie S \bowtie T$
    $= (R`-\Delta R) \bowtie (S`-\Delta S) \bowtie (T`-\Delta T)$
    $= (R` \bowtie S` \bowtie T`) - (R` \bowtie S` \bowtie \Delta T) - (R` \bowtie \Delta S \bowtie T`) … - (\Delta R \bowtie \Delta S \bowtie \Delta TS)$
- Viele Joins: $V = R \bowtie S \bowtie T … \bowtie Z$
  - Jeder Term ist eine Anfrage, die man auswerten muss
  - Wie viele Terme bei n Joins?
    - 2^{n+1}

Inhalt dieser Vorlesung

- Aktualisierung materialisierter Sichten
  - MV ohne Aggregate
  - Konsistenz in DWH
  - MV mit Aggregaten
- MV's in Oracle
- Auswahl materialisierter Sichten

Konsistenz in DWH

- Bisher sind wir immer von einem einzigen MV ausgegangen
- In realen DWH ist das Szenario komplizierter
  - Zyklische Definitionen sind natürlich immer zu vermeiden
- In welchem Zustand befindet sich das DWH während einige Aktualisierungen laufen?
  - Und in welcher Reihenfolge sollen die laufen?

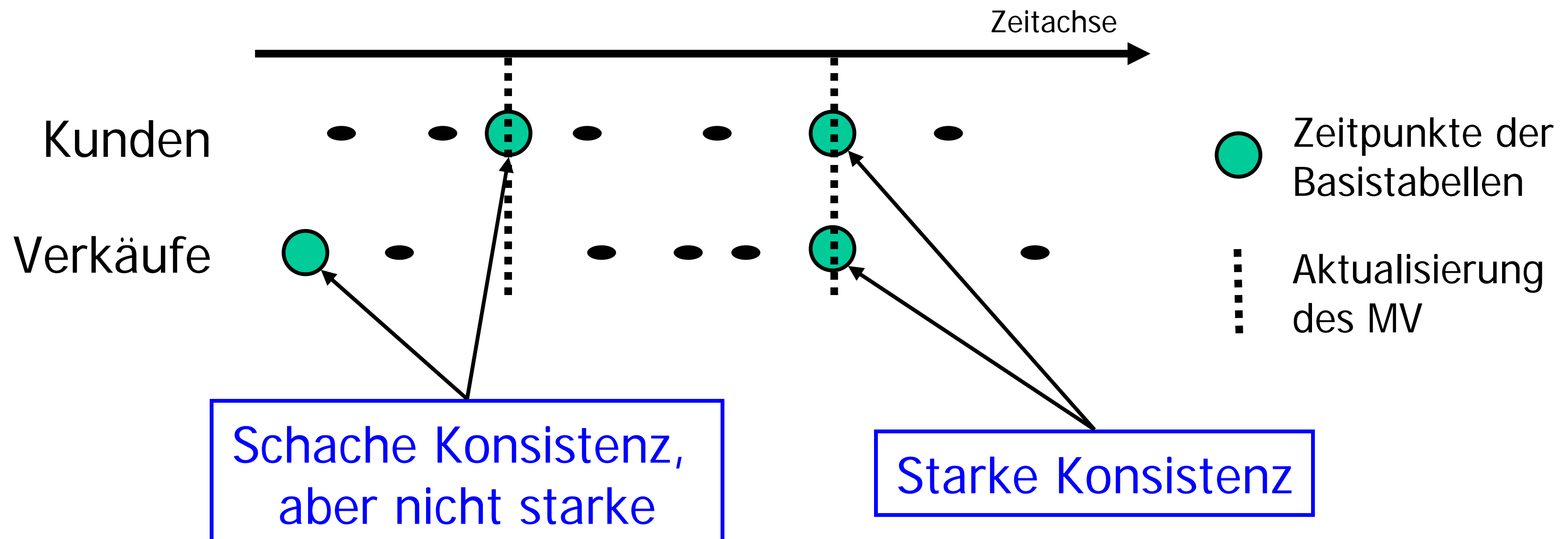


Lokale und globale Konsistenz

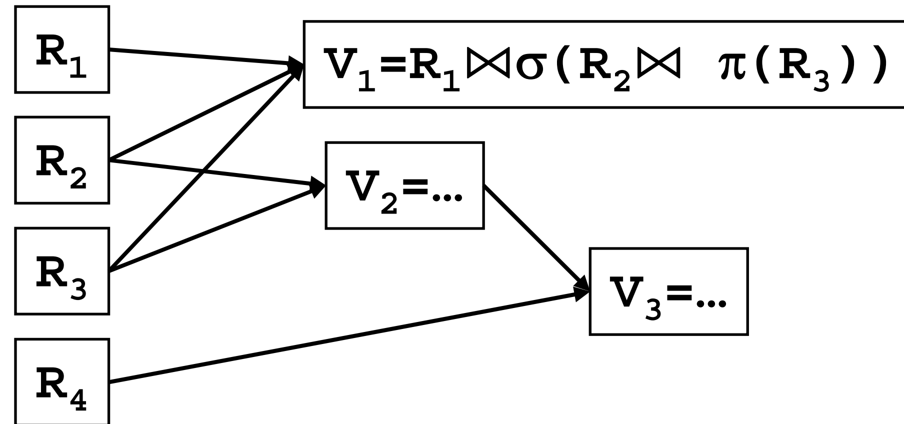
- Wunschdenken
  - Alle Quellen exportieren nur in-sich konsistente Zustände
  - Wenn verschiedene Quellen zu einem Zeitpunkt X Exportfiles erstellen, dann ist der Zustand über alle Exportfiles konsistent
    - „Globale Konsistenz“
    - Das ist eine Annahme; man muss das geeignet sicherstellen
    - Das DWH kann diese Bedingung nicht überwachen
    - Alternative: Einsatz von Transaktionsmonitoren, 2PC, …
- Folge: Bei der Aktualisierung von MV dürfen die Delta-Files erst nach Abschluss aller betroffenen Transaktionen angefasst werden
- Im allgemeinen exportieren die Quellen aber zu unterschiedlichen Zeitpunkten

Konsistenzgrade

- Schwache Konsistenz
  - Alle Views sind bzgl. jeder Basisrelation konsistent. Damit ist nicht sichergestellt, dass ein global gültiger Zustand erreicht ist
- Starke Konsistenz
  - Alle Views entsprechen einem global gültigen Zustand
  - Erfordert synchronisierte Exports / Logfiles



Aktualisierung vieler Views



- In welcher Reihenfolge aktualisiert man die Views?
  - Erfolgt die Aktualisierung jedes Views als eigene Transaktion, so haben die Views zwischen diesen Transaktionen inkonsistente Zustände
    - Beispiel: Erst V1 aktualisieren, dann Zugriff auf V1 und V2
  - Packt man alles in eine Transaktion, steht die Datenbank für lange Zeit
- Besser: Simple Painting [ZWG97]
  - Zerlegung der Deltas in „Mini-Deltas"
  - Einspielen eines Mini-Deltas in alle Views ist eine Transaktion
  - Minimierte Sperrzeiten, keine inkonsistenten Zustände

Inhalt dieser Vorlesung

- Aktualisierung materialisierter Sichten
  - MV ohne Aggregate
  - Konsistenz in DWH
  - MV mit Aggregaten
- MV's in Oracle
- Auswahl materialisierter Sichten

MV mit Aggregation

- Wir betrachten nur algebraische Funktionen (SUM, COUNT, ...)
- Beispielanfrage
  - ```
    SELECT   T.day, P.name, R.region, sum(amount*price)
    FROM     sales S, product P, time T, region R
    WHERE    …
    GROUP BY T.day, P.name, R.region
    ```
- Einfache Variante: Wie gehabt
  - Log-File mitschneiden
  - Zur Aktualisierung Tupel für Tupel durchgehen
  - Entsprechende Gruppe suchen und Änderung addieren/subtrahieren
  - Wenn Gruppe noch nicht vorhanden: neues Tupel in den MV einfügen
- Problem: Langsam, viele Änderungen notwendig
  - Z.B. Einspielen von 100.000 Verkäufen eines Tages
  - MV muss die ganze Zeit gesperrt bleiben
- Besser: Log-File komprimieren
    ```

Kompression Log-Files

```
SELECT     …,   sum(amount*price)
FROM       sales S, …
WHERE      …
GROUP BY T.day, P.name, R.regio
```

- Idee: Gruppierung schon im Log-File vornehmen
  - Übernahme der Gruppierungsattribute aus der Anfrage

day	name	region	customer	amount	change
12	100	3	1	100	INS
12	100	3	3	20	DEL
12	100	3	1	25	INS
11	100	2	1	5	INS
11	100	2	3	20	INS
…	…	…	…	…	

  - Unterscheidung zwischen INS/DEL in Gruppierung codieren
  - Erster Schritt
    - ```
      SELECT day, name, regio, decode('INS',+1,-1), sum(amount)
      FROM   log
      GROUP BY day, name, regio, change
      ```
      ```

2. Schritt

```
SELECT   …,   sum(amount*price)
FROM     sales S, …
WHERE    …
GROUP BY T.day, P.name, R.regio
```

- Ergibt

day	product	region	SUM	change
12	100	3	125	+1
12	100	3	20	-1
11	100	2	25	+1

- 2. Schritt: Auflösung von INS / DEL
  - ```
    SELECT day, name, regio, sum(amount*change)
    FROM   log
    GROUP BY day, name, regio
    ```

day	product	region	SUM
12	100	3	105
11	100	2	25

  - Enthält alle relevanten Änderungen in komprimierter Form
    ```

3. Schritt

- Einbringen der Änderungen pro Gruppe in den MV
  - Komprimierte Log-Tabelle Tupel für Tupel durchgehen
  - Gruppe in MV suchen und
    - .. gefunden: Änderungen addieren
    - Wenn sich die Änderungen zu 0 addieren: Gruppe löschen
    - ... nicht gefunden: Neue Gruppe in MV einfügen
  - Jede Gruppe wird nur einmal geändert
- Achtung: 0 ≠ null
  - SQL unterscheidet zwischen einer Gruppe, deren Summe 0 ergibt (z.B. Verkäufe +20, +10 und -30), und einer Gruppe, die keine Verkäufe hat
    - Ist im Ergebnis nicht repräsentiert
  - Diese Unterscheidung geht bei obiger Methode verloren
    - Wurde der einzige Kauf einer Gruppe storniert?
    - Haben sich verschiedene Verkäufe nur zu 0 addiert?
  - Lösung: Zählspalten einführen
    - Wie viele Tupel hat die Gruppe?

Self Maintainability

- Solche Views heißen „self maintainable“
  - Aktualisierung des Views benötigt nur den alten View und das LOG File
  - Kein Rückgriff auf Basisrelation notwendig
  - Nützlich: Z.B. muss man den Cube selber nicht mehr vorhalten, sondern aktualisiert die MV direkt aus den Change-Files der Quellen
- Die wenigsten Views sind self maintainable
  - Erfordert starke Einschränkung der erlaubten Prädikate
    - Kritisch sind Joins, DISTINCT, Projektionen, ...
    - Kritisch sind holistische Aggregationen
    - Gut sind Schlüssel in den Views, einfache Selektionen, ...
  - Reichhaltige Literatur vorhanden

Inhalt dieser Vorlesung

- Aktualisierung materialisierter Sichten
- MV's in Oracle
  - Anlegen von materialisierten Sichten
  - Aktualisierungstechniken
  - Optimierung / Rewriting mit materialisierten Sichten
- Auswahl materialisierter Sichten

Definition von MVs

```
CREATE MATERIALIZED VIEW product_sales_mv
BUILD IMMEDIATE
REFRESH FAST
ENABLE QUERY REWRITE
AS
SELECT P.prod_name, SUM(S.amount), COUNT(*)
FROM   sales S, products P
WHERE  S.product_id = P.id
GROUP BY P.prod_name;
```

- Materialisierte Sicht wird als Tabelle angelegt
  - Indexierung, Partitionierung, STORAGE Klauseln, etc.
- „Query Rewrite" muss explizit erlaubt werden

Reengineering

- Materialisierte Sichten können über existierende Tabellen gelegt werden

```
CREATE MATERIALIZED VIEW sum_sales_tab
ON PREBUILT TABLE
ENABLE QUERY REWRITE
AS
...
```

- Nachträgliche Benutzung der Rewrite / Aktualisierungsmechanismen

Aktualisierungsstrategien

- Wann wird aktualisiert
  - ON DEMAND: Nur bei Aufruf von speziellen Funktionen im Package DBMS_MVIEW
    - `dbms_mview.refresh (mv), refresh_all, refresh_dependent(tab)`
  - ON COMMIT: Beim Abschluss jeder Transaktion, die mindestens eine Basistabelle verändern
    - Bestmögliche Aktualität
    - Schwieriger Konsistenzbegriff
- Wie wird aktualisiert
  - COMPLETE: Neuberechnung des Views bei Änderungen an mindestens einer Basistabelle
  - FAST: Inkrementelles Nachführen von Änderungen
  - FORCE: Ausführung von FAST, wenn möglich; sonst COMPLETE

Kombinationen

	On commit	On demand
Complete		OK
Fast (incremental)	OK	

- Grün sind die sinnvollen Kombinationen
- Möglich sind alle

Inkrementelle Aktualisierung (fast)

- Erfordert das explizite Anlegen eines MV Logs

```
CREATE MATERIALIZED VIEW LOG ON sales
WITH SEQUENCE, ROWID
(prod_id, cust_id, time_id, channel_id,
 promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

- Legt Trigger auf Mastertabellen an
  - MV-Log speichert Deltas von Änderungen
  - FAST Refresh spielt diese Deltas in MV ein
- Diverse Einschränkungen
  - Benötigt immer ein count(*) bei Gruppierung – Zählvariable
  - Rowids müssen immer dabei sein

Nested MV

- Die Definition eines MV kann einen anderen MV als Basistabelle benutzen
- Dann muss (für FAST refresh) ein MV Log auf dem MV angelegt werden
  - Und der nested MV muss Joins oder Aggregate enthalten (?)
- Vorsicht: Unterschiedliche Refresh-Optionen innerhalb eines solchen MV Baums führen zu eigenwilligen Effekten

Rewrite Methoden

- Oracle versucht Rewriting auf mehrere Arten (V9.2)
- Vorsicht: Der benutze MV kann nicht aktuell sein
  - Optionen STALE_TOLERATED etc.
- Text Match
  - „Full Text Match“ und „Partial Text Match“
  - Rein syntaktischer Match
    - Keine Unterscheidung von Groß-/ Kleinschreibung und Leerzeichen
    - Aber: Reihenfolge von Bedingungen, andere Tabellenaliase, etc.
  - Keine logische Implikation, kein Mapping, etc.
- General Query Rewrite
  - Nicht anwendbar bei „Complex materialized views“
  - Unterschiedliche Methoden ja nach Art des MV
    - Only join – only aggregate – join and aggregate - complex
  - Viele Einschränkungen – Dokumentation
- Seit 10gR2 können auch mehrere MVs für eine Query verwendet werden

Inhalt dieser Vorlesung

- Aktualisierung materialisierter Sichten
- MV's in Oracle
- Auswahl materialisierter Sichten
  - Statische Auswahl: Gewinn versus Platzverbrauch
  - Aktualisierungskosten
  - Dynamische Auswahl

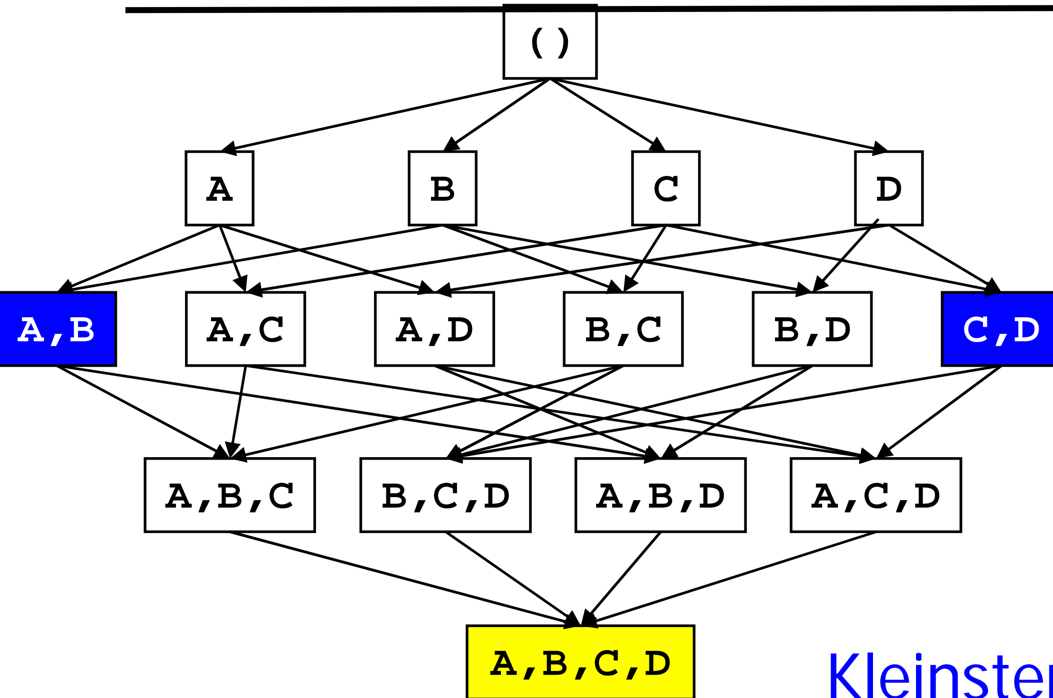
Auswahl materialisierter Sichten

- Wir betrachten nur MV mit Aggregation
  - Berechnung von Aggregaten sind tendenziell teuer
  - Wiederholtes Scannen sehr großer Tabellen
  - Vorberechnung ist daher attraktiv
- Problem: Der vollständige CUBE benötigt viel zu viel Platz
  - Beispiel: 5 Dimensionen a 50 Knoten
    - Gleichverteilung der Werte auf alle Knoten
  - Die höchste Granularität hat $\sim 50^5 = 3*10^8$ Tupel
  - Dazu: $\sim 5*50^4+20*50^3+20*50^2+5*50 = 3.3*10^7$ Tupel mit weniger Gruppierungsattributen
  - Tatsächliche Zahlen abhängig vom Füllgrad des Cube ab
    - Wie schätzen?

Auswahlkriterien

- Welche MV soll man nun berechnen?
  - Die, die oft in Anfragen verwendet werden
  - Die, aus denen man viele Anfragen schnell ableiten kann
  - Die, die viel Arbeit in Anfragen sparen
  - Die, die wenig Platz brauchen
  - Die, die man schnell aktualisieren kann
- Also: Auswahl abhängig von
  - Workload
  - Struktur des Aggregationsgitter
  - Größe der MV
  - Verfügbarer Platz
  - Aktualisierungskosten
- Ohne Größenbeschränkung ist das Problem trivial
  - Warum?

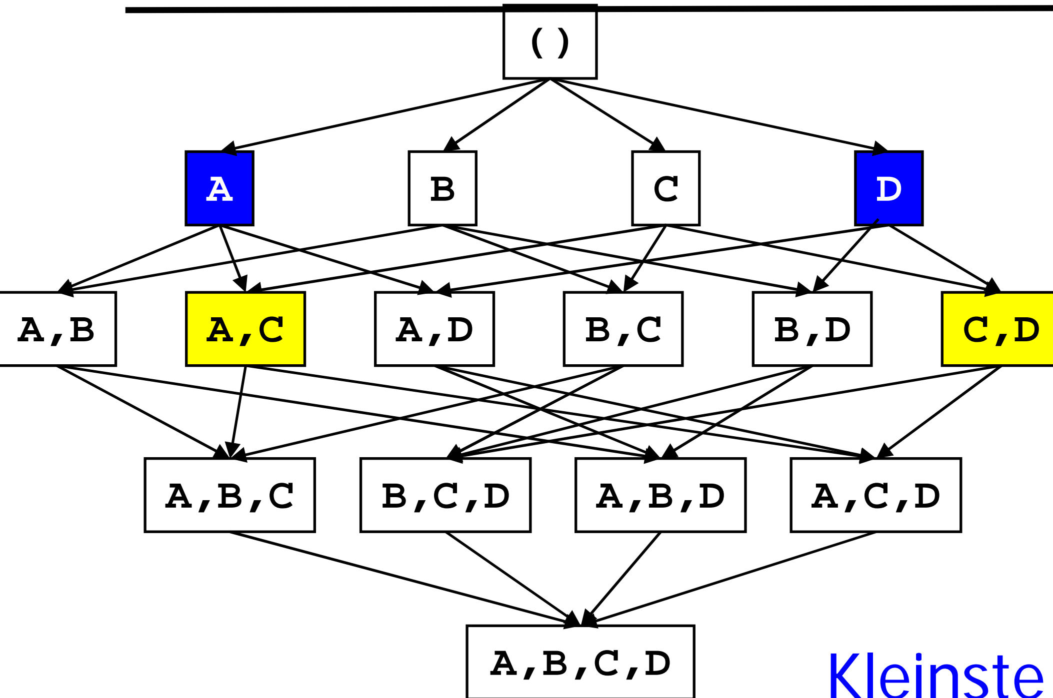
Kleinster gemeinsame Vorfahre



Kleinster gemeinsamer Vorfahr

- I.A mehrere zur Auswahl
- Verlangt MV mit hoher Auflösung – braucht Platz

Kleinste direkte Vorfahren



Kleinste direkte Vorfahren

- I.A. mehrere zur Auswahl
- MV mit niedrigerer Auflösung – weniger Platz
- Dafür mehrere MV notwendig

Statische Auswahl mit Platzbeschränkung

- Gegeben
  - Workload: Anfragen $Q=\{q_1,\dots,q_n\}$ mit Frequenzen $h_1,\dots,h_n$
  - Aggregationsgitter mit $A=\{a_1,\dots,a_k\}$ Kombinationen
    - $|A|=2^d$ für d Dimensionen (ohne Hierarchien)
    - Jedes a_i hat $|a_i|$ Tupel
  - Menge an verfügbarem Platz: m
  - Geschätzte Kosten $c(q_i,a_j)$ zur Ableitung von q_i aus MV a_j
- Wir ignorieren zunächst
  - Aktualisierungskosten
  - Änderungen der Workload
- Gesucht: Optimale Menge M von MV

Problem

- Definition
  Gegeben eine Workload Q, |Q|=n, mit Anfragefrequenzen h, ein Aggregationsgitter A, eine Kostenmatrix c und eine Konstante m. Das MV-Selektion Problem sucht die Menge M⊆A, die den folgenden Ausdruck minimiert

$$c(Q,M) = \sum_{i=1 \ldots n} h_i * \min_{a_j \in M}\left(c(q_i, a_j)\right)$$

unter Beachtung der Nebenbedingung

$$\sum_{a_i \in M} |a_i| \leq m$$

Komplexität?

- Kann auf das Rucksach-Problem reduziert werden
- Das Problem ist NP-hart
- Wir brauchen Heuristiken

Greedy-Heuristik [HRU98]

- Grundidee: Iteratives Greedy-Verfahren
  - MV werden nacheinander ausgewählt
  - Auswahl erfolgt nach dem „Nutzen pro Platz“ des neuen MV
- Definition

  Gegeben eine Anfragemenge Q und eine Menge M von MV. Der Nutzen eines MV $a \notin M$ ist

$$B(a,Q,M) = c(Q,M) - c(Q, M \cup a)$$

  Der Benefit-per-Space (BS) von a ist

$$BS(a,Q,M) = \frac{B(a,Q,M)}{|a|}$$

Humboldt Universität Informatik

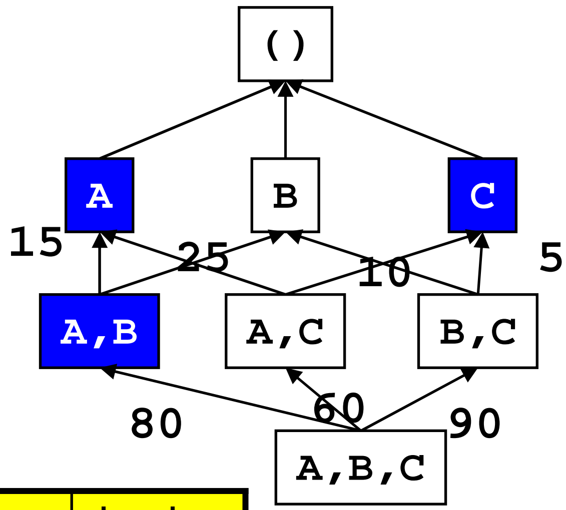
Sketch des Algorithmus

```
Input: Q, A, m;
Output: M;                    \\ Menge ausgewählter MV
while m>0
  opt := 0;                   \\ Optimaler BS bisher
  for all a∈A-M
    b := 0;                   \\ Nutzen von a
    for all q∈{Anfragen ableitbar aus a}
      b =+ c(q,M) - c(q,M∪a);  \\ Nutzen von a für q
    end for;
    if ((b\|a|)>opt)
      opt := b;               \\ Neues Optimum
      a_opt := a;
  end for;
  if ((m-|a_opt|)>0)
    M := M ∪ a_opt;           \\ Menge selektierter MV anpassen
    m =- |a_opt|;           \\ Platz anpassen
  else
    m := 0;                   \\ Terminieren
end while;
return M;
```

Eigenschaften

- Nutzt den Platz nicht perfekt
  - Terminiert, wenn der nächste beste MV nicht mehr passt
  - Verbesserung: Abarbeitung einer nach Platz und BS sortierten Liste von MV
- Theorem
  Der vorherige Algorithmus liefert eine Menge M von MV, deren Nutzen mindestens (63-x)% des optimalen Nutzen beträgt
  - *X: Größe des größten MV in Prozent der Gesamtgröße aller MV*
- Beweis: Literatur
- Beispiel:
  - Wenn kein View mehr als 10% der Gesamtgröße aller MV benötigt, garantiert der Algorithmus eine Güte von 53%

Beispiel

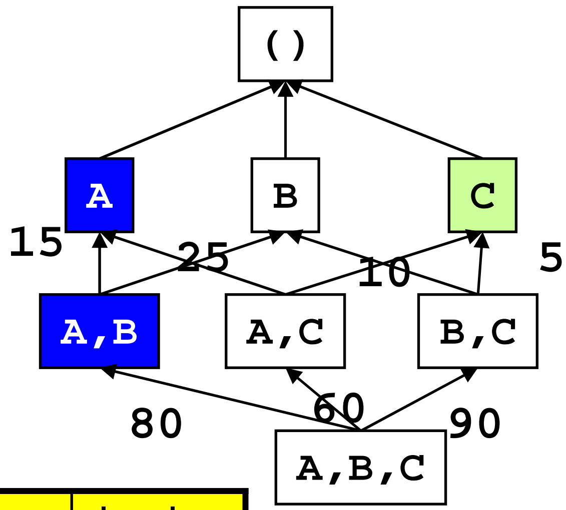


- Annahmen
  - m=200
  - Anfragefrequenzen alle 1
  - Kosten zur Berechnung ohne MV: jeweils 200

a_i	$\|a_i\|$
()	1
A	24
B	10
C	5
A,B	380
B,C	120
A,C	180
A,B,C	2200

a_i	B(A)	B(A,B)	B(C)	BS(a_i)
()	0	0	0	0
A	200	0	0	8.3
B	0	0	0	0
C	0	0	200	40
A,B	185	200	0	1
B,C	0	0	195	1.6
A,C	175	0	190	2
A,B,C	115	120	130	0.2

Beispiel 2

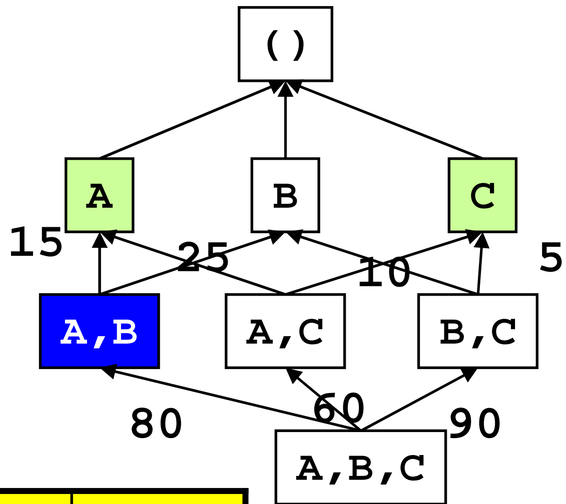


a_i	$\|a_i\|$
()	1
A	24
B	10
C	5
A,B	380
B,C	120
A,C	180
A,B,C	2200

- Annahmen
  - m=200
  - Anfragefrequenzen alle 1
  - Kosten zur Berechnung ohne MV: jeweils 200

a_{i1}	B(A)	B(A,B)	B(C)	$BS(a_i)$
()	0	0	0	0
A	200	0	0	8.3
B	0	0	0	0
A,B	185	200	0	1
B,C	0	0	0	0
A,C	175	0	0	1
A,B,C	115	120	0	0.1

Beispiel 3



a_i	$\|a_i\|$
()	1
A	24
B	10
C	5
A,B	380
B,C	120
A,C	180
A,B,C	2200

- Kein weiterer MV passt in m
  - Und (B,C) bringt nichts
- Kosten für eine Workload
  - Ohne MV: 600
  - Wie berechnet (A und C): 200
  - Materialisierung von (A,B,C): 165
    - Aber nicht genug Platz

Verbesserung [SDN98]

- Komplexität des Verfahrens?
  - O(|M|*|A|*|Q|)
  - Wenn man 20 Views für 50 Queries bei 5 Dimensionen a 4 Stufen annimmt: $20*50*2^{20}=1E9$
- Schnellere Methode: Pick-by-size
  - MV nach Größe sortieren (O(|A|*log(|A|)))
  - Greedy MV so lange auswählen, bis m erschöpft ist
- Liefert unter bestimmten Vorausstetzungen gleiche Güte
  - Wenn alle Knoten ihre Tupel relativ gleichmäßig auf alle Nachkommen verteilten
  - Das ist z.B. bei Unabhängigkeit aller Klassifikationsknoten gegeben

Inhalt dieser Vorlesung

- Aktualisierung materialisierter Sichten
- MV's in Oracle
- Auswahl materialisierter Sichten
  - Statische Auswahl: Gewinn versus Platzverbrauch
  - Aktualisierungskosten
  - Dynamische Auswahl

Einbeziehung von Aktualisierungskosten

- Die bisherige Kostenfunktion ist monoton

$$B(\{u,v\},Q,M) \leq B(u,Q,M) + B(v,Q,M)$$

- Das garantiert, dass man iterativ vorgehen kann
  - Gilt Monotonie nicht, kann (A,B) zu teuer sein, aber (A,B,C) plötzlich wieder billig
- Ohne Monotonie entfällt die (63-X)% Garantie
- Berücksichtigt man aber Aktualisierungskosten als Kostenmaß, gilt die Monotonie nicht mehr
  - Wenn man (A,B,C) aktualisiert, kriegt man (A,B) und (A) praktisch umsonst dazu
- Keine ähnlich effektiven Heuristiken bekannt
  - Also: Random Restart mit Hill Climbing, Tabu Search, Simulated Annealing, etc.

Dynamische Auswahl

- Bisher optimieren wir MV-Auswahl für eine feste Menge von Anfragen
- Besser: Das System soll sich laufend anpassen
- Das entspricht einem materialisierten Cache
  - Benötigt natürlich Warm-Up
- Beispiel: Watchman
  - Speicherung von Anfrageergebnissen zusammen mit der Anfragedefinition
  - „Semantisches Caching"
  - Einbeziehung des erwarteten Nutzen in die Cache-Verdrängungsstrategie
    - Nutzen: Gesparte Kosten bei weiteren Ausführungen derselben Anfrage

Literatur

- [Leh03]: 6.1, 6.2 (Konsistenz), 7.4 (Aktualisierung), 7.3 (Auswahl)

- [GM95] Gupta, A. and Mumick, I. S. (1995). "Maintenance of Materialized Views: Problems, Techniques and Applications." *IEEE Quarterly Bulletin on Data Engineering* **18**(2): 3-18.
- [BDD+98] Bello, R. G., Dia, K., Downing, A., Feenen Jr, J., Norcott, W. D., Sun, H., Witkowski, A. and Ziauddin, M. (1998). "Materialized Views in ORACLE". 24th Conference on Very Large Database Systems, New York. pp 659-664.
- [CKPS95] Chaudhuri, S., Krishnamurthy, R., Potamianos, S. and Shim, K. (1995). "Optimizing Queries with Materialized Views". 11th Int. Conference on Data Engineering, Los Alamitos, CA, IEEE Computer Soc. Press. pp 190-200.
- [GL01] Goldstein, J. and Larson, P.-A. (2001). "Optimizing Queries Using Materialized Views: A Practical, Scalable Solution". ACM SIGMOD Int. Conference on Management of Data 1998, Seattle, Washington.
- [SDN98] Shukla, A., Deshpande, P. and Naughton, J. F. (1998). "Materialized View Selection for Multidimensional Datasets". 24th Conference on Very Large Database Systems, New York. pp 488-499.
- [ZCL+00] Zaharioudakis, M., Cochrane, R., Lapis, G., Pirahesh, H. and Urata, M. (2000). "Answering Complex SQL Queries Using Automatic Summary Tables". SIGMOD Conference, Dallas, Texas. pp 105-116.