Informationsintegration

Answering Queries using (only) Views

Ulf Leser

Inhalt dieser Vorlesung

- Answering Queries using Views
  - Längenbeschränkung
  - Bucket Algorithmus
  - Komplexität und Vollständigkeit
- Zwei weitere Algorithmen
- Logische Optimierung basierend auf Query Containment

Einleitung

- Query Containment haben wir bisher nur für einzelne View / Query Paare betrachtet
- Informationsintegration ist gerade die Verknüpfung von Daten bzw. Views (Quellen)
- „Answering Queries Using Views“ (AQUV)
  - Welche Views wollen wir miteinander verknüpfen?
  - Wie sollen Views miteinander verknüpft werden?
  - Wie berechnen wir alle korrekten Verknüpfungen von Views?

Beispiel

- Anfrage: **`q(S,K,N) :- spielt(_,S,_,K),schauspieler(S,N)`**
- Keine Quelle kann diese Anfrage alleine beantworten
  - **`spielfilme`**, **`kurzfilme`**, **`filmkritiken`**, **`spielfilm_kritiken`** fehlt die Relation **`schauspieler`**
  - **`us_spielfilme`** liefert keine **`kritik`** (aber **`nationalitaet='US'`**)
  - **`kurzfilm_rollen`** liefert keine **`kritik`** (aber **`nationalitaet`**)
- Wir müssen also Quellen joinen
- Aber welche – und wie viele?

```
                film(T,Y,R,L),L>79,Y='Spielfilm'   ⊇   spielfilme(T,R,L)
                 film(T,Y,R,L),L<11,Y='Kurzfilm'   ⊇   kurzfilme(T,R)
film(T,_,R,_),spielt(T,S,O,K),O='Hauprolle'        ⊇   filmkritiken(T,R,S,K)
                 film(T,Y,_,L),spielt(T,S,_,_),
    schauspieler(S,N), N='US',Y='Spielfilm'        ⊇   us_spielfilm(T,L,S)
film(T,Y,_,_),spielt(T,_,O,K),Y='Spielfilm'        ⊇   spielfilm_kritiken(T,O,K)
                film(T,Y,_,_),spielt(T,S,O,_),
                schauspieler(S,N), Y='Kurzfilm'    ⊇   kurzfilm_rollen(T,O,S,N)
```

Minimale Pläne

- Prinzipiell kann man unendliche viele Pläne erzeugen
- Definition
  Ein Anfrageplan p für q aus Views $V=\{v_1,...,v_n\}$ heißt minimal, wenn es keinen Plan p‘ für q gibt mit Views V‘ für den gilt:
  1. $V' \subset V$
  2. $result(p) \subseteq result(p')$
- Bemerkung
  - Offensichtlich sind wir nur an minimalen Anfrageplänen interessiert
  - Wir zeigen, dass die eine maximale Länge haben

Längenbeschränkung 1

- Theorem
  Sei p ein Anfrageplan für Anfrage q mit |p|>|q| (d.h., p enthält mehr Views als q Literale). Dann ist p nicht minimal.
- Beweisidee
  - Da p ein Plan für q ist, gibt es ein CM h von q nach p
  - Jedes Literal aus q hat in h genau ein Zielliteral. Damit kann es höchstens |q| verschiedene Zielliterale in p geben
  - Diese können in höchstens |q| verschiedenen Views liegen. Diese müssen auch alle verlangten Variablen exportieren
  - Damit enthält p |p|-|q| Views, die kein Zielliteral aus h enthalten und die auch keine verlangte Variable exportieren

Längenbeschränkung 2

- Theorem
  Sei p ein Anfrageplan für Anfrage q mit |p|>|q| (d.h., p enthält mehr Views als q Literale). Dann ist p nicht minimal.
- Beweisidee
  - ...
  - Wenn wir diese aus p entfernen, kann das durch das verkleinerte p berechnete Ergebnis nur größer werden (mehr Tupel)
    - Die |p|-|q| Views können bestenfalls (durch Joins) Tupel wegfiltern
    - Nicht vergessen: Wir sind in SET Semantik
  - Diese Views sind überflüssig, p ist nicht minimal

Längenbeschränkung

- Lemma
  Sei p ein Anfrageplan für Benutzeranfrage q mit |p|>|q|.
  Dann gibt es einen Plan p‘ mit
    1. *|p‘| < |q|*
    2. *result(p)⊆result(p‘)*
- Bemerkung
  - Alle Pläne länger als |q| berechnen also nur Tupel, die auch von mindestens einem Plan berechnet werden, der kürzer als q ist
  - Also müssen wir nur Pläne aus höchstens |q| Views betrachten

Generate-and-Test Algorithmus

- Wir können nun einen ersten Algorithmus zur Anfrageplanung formulieren
  - Wir ignorieren bis auf weiteres Bedingungen der Art „<“ und „>“
- Generate-and-Test Algorithmus
  - Gegeben eine globale Anfrage q und eine Menge V von Views
  - Gesucht: Alle semantisch korrekten Anfragepläne für q
    - Bilde alle Kombinationen $p=\{v_1,...\}$, $v_i \in V$, mit $|p| \leq |q|$
      - Wir müssen auch Pläne mit $|p|<|q|$ betrachten – Beispiel später
    - Zu jedem p füge die Joins aus q hinzu, wenn nicht schon vorhanden (ergibt p‘)
    - Prüfe, ob die neuen Joins ausführbar sind
    - Teste Containment von p‘ in q

Komplexität

- Wir haben $>|V|^{|q|}$ mögliche Kombinationen
- Für jede Kombination p müssen wir in $O(|p|^{|q|})$ Containment testen
- Teuer

Suchraum verkleinern

- Beobachtung
  - Wir müssen nur Kombinationen testen, die jede Relation aus q mindestens einmal erhalten
    - Mehrere Literale aus q können in ein Literal eines Anfrageplans gemapped werden
  - Für jedes potentielle Zielliteral muss es ein partielles Containment Mapping geben
  - Wir können Views nur zu einem Plan verknüpfen, wenn die partiellen CM kompatibel sind
- Diese Forderungen können wir benutzen, um weniger Plankandidaten aufzählen zu müssen

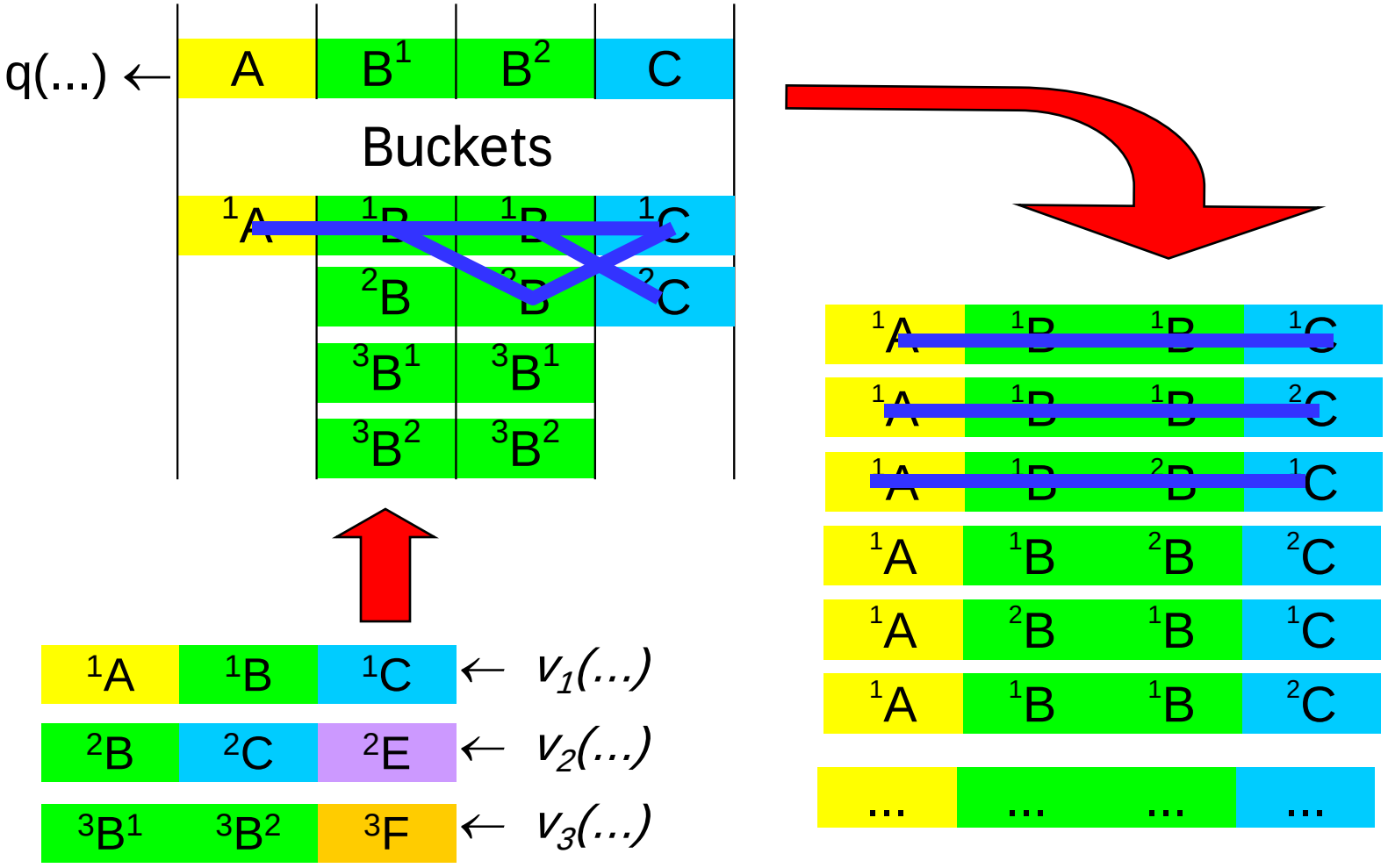
Inhalt dieser Vorlesung

- Answering Queries using Views
  - Längenbeschränkung
  - Bucket Algorithmus
  - Komplexität und Vollständigkeit
- Zwei weitere Algorithmen
- Logische Optimierung basierend auf Query Containment

Bucket – Algorithmus (BA)

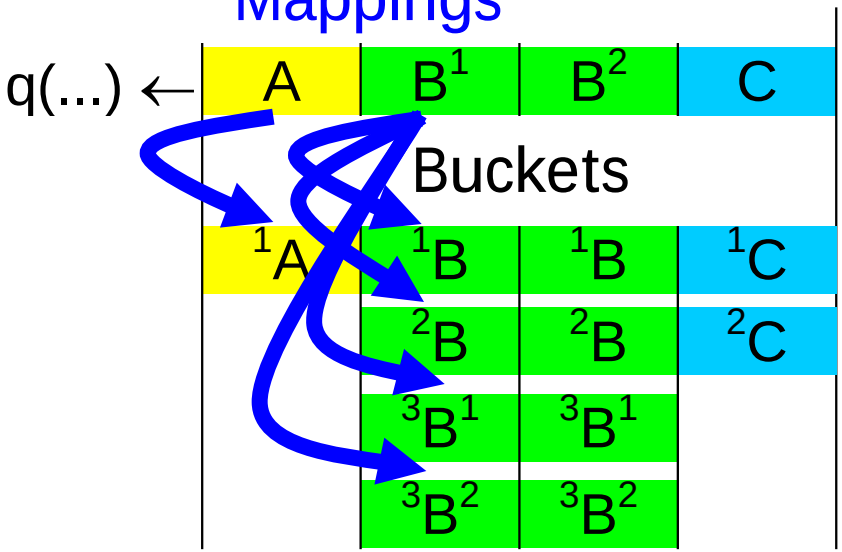
- Views in Buckets für Literale von q sortieren
- Elemente verschiedener Buckets geschickt kombinieren
- Währenddessen wachsende partielle CM berechnen
- Bei Inkompatibilitäten prüfen, ob der Plan durch neue Joins gerettet werden kann
  - Das ist anders als beim DF-Algorithmus für Query Containment
- Wenn ein partielles CM alle Literale von q abdeckt, sind wir fertig – Plan und CM ist gefunden

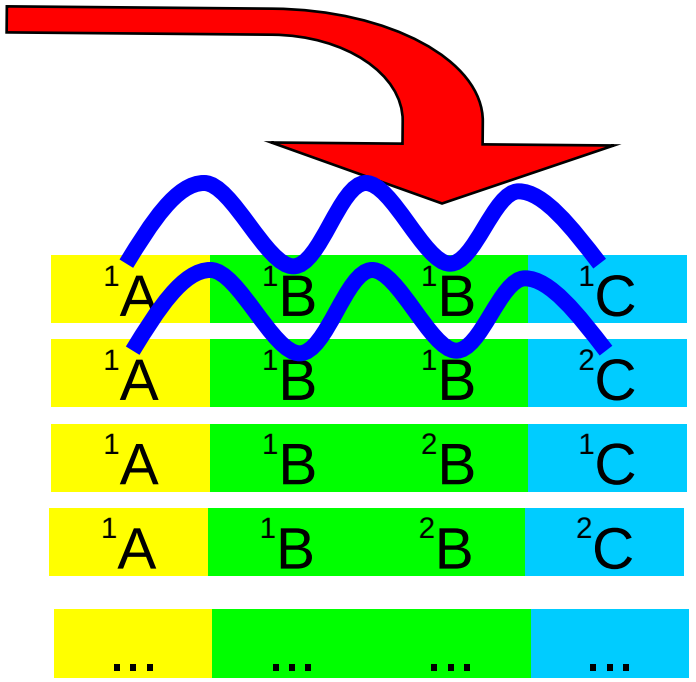
Buckets


q(...) ←
A
B1
B2
C
Buckets
1A
1B
1B
1C
2B
2B
2C
3B1
3B1
3B2
3B2
1A 1B 1C ← v1(...)
2B 2C 2E ← v2(...)
3B1 3B2 3F ← v3(...)
1A 1B 1B 1C
1A 1B 1B 2C
1A 1B 2B 1C
1A 1B 2B 2C
1A 2B 1B 1C
1A 1B 1B 2C
... ...

Buckets

Partielle Containment Mappings





Kompatibilität partieller Containment Mappings

Probleme reparieren

- Wann sind CM inkompatibel?
  - Nicht rettbar
    - Eine Konstante wird auf eine andere Konstante gemapped
    - Eine Variable wird auf unterschiedliche Konstanten gemapped
  - Vielleicht rettbar
    - Eine Variable wird auf zwei unterschiedliche Symbole gemapped, von denen nicht beide eine Konstante sind
    - Eine Konstante wird auf eine Variable gemapped
- In rettbaren Fällen müssen wir testen, ob man den Plan so modifizieren kann, dass er nur noch korrekte Tupel erzeugt
  - Hinzufügen von Selektionen und Joins aus Query
  - Geht nur, wenn wir die zusätz. Bedingungen auswerten können
  - Diese müssen also auf exportierten Variablen liegen

Planning != Containment

- Beispiel
  - **`v(A,B) :- cites(A,B)`**
  - **`q(X) :- cites(X, ‘Felix Naumann‘)`**
- Planung : **`‘Felix Naumann‘`**$\rightarrow$**`B`** ist nicht erlaubt
  - **`v`** ist nicht in **`q`** enthalten
- Also gibt es primär keinen korrekten Anfrageplan
- Wir müssen Pläne erweitern dürfen
  - Erzeuge **`plan(X,Y) :- v(X,Y),Y=‘Felix Naumann‘`**
  - Die Bedingung **`Y=‘Felix Naumann‘`** entnehmen wir der Query
  - Kann man das schon bei der Berechnung partieller CM prüfen
  - Kompatibilität muss auch Erfüllbarkeit aller Bedingungen prüfen

Bucket Algorithmus: Beispiel

- Phase 1: Buckets bauen
  - Für alle Literale l der Query q
  - Finde alle Views, die ein Literal l‘ enthalten, für das es ein partielles CM von l nach l‘ gibt
    - Benennen dabei alle Variablen in l‘ um mit „frischen“ Variablen
  - Packe alle diese Views mit allen partiellen CM in einen Buckets für l
  - Wenn eine Sicht v mehrere passende Literale für l enthält, muss v auch mehrmals (mit unterschiedlichen CM) in den Bucket

Views und Query

```
lehrt( prof, kurs_id, sem)
eingeschrieben( stud, kurs_id, sem)
kurs( kurs_id, titel)
```

```
V1(stud, titel, sem, kurs_id) :-
    E(stud,kurs_id,sem),
    K(kurs_id,titel),
    kurs_id≥500, sem≥WS98;
```

```
V2(stud, prof, sem, kurs_id) :-
    E(stud, kurs_id, sem),
    L(prof, kurs_id, sem)
    kurs_id<400;
```

```
V3(stud, kurs_id) :-
    E(stud, kurs_id, sem),
    sem≤WS94;
```

```
V4(prof, kurs_id, titel, sem) :-
    L(prof, kurs_id, sem),
    K(kurs_id, titel),
    E(stud, kurs_id, sem),
    sem≤WS97;
```

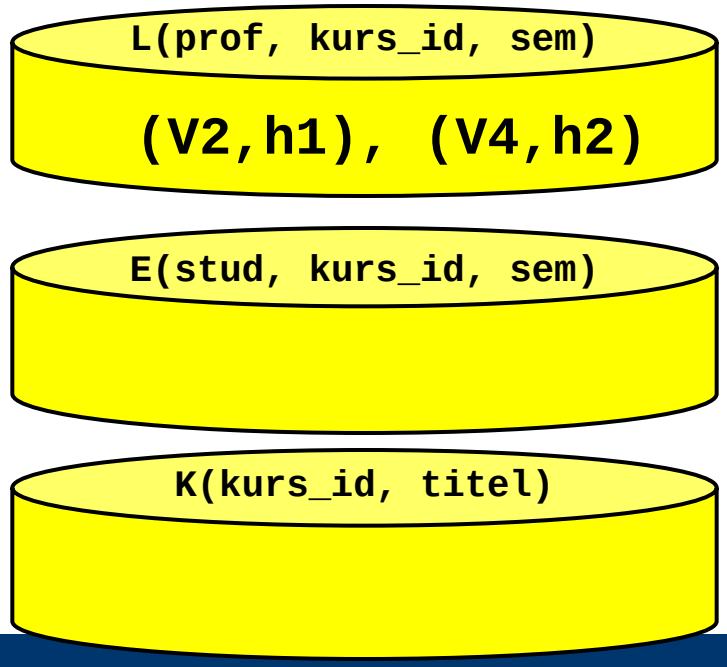
```
q(stud, kurs_id, prof) :-
   L(prof, kurs_id, sem),
   E(stud, kurs_id, sem),
   K(kurs_id, titel),
   sem≥WS95, kurs_id≥300;
```

Bucket **lehrt**

```
V1(stud, titel, sem, kurs_id) :-
     E(stud,kurs_id,sem),
     K(kurs_id,titel),
     kurs_id≥500, sem≥WS98;

V2(stud, prof, sem, kurs_id) :-
     E(stud, kurs_id, sem),
     L(prof, kurs_id, sem),
     kurs_id<400;

V3(stud, kurs_id) :-
     E(stud, kurs_id, sem),
     sem≤WS94;

V4(prof, kurs_id, titel, sem) :-
     L(prof, kurs_id, sem),
     K(kurs_id, titel),
     E(stud, kurs_id, sem),
     sem≤WS97;
```

```
q(stud, kurs_id, prof) :-
   L(prof, kurs_id, sem),
   E(stud, kurs_id, sem),
   K(kurs_id, titel),
   sem≥WS95, kurs_id≥300;
```

- V1, V3: Enthalten kein Literal für „lehrt“
- V2: Kann bzgl. **kurs_id** eingeschränkt werden
- V4: Kann bzgl. **sem** eingeschränkt werden



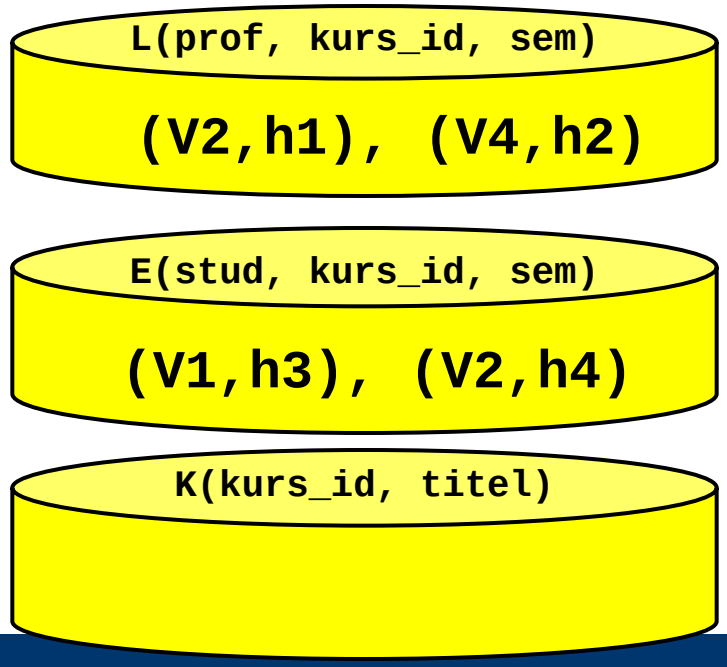
Bucket **eingeschrieben**

```
V1(stud, titel, sem, kurs_id) :-
     E(stud,kurs_id,sem),
     K(kurs_id,titel),
     kurs_id≥500, sem≥WS98;

V2(stud, prof, sem, kurs_id) :-
     E(stud, kurs_id, sem),
     L(prof, kurs_id, sem),
     kurs_id<400;
```

```
V3(stud, kurs_id) :-
     E(stud, kurs_id, sem),
     sem≤WS94;

V4(prof, kurs_id, titel, sem) :-
     L(prof, kurs_id, sem),
     K(kurs_id, titel),
     E(stud, kurs_id, sem),
     sem≤WS97;
```

```
q(stud, kurs_id, prof) :-
   L(prof, kurs_id, sem),
   E(stud, kurs_id, sem),
   K(kurs_id, titel),
   sem≥WS95, kurs_id≥300;
```

- V1: OK (alles exportiert, Bedingungen erfüllt)
- V2: OK (alles exportiert, Bedingungen erfüllbar)
- V3: Konflikt in Bedingungen für **sem**
- V4: **stud** wird nicht exportiert



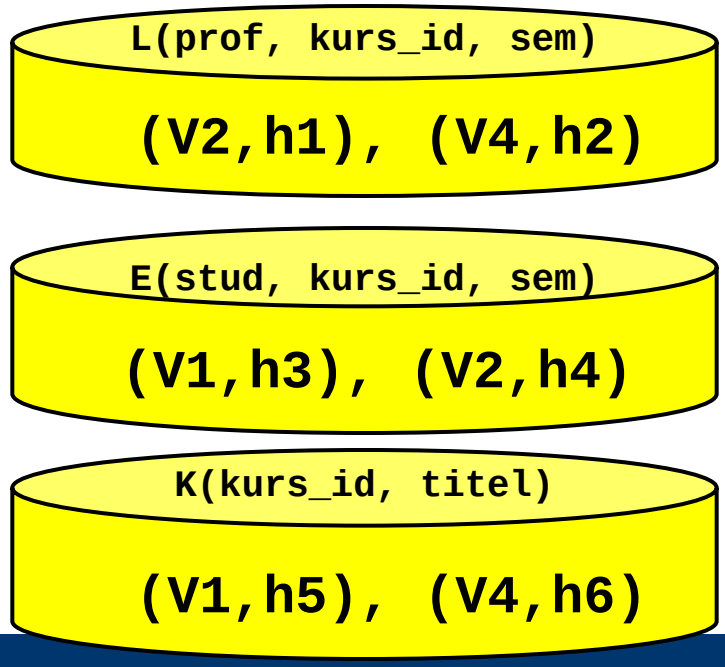
Bucket **kurs**

```
V1(stud, titel, sem, kurs_id) :-
     E(stud,kurs_id,sem),
     K(kurs_id,titel),
     kurs_id≥500, sem≥WS98;

V2(stud, prof, sem, kurs_id) :-
     E(stud, kurs_id, sem),
     L(prof, kurs_id, sem),
     kurs_id<400;
```

```
V3(stud, kurs_id) :-
     E(stud, kurs_id, sem),
     sem≤WS94;

V4(prof, kurs_id, titel, sem) :-
     L(prof, kurs_id, sem),
     K(kurs_id, titel),
     E(stud, kurs_id, sem),
     sem≤WS97;
```

```
q(stud, kurs_id, prof) :-
   L(prof, kurs_id, sem),
   E(stud, kurs_id, sem),
   K(kurs_id, titel),
   sem≥WS95, kurs_id≥300;
```

- V1: OK
- V2: kein Literal **kurs**
- V3: kein Literal **kurs**
- V4: OK



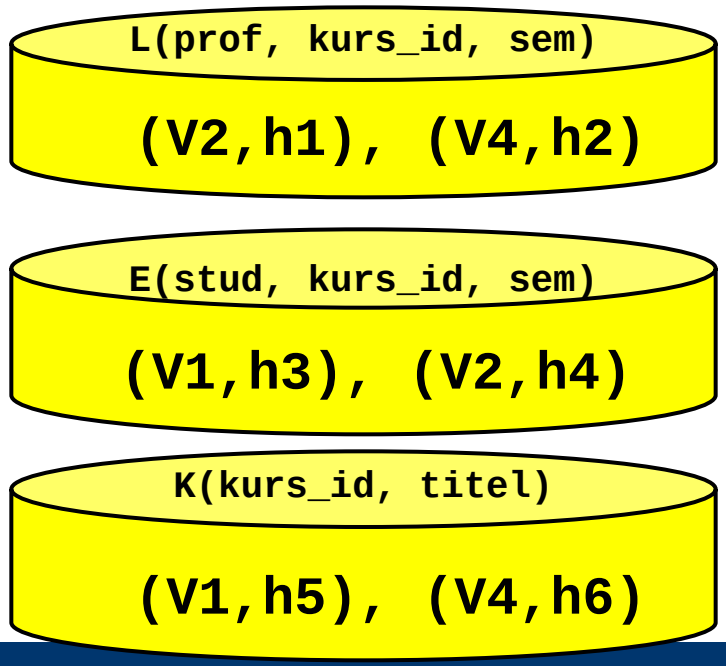
Phase 2

- Phase 2: Teilpläne bauen
  - Bilde inkrementell alle Kombinationen aus Views so, dass aus jedem Bucket genau ein View genommen wird
  - Immer, wenn ein wachsender Teilplan (p, h) um ein Literal (k, h‘) aus v erweitert werden soll,
    - Teste, ob h und h‘ kompatibel sind
    - Wenn ja, bilde p:=p∪v und h=h°h’ und gehe zum nächsten Bucket
    - Wenn nicht, und das Problem ist eine Variable aus q, die auf Variable X in p und auf Variable Y in k gemapped wird
      - Teste, ob X in p und Y in k exportiert wird
      - Wenn ja, p:=p∪v ∪(X=Y) und h=h°h’
    - Teste nächste Kombination (k, h‘) aus dem aktuellen Bucket
- Aber: Pläne sind nun immer genauso lang wie die Query
  - Das wird uns noch Kummer machen

Phase 2: Teilpläne bilden

```
V1(stud, titel, sem, kurs_id) :-
     E(stud,kurs_id,sem),
     K(kurs_id,titel),
     kurs_id≥500, sem≥WS98;

V2(stud, prof, sem, kurs_id) :-
     E(stud, kurs_id, sem),
     L(prof, kurs_id, sem),
     kurs_id<400;
```

```
q(stud, kurs_id, prof) :-
   L(prof, kurs_id, sem),
   E(stud, kurs_id, sem),
   K(kurs_id, titel),
   sem≥WS95, kurs_id≥300;
```

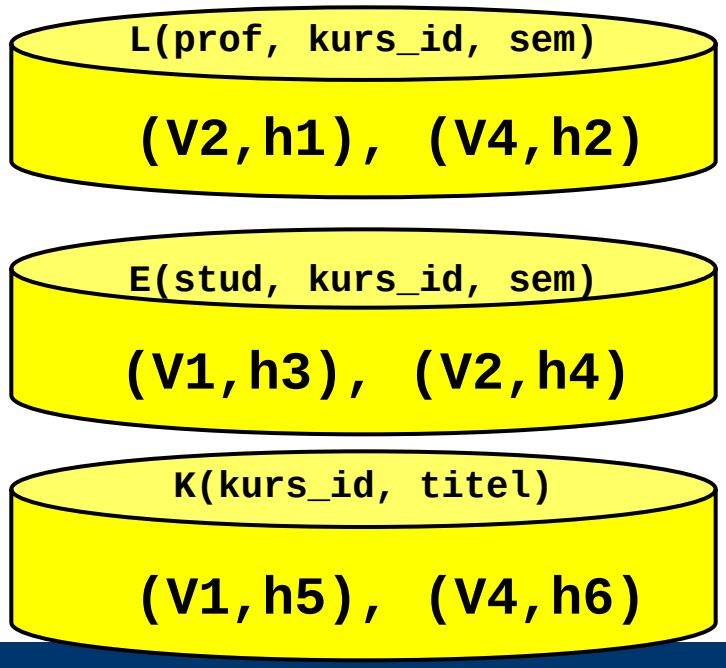


- Kombination V2, V1, V1
  - Teste h_1 und h_3
  - Ist **prof→prof$_1$, kurs_id→kurs_id$_1$, sem→sem$_1$**
  - Kompatibel zu **stud→stud$_2$, kurs_id→ kurs_id$_2$, sem→sem$_2$**
  - Nein, aber **kurs_id$_1$, kurs_id$_2$, sem$_1$** und **sem$_2$** werden exportiert
  - Aber: **kurs_id$_1$≥500∧kurs_id$_2$<400 ∧ kurs_id$_1$=kurs_id$_2$** ist inkompatibel
  - Abbruch, nächste Kombination

Phase 2: Teilpläne bilden

```
V2(stud, prof, sem, kurs_id) :-
     E(stud, kurs_id, sem),
     L(prof, kurs_id, sem),
     kurs_id<400;

V4(prof, kurs_id, titel, sem) :-
     L(prof, kurs_id, sem),
     K(kurs_id, titel),
     E(stud, kurs_id, sem),
     sem≤WS97;
```

```
q(stud, kurs_id, prof) :-
   L(prof, kurs_id, sem),
   E(stud, kurs_id, sem),
   K(kurs_id, titel),
   sem≥WS95, kurs_id≥300;
```

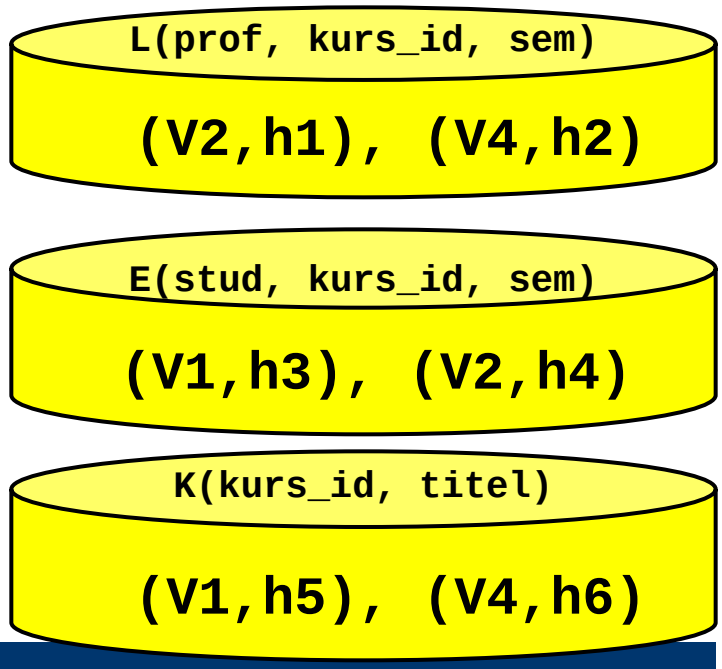


- Kombination V2, V2, V4
  - Teste h_1 und h_4
  - Ist **prof→prof$_1$, kurs_id→kurs_id$_1$, sem→sem$_1$**
  - Kompatibel zu **stud→stud$_2$, kurs_id→kurs_id$_2$, sem→sem$_2$**
  - Nein, aber **kurs_id$_1$, kurs_id$_2$, sem$_1$** und **sem$_2$** werden exportiert
  - Also berechne ($h_1 \circ h_4$)
    - **prof→prof$_1$, kurs_id→kurs_id$_1$, sem→sem$_1$, stud→stud$_2$, kurs_id$_2$=kurs_id$_1$, sem$_2$=sem$_1$**

Phase 2: Teilpläne bilden

```
V2(stud, prof, sem, kurs_id) :-
     E(stud, kurs_id, sem),
     L(prof, kurs_id, sem),
     kurs_id<400;

V4(prof, kurs_id, titel, sem) :-
     L(prof, kurs_id, sem),
     K(kurs_id, titel),
     E(stud, kurs_id, sem),
     sem≤WS97;
```

```
q(stud, kurs_id, prof) :-
   L(prof, kurs_id, sem),
   E(stud, kurs_id, sem),
   K(kurs_id, titel),
   sem≥WS95, kurs_id≥300;
```



- Kombination V2, V2, V4
  - Teste ($h_1 \circ h_4$) und h_6
  - Ist **prof→prof$_1$, kurs_id→kurs_id$_1$, sem→sem$_1$, stud→stud$_2$, kurs_id$_2$=kurs_id$_1$, sem$_2$=sem$_1$**
  - Kompatibel zu **kurs_id→kurs_id$_3$, titel→titel$_3$**
  - Nein, aber alles wichtige wird exportiert
  - Also berechne (($h_1 \circ h_4$) $\circ$ h_6)
    - **prof→prof$_1$, kurs_id→kurs_id$_1$, sem→sem$_1$, stud→stud$_2$, titel→titel$_3$ kurs_id$_2$=kurs_id$_1$=kurs_id$_3$, sem$_2$=sem$_1$**

Inhalt dieser Vorlesung

- Answering Queries using Views
  - Längenbeschränkung
  - Bucket Algorithmus
  - Komplexität und Vollständigkeit
- Zwei weitere Algorithmen
- Logische Optimierung basierend auf Query Containment

Komplexität

- Containment wird im BA nie getestet
- Wo ist dann die exponentielle Komplexität?
  - In der exponentiellen Anzahl von Schritten zur Teilplanerweiterung
  - In jedem Bucket sind maximal |V|*n viele Sichten
    - Mit max(|v|)=n
  - Also gibt es maximal $(|V|*n)^{|q|}$ viele Kandidaten
- Wie teuer ist eine Teilplanerweiterung?
  - Ein partielles CM kann maximal n=|sym(q)| Einzelmappings haben
  - Die kann man sortiert halten
  - Dann ist der Test auf Kompatibilität bzgl. Variablenziele linear in n
  - Auch Test auf Kompatibilität bzgl. Bedingungen ist linear
    - In der Zahl von Bedingungen

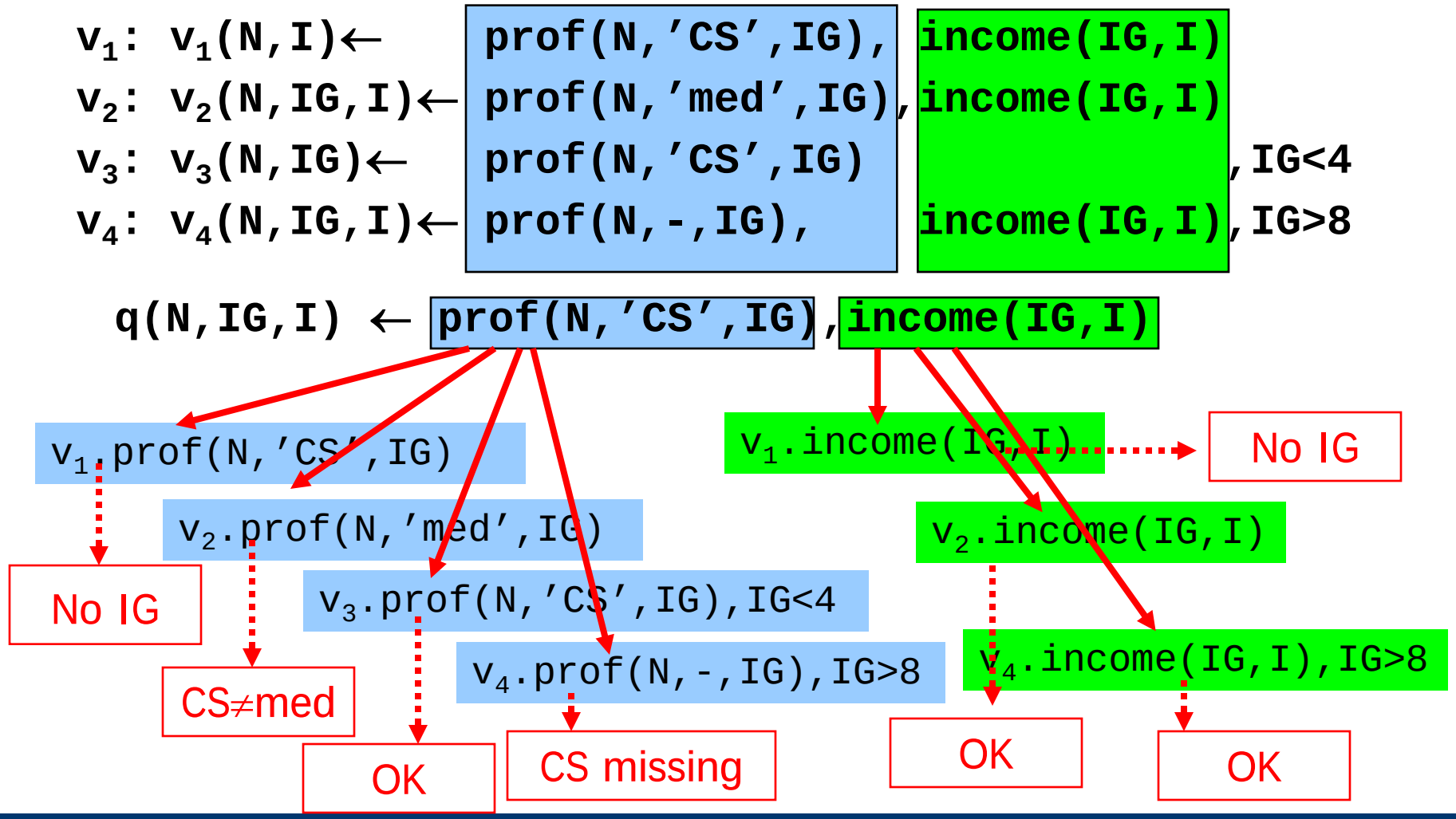
AQUV in Real Life

- Oftmals sind die meisten Buckets eher leer
  - Für viele Relationen des globalen Schemas gibt man nur eine oder wenige Quellen
  - Wenn jede Relation nur einmal in allen Views vorkommt, ist Anfrageplanung linear
- Die meisten Teilplanerweiterungen scheitern; dann muss man den Suchraum nicht komplett ablaufen
- AQUV ist harmlos in den meisten Anwendungen
  - Ausnahme: Anfragen auf Strukturen (Graphen, Bäume, etc.)

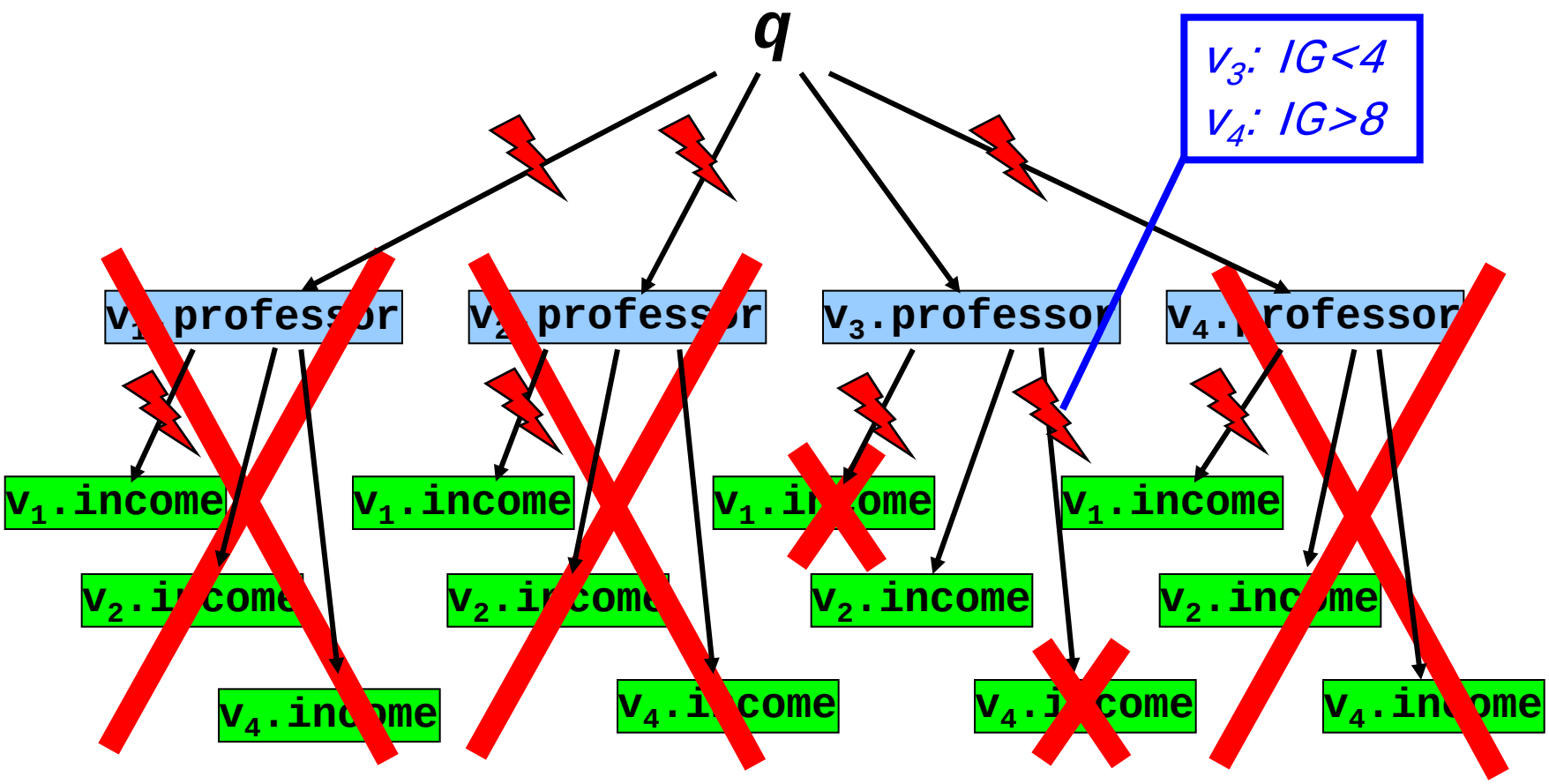
AQUV und Containment

- Bucket-Algorithmus und Depth-First Containment Test sind sich ähnlich
  - Der Bucket Algorithmus pickt sich potentielle Zielliterale aus den Views
  - Beim Containment findet man die in der zweiten Query
  - Kompatibilität und UNION von partiellen CM
    - BA darf Kompatibilität erzwingen, Containment nicht
    - Zur Anfrageplanung (später) wollen wir auch fast-passende materialisierte Views zurechtbiegen
  - Der Suchraum ist ähnlich strukturiert
  - Die Komplexität ist vergleichbar: Zahl notwendiger Zielliterale hängt von der Query ab, nicht von der Länge der Views

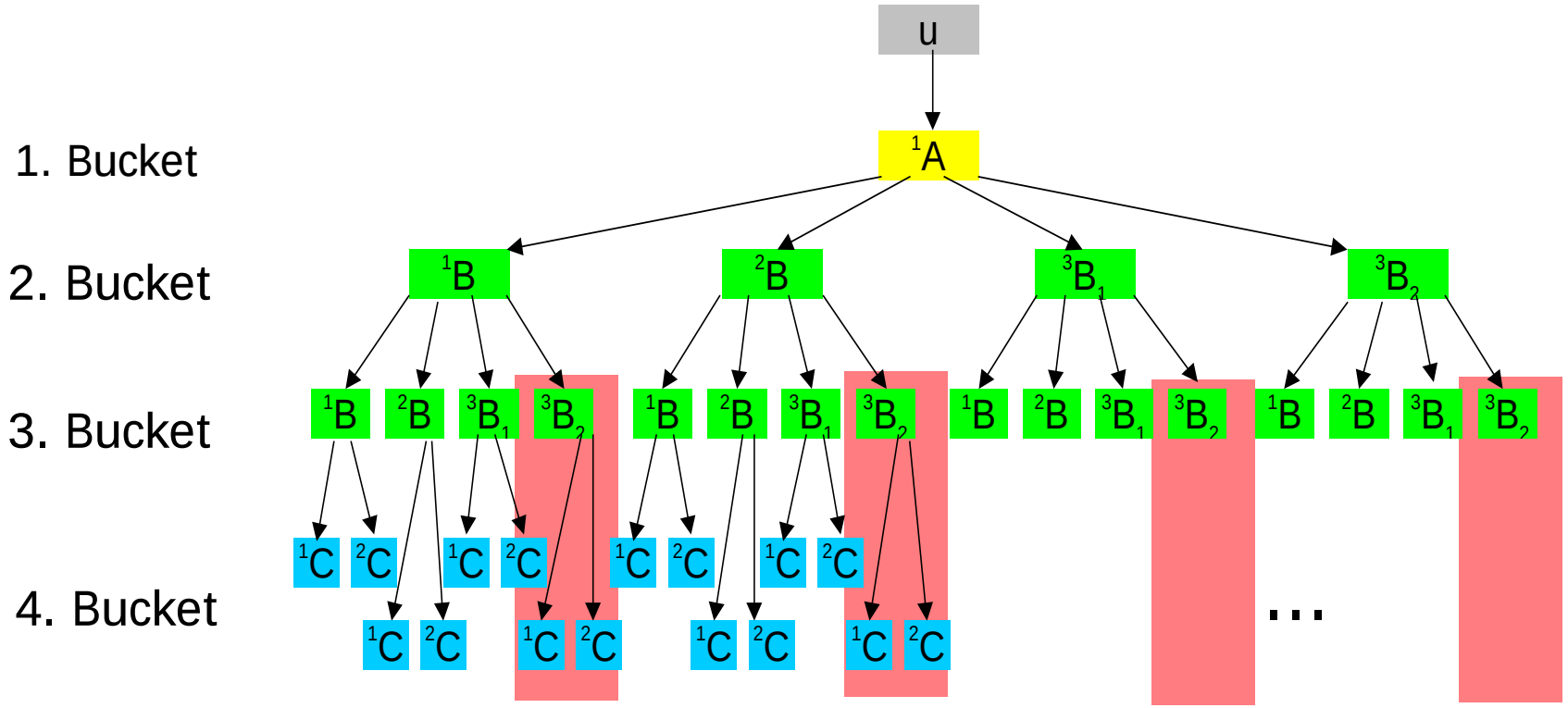
Noch ein Beispiel



Suchraum


q
v3: IG<4
v4: IG>8
v1.professor
v2.professor
v3.professor
v4.professor
v1.income
v1.income
v1.income
v1.income
v2.income
v2.income
v2.income
v2.income
v4.income
v4.income
v4.income
v4.income

Doppelte Arbeit



Beobachtung

- Wann sind partielle CM inkompatibel?
  - Hängt von der Anfrage ab (Exportiert, Bedingungen, Joins)
  - Hängt von den Views ab
- Literale verschiedener Views können auch unabhängig von einer Anfrage inkompatibel sein
- Das wird im BA u.U. immer und immer wieder getestet
- Die Zahl tatsächlich durchgeführter Tests hängt von der Reihenfolge der Buckets und der Literale in den Buckets ab
- Optimierungspotential (Dynamic Programming)

Unvollständigkeit

- View
  - **`v1(A)     :- cites(A,B), cites(B,A)`**
  - **`v2(F,G,H) :- cites(F,G), cites(G,H)`**
- Query **`q(X)  :- cites(X,Y), cites(Y,X)`**
- Buckets
  - **$cites^1$: { (v11: X→A1, Y→B1), (v21: X→F1, Y→G1), (v22: X→G1, Y→H1)}**
  - **$cites^2$: { (v12: Y→B2, X→A2), (v21: Y→F2, X→G2), (v22: Y→G2, X→H2)}**
- Einige Pläne
  - v11⋈v12: Inkompatibel, da B1=B2 nicht sichergestellt werden kann
  - v11⋈v21: Inkompatibel, da B1=F2 nicht sichergestellt werden kann
  - v21⋈v21: OK, da G1=F2 sichergestellt werden kann
  - ...
- Was ist komisch?

Unvollständigkeit

- View
  - `v1(A)     :- cites(A,B), cites(B,A)`
  - `v2(F,G,H) :- cites(F,G), cites(G,H)`
- Query `q(X)  :- cites(X,Y), cites(Y,X)`

- Einige Pläne
  - v11⋈v12: Inkompatibel, da B1=B2 nicht sichergestellt werden kann
- Der Join ist doch schon in dem View
- BA verliert den Join durch das Zerbrechen in Literale
- Nur der kürzere Plan <v1> ist korrekt (und ausführbar)

BA ist nicht vollständig

- BA ist korrekt, aber nicht vollständig
  - Der BA zählt nur Pläne der Länge |q| auf
  - Es kann aber kürzere Pläne geben, die korrekt und nicht in einem ausführbaren längeren Plan enthalten sind
    - Joins über nicht-exportierte Variable
    - Im kurzen Plan können die durch den View selber sichergestellt sein
    - Im langen Plan bräuchte man einen zusätzlichen Join – geht nicht

Improved Bucket Algorithmus

- Teste beim Hinzufügen eines Views aus einem Bucket, ob man tatsächlich einen neuen View hinzufügen muss
- Versuche, mehrere Literale mit einem View abzudecken
- Liefert einen vollständiger und korrekter Algorithmus
- Bessere Komplexität als vollständige BA Variante
  - Die muss auch alle kürzeren Pläne aufzählen

Inhalt dieser Vorlesung

- Answering Queries using Views
- Zwei weitere Algorithmen
- Logische Optimierung basierend auf Query Containment

Algorithmus 1: Minicon [PL01]

- Probleme machen vor allem Joins in der Benutzeranfrage
  - Sind die in einem Plan?
  - Kann man sie in einem Plan erzwingen?
  - Werden die notwendigen Variablen exportiert?
- Diese Beobachtung nutzt MiniCon, um
  - Kleinere Buckets zu bauen – manche Views kann man früh ausschließen
  - Weniger Pläne aufzuzählen – die „Building Blocks“ werden größer
  - Keine bessere WC Komplexität, aber besser im AC
- Grundaufbau wie BA
  - Buckets pro Literal der Query, partielle CM, UNION, ...
  - Aufzählen der potentiell problematischen Joins statt der Literale

Kleinere Buckets

- Sei v ein View für Literal l mit partiellem CM h: l→k
- Wenn l in q einen Join über Variable X mit Literal l‘ macht, und X in k nicht exportiert wird (also auch nicht in l)
  - Prüfe, ob es ein Literal k‘ in v gibt, das ein potentielles Ziel für l‘ ist
  - Wenn !∃k‘∈v: Füge v nicht in B_l ein („Vorausschau“)
    - Der BA würde den einfügen
    - Aber wir können den Join über X niemals herstellen
  - Wenn ∃k‘∈v und der Join über X ist nicht in v: Füge v nicht in B_l ein
    - Der BA würde den einfügen
    - Aber wieder – der Join kann in keinem Plan mit v hergestellt werden
  - Sonst: Füge v in Bucket B_l ein
    - Merke außerdem das Literal k‘, das mit k über X gejoined wird
    - Wenn v X nicht exportiert, müssen k und k‘ in jedem Plan mit v von v abgedeckt werden
      - Das muss beim Aufzählen berücksichtigt werden

Algorithmus 2: Inverse Rules [GKD97]

- Beobachtung: Ein View liefert Tupel für jedes seiner Literal
- IR drückt dies in einer Regel pro Literal jedes Views aus
  - Kopf der Regel ist das Literal
  - Rumpf der Regel ist der Kopf des Views
  - Inverse Rules – Views werden zu Regeln umgedreht
- Beispiel: `V3(F,H) :- cites(F,G),cites(G,H),sametopic(F,G)`
  - `cites(F,?) :- V3(F,H)`
  - `cites(?,H) :- V3(F,H)`
  - `sametopic(F,?) :- V3(F,H)`
- Problem: Nicht-exportierte Variable und Joins

Skolemization

- Nicht-exportierten Variable
  - Wenn nicht in Joins: Unerheblich, ersetzen durch „_“
  - Wenn in Joins: Information muss erhalten bleiben
    - Existenz eines Wertes, ohne ihn zu kennen
    - Eine Skolemfunktion liefert konzeptionell pro Kombination von Werten der exportierten Variablen einen eindeutigen Wert
    - Trick: Einfach den Aufruf (als String) verwenden
- Beispiel: `v3(F,H) :- cites(F,G),cites(G,H),sametopic(F,G)`
  - `cites(F,f(F,H)) :- v3(F,H)`
  - `cites(f(F,H),H) :- v3(F,H)`
  - `sametopic(F,f(F,H)) :- v3(F,H)`
- Gleiche Skolemausdrücke dürfen gejoined werden
- Skolemausdrücke dürfen nicht im Ergebnis erscheinen

Planung mit Inverse Rules

- Wir interpretieren die Regeln als „externe“ Datalog-Regeln
  - Die Instantiierungen (Ground-Fakten) liegen in den Quellen
  - Durch Ausführen einer Regel können wir sie berechnen
- Die Anfrage q können wir einfach als Programm auf diesen Regeln laufen lassen
- Der Rest ist PROLOG / DATALOG
  - Unifikation der Variablen und Konstanten
  - Gleiche Skolemterme dürfen unifiziert werden
  - Skolemterme dürfen nicht als Ergebnis zurückgegeben werden

Inhalt dieser Vorlesung

- Answering Queries using Views
- Zwei weitere Algorithmen
- Logische Optimierung basierend auf Query Containment
  - Redundante Pläne
  - Redundante Teilpläne

Globale Anfrageoptimierung

- Zur Erinnerung: $\mathbf{result(q) = \cup\ result(p)}$

$$\begin{aligned} &= p_1 \cup p_2 \cup \ldots \cup p_n \\ &= (v_{11} \bowtie v_{12} \bowtie \ldots \bowtie v_{1n}) \cup \\ &\quad (v_{21} \bowtie v_{22} \bowtie \ldots \bowtie v_{2n}) \cup \\ &\quad \ldots \quad \cup \\ &\quad (v_{m1} \bowtie v_{m2} \bowtie \ldots \bowtie v_{mn}) \end{aligned}$$

- Globale Optimierung: Finde die minimale Menge von Ausführungen von v‘s um result(q) zu berechnen
  - Die v_{ij} sind idR nicht alle unterschiedlich
  - Daher kann man sich uU Ausführungen sparen
  - Globale Optimierung betrachtet alle Anfragepläne auf einmal

Möglichkeiten

- Redundante Pläne entfernen
  - Pläne, die nichts neues zum Ergebnis beitragen können
- Redundante Views innerhalb eines Anfrageplans entfernen
  - Views aus dem Plan entfernen, die das Ergebnis nicht beeinflussen
- Redundante Teilpläne zwischen Anfrageplänen entfernen
  - Gesamtzahl von Viewausführungen minimieren
  - Multiple Query Optimization
- Redundanz kann und muss auf Basis der Korrespondenzen bewiesen werden

Redundante Anfragepläne

- Definition
  Sei q eine Benutzeranfrage und P die Menge aller semantisch korrekten Anfragepläne für q. Ein Plan $p \in P$ ist redundant, wenn

$$\bigcup_{k \in P} result(k) = \bigcup_{k \in P \setminus p} result(k)$$

- Redundante Pläne können offensichtlich ignoriert werden

Erste Idee

- p ist redundant, wenn $\exists p' \in P: p \subseteq p'$
  - Aber: Was heißt denn $p \subseteq p'$?
  - Beispiel
    - **v1: film(T,R) :- doku(T,R)**
    - **v2: film(T,R) :- spielfilm(T,R)**
    - **q: film(T,R)**
- Können wir Query Containment auf Planebene benutzen?
  - Zwei Pläne: **p1:-v1 und p2:-v2**
  - Formal gilt (Expansion der globalen Anfragen): **p1≡p2**
  - Trotzdem sollten beide zur Beantwortung von q benutzt werden
  - Denn die beiden **film**-Literale haben unterschiedliche Extension, wenn sie aus unterschiedlichen Quellen kommen

Wrapperanfragen analysieren

- Lemma
  Sei q eine Benutzeranfrage und p_1, p_2 zwei Anfragepläne für q. Sei loc(p_1) bzw. loc(p_2) die Expansion von p_1, p_2 mit den jeweiligen Wrapperqueries. Dann gilt

$$p_1 \subseteq p_2 \quad gdw. \quad loc(p_1) \subseteq loc(p_2)$$

- Bemerkung
  - Relationen in loc() sollten mit der Quelle „gepräfixed“ werden (name space)

Beispiel

- Beispiel
  - **v1: film(T,R) :- q.doku(T,R)**
  - **v2: film(T,R), reg(R,N) :- q.doku(T,R), q.reg(R,N)**
  - **q: film(T,R)**
  - **loc(v1) = <q.doku(T,R)>**
  - **loc(v2) = <q.doku(T,R), q.reg(R,N)>**
  - Damit ist **p2⊆p1, p2** ist redundant
- Ursachen für das Auftreten redundanter Pläne
  - Überlappende, sich enthaltende oder redundante Korrespondenzen
  - Durch Anfrageplanung erzeugt

Inhalt dieser Vorlesung

- Answering Queries using Views
- Zwei weitere Algorithmen
- Logische Optimierung basierend auf Query Containment
  - Redundante Pläne
  - Redundante Teilpläne

Multiple Query Optimization

- Verschiedene Pläne benutzen oft dieselben Views
- Dann kann man sich ev. die doppelte Ausführung sparen
- Spezialfall der Multiple Query Optimization (MQO)
  - Gegeben eine Menge Q von Queries
  - MQO sucht nach common subexpressions (gleichen Teilanfragen)
  - Diese werden nur einmal ausgeführt
  - Queries in Q umschreiben, um Ergebnisse der common subexpressions zu benutzen
- In unserem Fall: „common subexpression“ = gleiche Views

Der einfache Fall

- Wenn wir weder Projektionen noch Selektionen noch Joins pushen wollen, ist das Problem einfach
- Jeder View muss nur einmal ausgeführt werden
- Alle weiteren Ausführungen benutzen dieses Ergebnis

Beispiel

- **v1: $film_1$(T,R) :- doku(T,R)**
- **v2: $film_2$(T,R) :- spielfilm(T,R)**
- **v3: reg(R,N) :- reg(R,N)**
- **q(T,R,N) :- film(T,R), reg(R,N)**
- **p1: v1 ⋈ v3 = $film_1$(T,R),reg(R,N)**
- **p2: v2 ⋈ v3 = $film_2$(T,R),reg(R,N)**

- Kein Plan ist redundant
- v3 ist doppelt und muss nur einmal ausgeführt werden
- Voraussetzung: Keine Selektionen pushen

Kompliziertere Fälle

- **v1: $film_1$(T,R) :- doku(T,R)**
- **v2: $film_2$(T,R) :- spielfilm(T,R)**
- **v3: reg(R,N) :- reg(R,N)**
- **q(T,R,N) :- film(T,R), reg(R,N)**
- **p1: v1 ⋈ v3 = $film_1$(T,R),reg(R,N)**
- **p2: v2 ⋈ v3 = $film_2$(T,R),reg(R,N)**
- Alternative
  - Erst v1 bzw. v2 ausführen und Werte für R an v3 pushen
  - Da v1 und v2 unterschiedliche Extensionen haben, muss v3 mehrmals aufgerufen werden
  - Man kann Aufrufe sparen, aber v3 in keinem Plan eliminieren
- Was ist schneller? Hängt davon ab ...