1 Fehlerbehandlung, Recovery (Blatt 01)

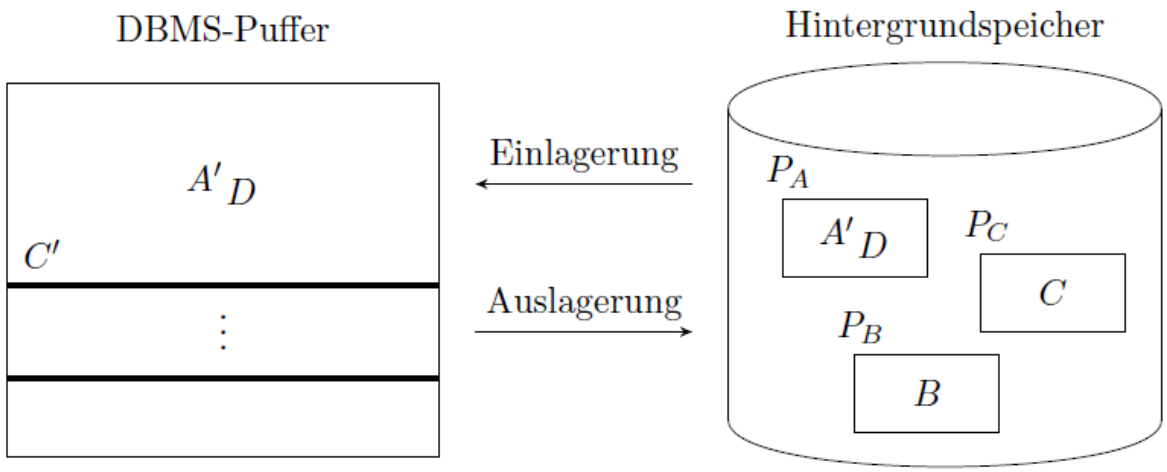
1.1 Phasen des Wiederanlaufs

Situation: System stürzt ab → Hintergrundspeicher in ungewissem Zustand

1. Vollständige Analyse der Log-Datei: Winner- und Loser-Tx identifizieren
2. Redo: **alle** protokollierten Änderungen einbringen
3. Undo **der Loser**: Änderungsoperationen der Loser **in umgekehrter Reihenfolge** rückgängig machen

1.2 Einlagern und Auslagern von Pufferseiten: steal / force

- Damit eine Tx mit Daten arbeiten kann, muss die entsprechende Seite in den Puffer
- Eine Pufferseite, die bearbeitet wurde, ist **dirty**
  - Commit: Änderungen werden festgeschrieben
  - Abort: Änderungen werden verworfen
- Seiten, die gerade bearbeitet werden, sind **fixiert**



1.2.1 Steal

Seiten im Puffer wechseln ständig, wenn Tx kommen und gehen → u.U. reicht der Speicher nicht, um Seiten für eine neue Tx zu laden...

- *steal*-Strategie: jede nicht fixierte Seite kann verdrängt werden → **Undo nötig**
- ¬*steal*-Strategie: keine dirty Seite darf verdrängt werden (d.h. Seiten, die von einer noch laufenden Tx bearbeitet wurden)

1.2.2 Force

Falls eine Tx erfolgreich war (commit), muss die Änderung im HGS festgeschrieben werden.

- $force$ -Strategie: wenn eine Tx committet, müssen ihre Änderungen **umgehend** festgeschrieben werden
- $\neg force$-Strategie: Änderungen werden „irgendwann“ festgeschrieben und gehen bei Systemabsturz ggf. verloren → **Redo nötig**

1.2.3 Kombinierbarkeit von steal und force

„Idealkombination“ $force \wedge \neg steal$ kann nicht umgesetzt werden (Aufgabe 01.1):

- Zwei Tx laufen gleichzeitig und bearbeiten Daten auf der gleichen Seite
- Eine Tx committet…
    - Gemäß $force$ muss jetzt die Seite festgeschrieben werden
    - $\neg steal$: die Seite darf erst permanent werden, nachdem die zweite Tx fertig ist

1.2.4 Logeinträge

Physische Protokollierung: Die **Zustände** vor und nach einer Operation einer Tx werden gespeichert („Before- / After-Image“)

Logische Protokollierung: Die **Transaktionen** zwischen Before- und After-Image werden gespeichert

$$[LSN, TxID, PageID, \boldsymbol{Redo|After}, \boldsymbol{Undo|Before}, PrevLSN]$$

Compensation Log Records (CLR): In CLRs wird das Ausführen der Undo-Operationen der Loser-Tx protokolliert.

- Wenn das System während der Recovery abstürzt, wird durch CLRs sichergestellt, dass Undo-Operationen nicht mehrfach ausgeführt werden
- Wenn das System irgendwann später abstürzt, wird dank CLRs wieder ein konsistenter Zustand hergestellt
- $< LSN, TxID, PageID, \boldsymbol{Redo}, \boldsymbol{PrevLSN}, \boldsymbol{UndoNxtLSN} >$
    - Redo eines CLR = Undo der entspr. Loser-Operation
    - PrevLSN = LSN der letzten Operation **der Loser-Tx** (nicht insgesamt)
    - UndoNxtLSN = LSN der als nächstes rückgängig zu machenden Operation (für bessere Performance)

1.2.5 Idempotenz von Redo- und Undo-Phase

- Idempotenz = Fehlertoleranz

- **Redo:** Seiten speichern immer die LSN der letzten Operation, durch die sie bearbeitet wurden; auch während Redo → so ist sichergestellt, dass **Redo-Operationen nicht mehrfach** ausgeführt werden
- **Undo:** CLRs protokollieren, welche Undo-Operationen bereits durchgeführt wurden → **mehrfaches Ausführen von Undo-Operationen** vermieden

2 EIGENSCHAFTEN VON HISTORIEN (BLATT 02)

Definition „Konfliktoperation“: Operationen, die bei unkontrollierter Nebenläufigkeit zu Inkonsistenz führen können, d.h.:

- Min. zwei Tx greifen auf das gleiche Datum zu
- Min. eine Tx schreibt

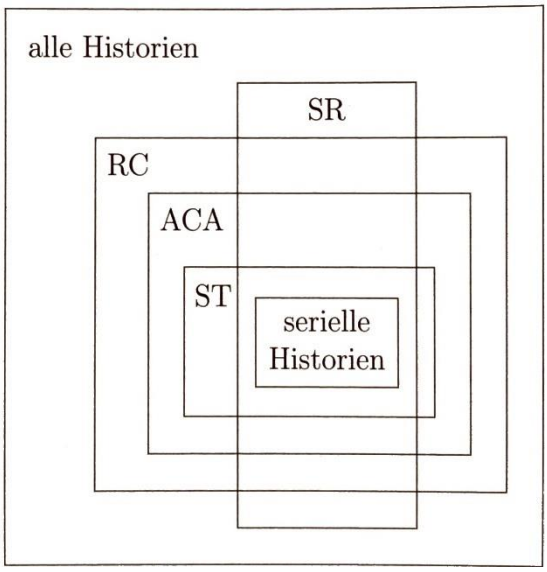
In diesen Fällen ist **die Reihenfolge der Ausführung ausschlaggebend für das Ergebnis.**

2.1 ANALYSE

2.1.1 Voraussetzungen

- Eine Tx **liest** von einer anderen, wenn sie ein Datum liest, das die andere davor geschrieben hat
- **Serialisierbarkeitsgraph:** Gib die Reihenfolge an, in der die Tx einer Historie konfliktfrei ausgeführt werden können (falls zyklisch, gibt es keine konfliktfreie Möglichkeit).
    - Jede Tx ist ein **Knoten**
    - Die **Kanten** werden durch die Reihenfolge von Konfliktoperationen zwischen Tx / Knoten vorgegeben

2.1.2 Eigenschaften



Serialisierbarkeit (SR): Der Serialisierbarkeitsgraph ist azyklisch → es existiert eine konfliktäquivalente serielle Historie.

Striktheit (ST)	$w_j(A) <_H o_i(A) \Rightarrow c_j \| a_j <_H o_i(A)$	**Hierarchie:** eine ST-Historie ist auch ACA, eine ACA-Historie ist auch RC
	Die Tx, die das Konfliktdatum geschrieben hat, muss committen, bevor die andere **irgendetwas** mit dem Datum macht.	
Kaskadierendes Rücksetzen vermeidend (ACA)	$w_j(A) <_H r_i(A) \Rightarrow c_j <_H r_i(A)$	
	Die Tx, die das Konfliktdatum zuerst geschrieben hat, muss committen, bevor die andere **das Datum liest**.	
Rücksetzbar (RC)	$w_j(A) <_H r_i(A) \Rightarrow c_j(A) <_H c_i(A)$	
	Die Tx, die das Konfliktdatum zuerst geschrieben hat, muss committen, bevor die andere **committet**.	

2.2 Two-Phase-Locking (2PL) Protokoll

Um ein Objekt zu benutzen, muss eine Tx eine Sperre darauf erwerben:

- Sperren werden auf jedes Objekt nur einmal angefordert
- Zum Lesen ist ein **S-Lock („shared“)** nötig, zum Schreiben ein **X-Lock („exclusive“)**

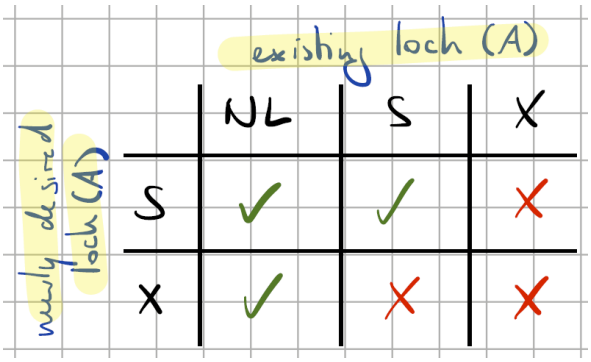


Figure 1. Verträglichkeit von Locks

Jede Tx hat zwei Phasen:

- **Wachstumsphase:** hier dürfen nur neue Sperren hinzukommen und keine freigegeben werden
- **Schrumpfungsphase:** hier dürfen nur Sperren freigegeben werden
- Im **strengen 2PL-Protokoll**, werden alle Sperren auf einmal beim EOT gelöst
    - S-Sperren können schon früher gelöst werden, ohne Striktheit (ST) zu verletzen

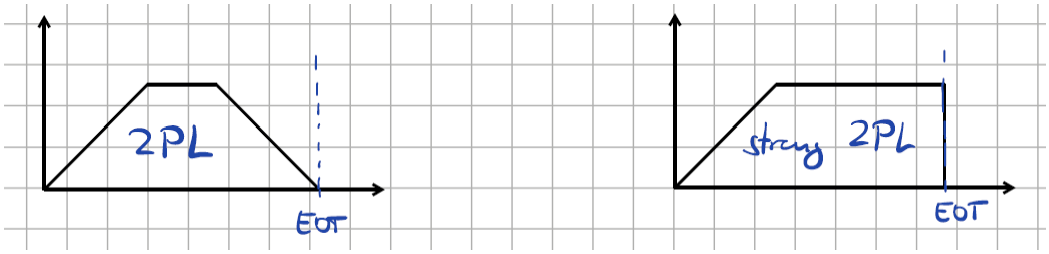
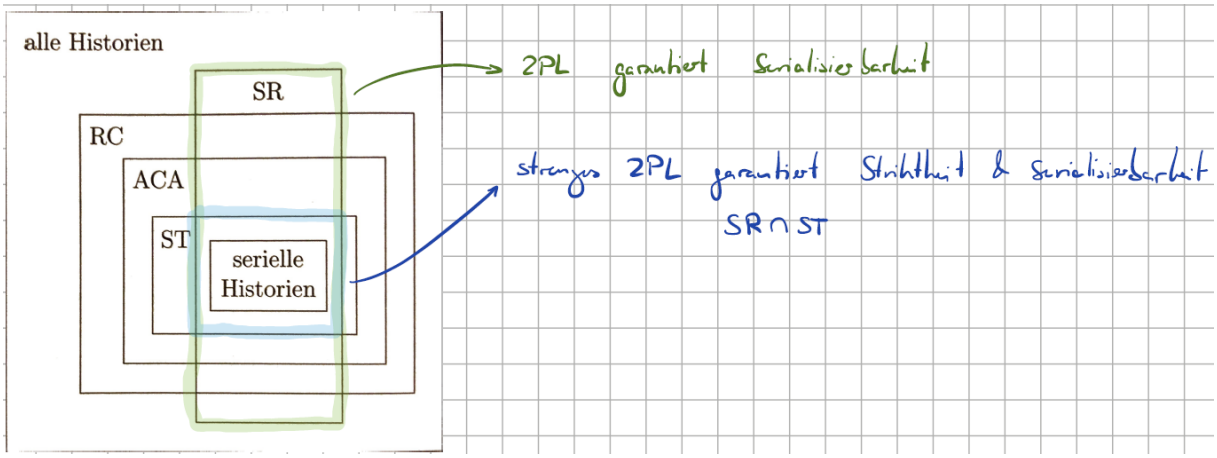


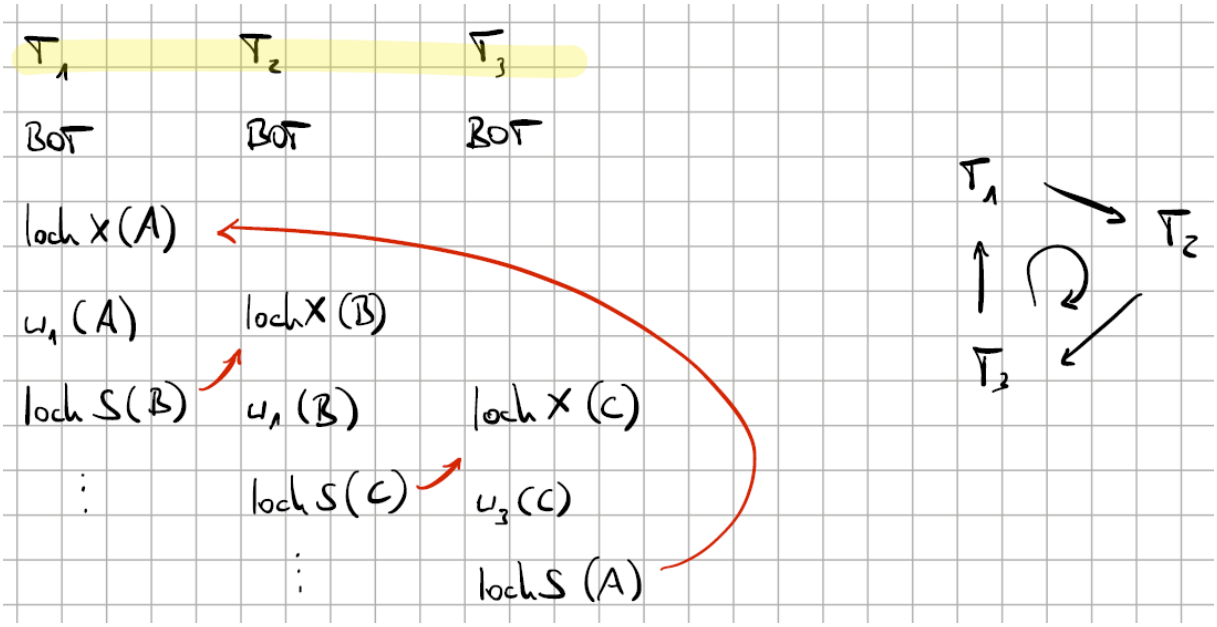
Figure 2. "normales" und strenges 2PL

Der **Vorteil** ist, dass alle nach 2PL erstellten Historien **serialisierbar** sind. Historien nach **strengem 2PL** sind sogar außerdem **strikt:**



2.3 DEADLOCKS

Deadlocks zeigen sich als **Zyklen im Wartegraph** (= SR-Graph mit umgekehrten Kanten). Z.B. ein Deadlock mit Zykluslänge 3:



2.4 TYPISCHE KONSISTENZPROBLEME UND ISOLATION LEVELS IN SQL

Table I. Welche Isolation Levels lösen welche Probleme?

		lost update	**dirty read**	**non-repeatable read**	**phantom problem**
Isolation Level	**read uncommited**	✓			
	read commited	✓	✓		
	repeatable read	✓	✓	✓	
	serializable	✓	✓	✓	✓

Lost Update: zwei Tx wollen parallel ein Datum schreiben → der frühere Wert „geht verloren"

Dirty Read: eine Tx liest ein Datum einer noch laufenden Tx → Risiko: die lesende Tx committet, aber die gelesene abortet

Non-repeatable Read: wiederholtes Lesen liefert unterschiedliche Werte

Phantomproblem: während eine Tx liest, ändert eine andere den zugrundeliegenden Datensatz (z.B.: erst Tx führt SELECT durch, während die zweite das Selektionsattribut eines Tupels ändert)

3 SECURITY (BLATT 03)

3.1 K-ANONYMITÄT

- In Anfragen sind nur Aggregatsfunktionen erlaubt
- Es müssen mindestens k Datensätze aggregiert werden

3.2 PREPARED STATEMENTS UND INPUT SANITIZATION

Input Sanitization: Sonderzeichen werden ersetzt / entfernt, z.B. `&quot;` statt Anführungszeichen. → Viele SQL-Injection Attacken werden damit unmöglich.

Prepared Statements: DB-Anfragen werden als Template vorbereitet, das Inputs bestimmter Art erwartet.

- **Struktur** der Anfrage ist fix, nur die Parameter sind variabel
- Bei richtiger Implementierung kann DB die Anfrage „vorbereiten“, um Latenzen zu verringern

```cpp
#include <pqxx/pqxx>
#include <iostream>
#include <string>
pqxx::result exec_unsafe(pqxx::connection& conn, int matrnr, std::string vorl){
  std::string q = "SELECT * FROM pruefung WHERE matrikelnummer=",
    q2 = "AND vorlesung='", q3 ="'";
  pqxx::work tx{conn, ""}; // Begin of transaction
  pqxx::result r(tx.exec(q + std::to_string(matrnr) + q2 + vorl + q3));
  tx.commit(); // Commit transaction
  return r;
}
int main(int argc, char* argv[]){
  pqxx::connection conn;
  auto r = exec_unsafe(conn, 123, "Grundzuege");
  for(auto row: r)
     std::cout << row["note"] << std::endl;
  return 0;
}
```

simple string concatenation

Figure 3. Ohne Template → Injection möglich

```cpp
#include <pqxx/pqxx>
#include <iostream>
#include <string>
pqxx::result exec_prep(pqxx::connection& conn, int matrnr, std::string vorl){
  pqxx::work tx{conn, ""}; // Begin of transaction
  auto r = tx.prepared("getGrade")(matrnr)(vorl).exec();
  tx.commit(); // Commit transaction
  return r;
}
int main(int argc, char* argv[]){
  pqxx::connection conn;
  conn.prepare("getGrade",
    "SELECT * FROM pruefung WHERE matrikelnummer=$1 AND vorlesung=$2");
  auto r = exec_prep(conn, 123, "Grundzuege");
  for(auto row: r)
     std::cout << row["note"] << std::endl;
  return 0;
}
```

Figure 4. Mit Template → keine Injection

3.3 Passwort-Hashing

- Statt in Klartext wird das PW als Hash gespeichert → korrekte Eingabe wird dann auch anhand der Hashes geprüft
- PROBLEM: Dank online **Rainbow-Tables** sind Hashes nicht unbedingt sicher (wenn das zugrundeliegende PW schlecht / schon mal geknackt ist). Helfen kann...
  - Mehrfaches hashen
  - **Salt** verwenden, z.B. PW + Erstellungsdatum
- Weitere Verbesserung der Sicherheit:
  - Hashfunktion verwenden, die mehr Rechenzeit benötigt → erschwert Brute Force
  - User zu PW-Best-Practices zwingen → erschwert Wörterbuchangriffe

4 DATALOG

4.1 AUSWERTUNG REKURSIVER PROGRAMME (BLATT 05)

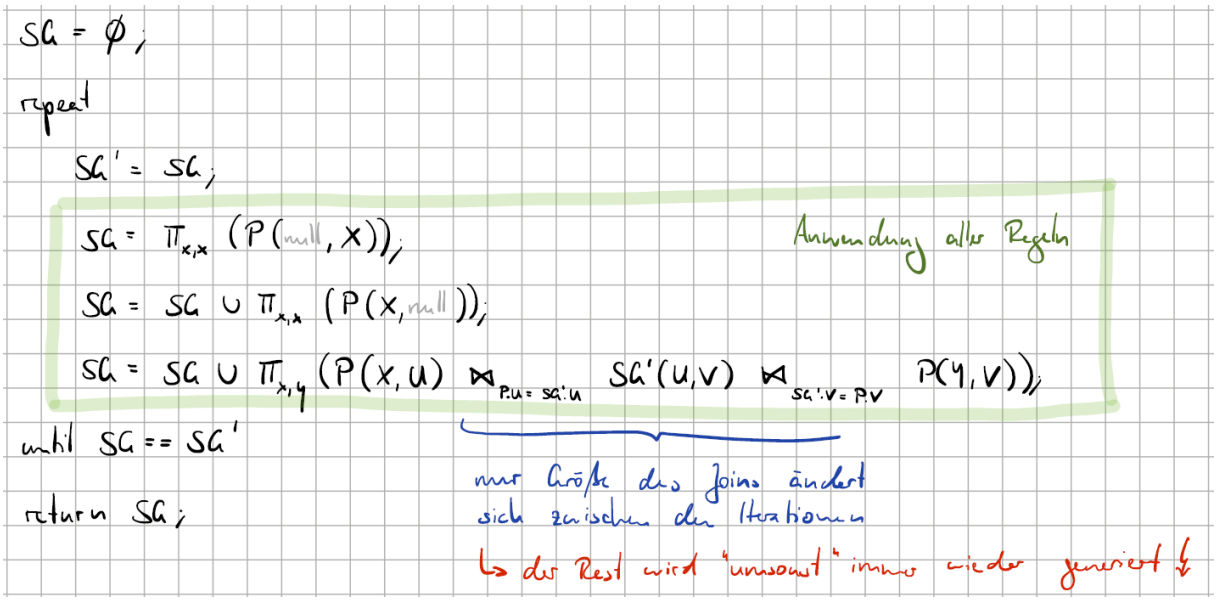
Beispiel: sind zwei Objekte Teil derselben Generation?

```
% trivial: dasselbe Objekt ist Teil derselben Generation
sameGeneration(X,X) :- parent(_,X).     % X ist Elternteil
sameGeneration(X,X) :- parent(X,_).     % X ist Kind

% bliebige Objekte sind in derselben Generation, wenn die Eltern in derselben
% Generation sind
sameGeneration(X,Y) :- sameGeneration(U,V), parent(X,U), parent(Y,V).
```

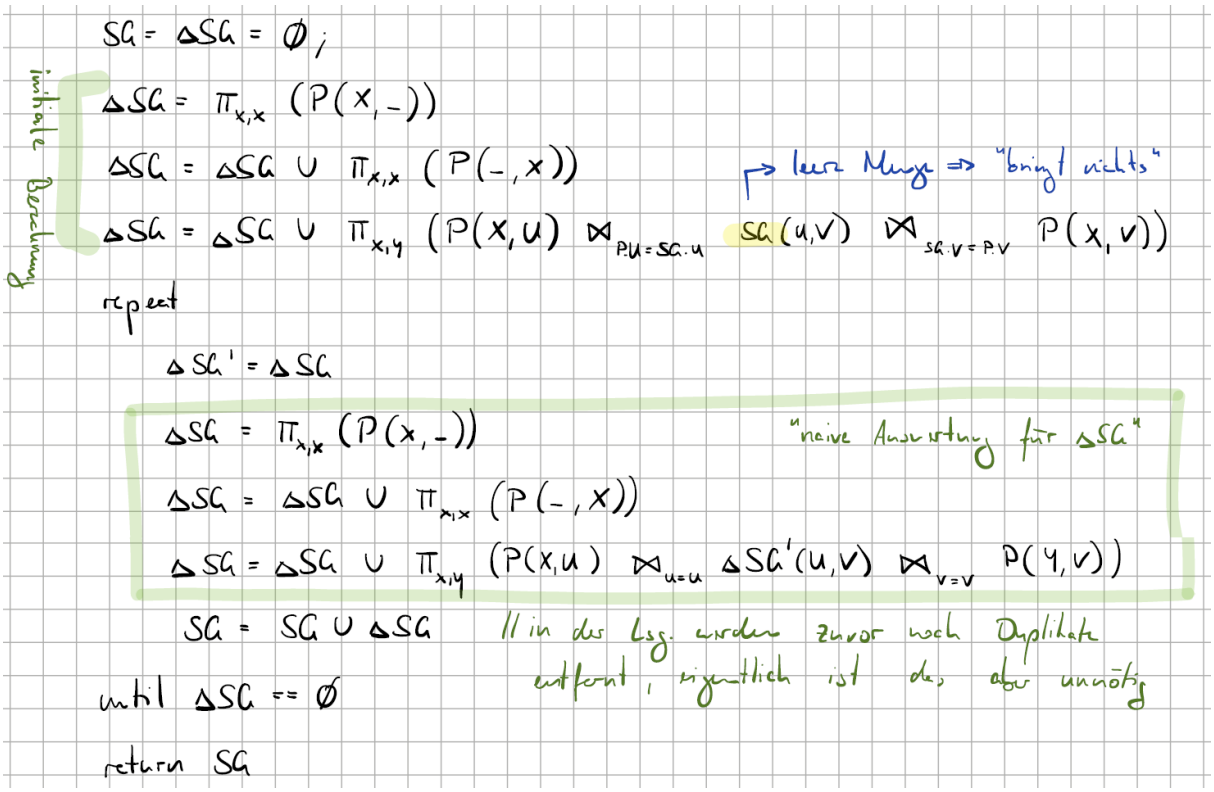
4.1.1 Naive Auswertung

In jeder Iteration wird viel Arbeit unnötigerweise wiederholt (**im Beispiel:** nur der Join ändert sich zwischen den Iterationen.



4.1.2 Semi-naive Auswertung

Statt alle Regeln voll auszuwerten, wird in jeder Iteration nur ein **Delta** berechnet:



4.1.3 Datalog-Sicherheit

Stratifizierte Programme: relevant für Programme mit negierten Prädikaten

- $p :- q_1, \dots, q_m, not(q_{m+1}), \dots, not(q_n)$
- Alle negierten Prädikate $q_{m+1}, \dots, q_n$ müssen unabhängig von p sein (→ keine Zyklen)
- Gleichbedeutend mit: alle **negierten Prädikate müssen schon vorher materialisiert** sein

Sichere Programme: durch Datalog erzeugte Relationen müssen endlich sein, d.h. …

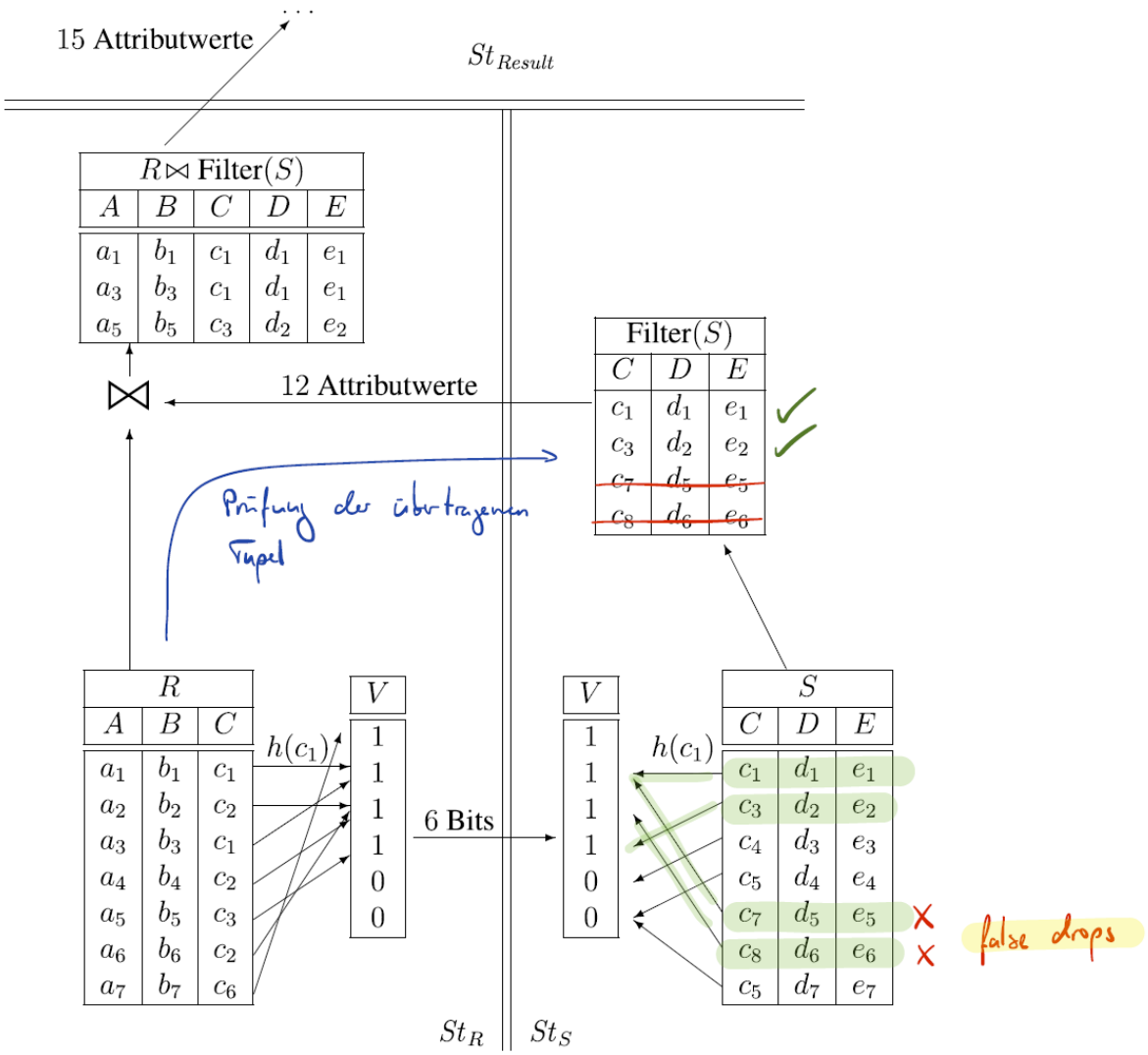
- $p(X) :- \cdots$
- Jede vorkommende Variable ist eingeschränkt, d.h.
  - X kommt in einem Prädikat vor ODER
  - $X = const.$ ODER
  - $X = Y$, wobei Y schon nachgewiesen ist
- Beispiel eines unsicheren Programms: $ungleich(X,Y) :- X \backslash= Y$

5 VERTEILTE DBMS

5.1 JOINS – BLOOM-FILTER (BLATT 05)

Herausforderung: **Joins von Relationen, die sich auf verschiedenen Stationen des VDBMS befinden**. Die triviale Vorgehensweise wäre alle Daten auf einer Station zu sammeln und dort zu joinen, das würde aber i.A. einen massiven Kommunikationsoverhead bedeuten. Stattdessen:

1. Schlüsselattribute der Ursprungsrelation R mit einer Hashfunktion h auf einen Bitvektor abbilden → den **Bloom-Filter**
2. Filter an die andere Station senden
3. Andere Station bildet ebenfalls alle Schlüsselattribute über h auf den Bitvektor ab
4. Für den Join werden **nur die Elemente zurückgeschickt, die durch h auf eine 1 im Bloom-Filter projiziert wurden**

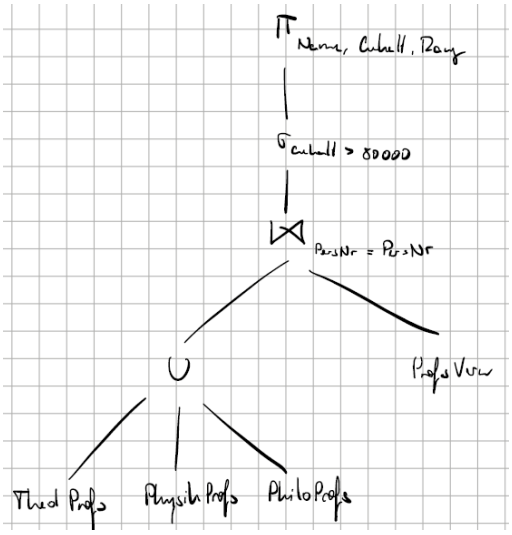


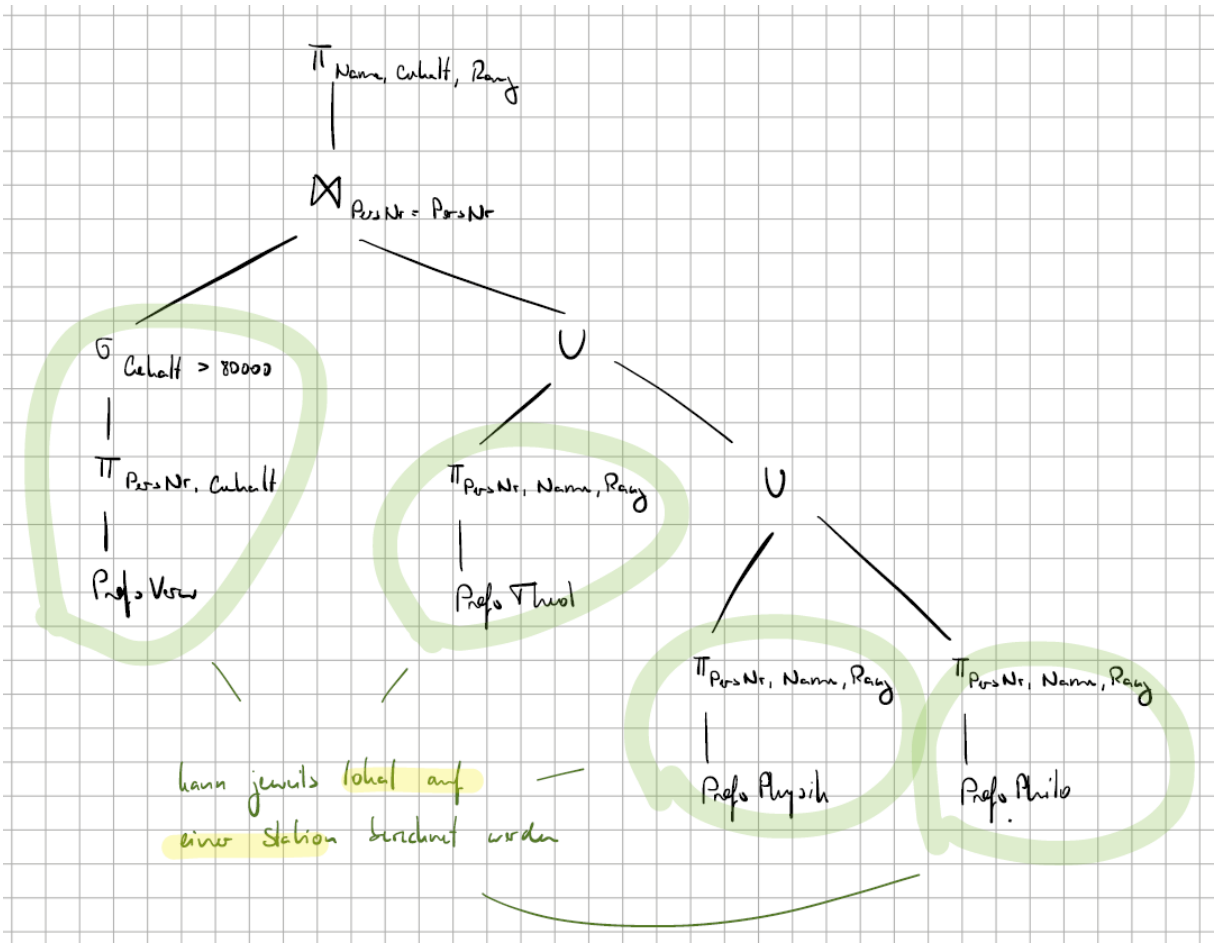
False Drops: Die Projektion durch h auf den Bitvektor ist nicht eineindeutig. Es kann also einzelne Tupel geben, die auf eine 1 abgebildet werden, obwohl sie eigentlich nicht zum Join gehören.

- False Drops werden bei der Durchführung des eigentlichen Joins aussortiert
- Eine False-Positive-Rate kann berechnet werden als $\frac{\Sigma false\ positive}{\Sigma matches}$

5.2 Anfrageoptimierung (Blatt 06)

Die Arbeit, die parallel auf einzelnen Stationen geleistet werden kann, sollte maximiert werden. Dazu bietet es sich an, Umformulierungen der relationalen Algebra zu nutzen, z.B.:





5.3 Quorum-Consensus Verfahren (Blatt 06)

In VDBMS gibt es u.U. viele Kopien desselben Datums A. Lesen einer Kopie ist unproblematisch, Schreiben kann aber zu Inkonsistenz führen. Das QC-Verfahren...

- Sperrenbasiertes Verfahren, wobei **eine einzelne S- / X-Sperre nicht ausreicht**
- Kopien des Datums haben Gewichte, sodass $W(A) := \sum_{i=1}^{n} w(A_i)$
- Lese- und Schreibquorum sind die Mindestanzahl an S- / X-Sperren, die eine Tx erwerben muss, um lesen / ändern zu dürfen:
    - Zum Schreiben sind mehr als die Hälfte der X-Sperren nötig: $2Q_w(A) > W(A)$
    - Während des Schreibens kann nicht gelesen werden: $Q_r(A) + Q_w(A) > W(A)$

5.3.1 Write-all / read-any-Prinzip

- **Write-all:** wenn eine Tx ein Datum ändert, müssen auch alle Kopien aktualisiert werden
    - ➔ Änderungs-Tx sehr aufwendig
- **Read-any:** somit kann von einer Tx jede beliebige Kopie gelesen werden
    - ➔ Lese-Tx sehr performant

- Weiteres Problem: wenn eine Station, die eine der Kopien hält, nicht verfügbar ist, muss die Änderungs-Tx warten
- Write-all / read-any ist ein Spezialfall des QC-Verfahrens, wobei
    - $W(A) = 1$
    - $Q_w(A) = W$ (write-all)
    - $Q_r(A) = 1$ (read-any)
- Gut geeignet für Online Analytical Processing (OLAP)

6 Big Data

6.1 K-Means Clustering

Optimierungsbedingung: Optimiert wird die Summe der Abstände aller Punkte zum Mittelpunkt des Clusters, dem sie zugeordnet wurden.

Iteration:

1. Punkte neu zu Clustern zuordnen (jeweils der mit dem nächstgelegenen Mittelpunkt)
2. Clustermittelpunkte neu berechnen (Mittel der Koordinaten aller beteiligten Punkte)

Terminierungsbedingungen:

- Clusterzentren ändern sich nicht mehr bzw. die Zuordnung der Punkte ändert sich nicht
- Eine vorher definierte max. Anzahl Iterationen ist erreicht

6.2 Top-k Anfragen

Top-k Anfragen geben nur k Elemente zurück, die hinsichtlich der Selektions- / Optimierungsbedingungen am besten sind.

6.2.1 Threshold-Algorithmus

- Alle Ausgangsrelationen **sortiert** von „besser“ nach „schlechter“ bzgl. Bewertungskriterium
- In jeder Iteration:
    - Eine Zeile in allen Ausgangsrelationen weiterrücken
    - Falls Objekt zum ersten Mal gelesen: suche die zugehörigen Objekte aus den anderen Relationen (wahlfrei / **Random Access**)
    - Füge neue Zeile ins Zwischenergebnis ein

    - Aktualisiere **Threshold** im Zwischenergebnis = Kombination der besten aktuellen Werte in jeder Kategorie
- **Terminierung,** wenn k Objekte im Zwischenergebnis besser sind als der Threshold

6.2.2 Non-Random-Access (NRA) Algorithmus

- Random Access zur Vervollständigung des Zwischenergebnisses wird vermieden
- Stattdessen sind die **Zwischenergebnisse zunächst variabel**
    - In jeder Iteration (i.e. mit jeder neuen Zeile) wird das Zwischenergebnis mit den neuen „Erkenntnissen“ aktualisiert
    - Abbruch, wenn die Werte der obersten k Elemente final feststehen

6.2.3 A-priori Algorithmus

Ziel ist **Frequent Itemsets** zu ermitteln, also Produkte, die häufig im Verbund gekauft werden.

Metrik $\boldsymbol{Support(A)} = Anzahl\ an\ Käufen\ von\ A$

Metrik $\boldsymbol{Confidence(A \rightarrow B)} = \frac{Support(A \cap B)}{Support(A)}$

Ablauf:

- Falls Initialschritt (k=1): erstelle Itemsets I_1 also ein Set für jedes einzelne Produkt
- Sonst: erstelle Itemsets I_k von jeweils k Produkten, die zusammen gekauft wurden
    - Jedes I_k mit Support < minsupport wird verworfen
    - Jedes I_k das ein verworfenes I_{k-1} enthält wird verworfen → **A-priori Eigenschaft:** jede Teilmenge eines Frequent Itemset ist selbst Frequent Itemset