Grundlagen: Datenbanken

TL;DR

(Version: 6. April 2021)

Zusammenfassung von Robyn Kölle zur Vorlesung Grundlagen: Datenbanken von Prof. Alfons Kemper (WS20/21). Das Skript kann Fehler enthalten, ich übernehme keinerlei Haftung. Fehler bitte melden.

Robyn Kölle

Grundbegriffe

- **Superschlüssel** sind Mengen von Attributen, die ein Tupel in einer Relation eindeutig identifizieren
- **Kandidatenschlüssel** sind minimale Superschlüssel
- Ein ausgewählter Kandidatenschlüssel heißt Primärschlüssel (primary key)
- Für Kandidatenschlüssel $K_1 \dots K_n$ einer Relation R heißt ein Attribut $A \in R \setminus (K_1 \cup \dots \cup K_n)$ **nicht prim / non-prime**. Attribute von $K_1 \dots K_n$ sind **prim**.
- **Funktionale Abhängigkeit (FD):** B ist funktional abhängig von A ($A \longrightarrow B$), wenn es zu jedem A höchstens ein B gibt
- Sei K ein Kandidatenschlüssel. B ist **vollständig / voll funktional** abhängig von K, wenn B von K funktional abhängig ist, aber nicht von Teilen von K.

  Beispiel:

prim	prim	nicht-prim
MatrNr	Spielt	Name
1	Minecraft	Chad
2	Fortnite	Virginia
1	CS:GO	Chad

$K = \{MatrNr, Spielt\}$

Name ist nicht vollständig funktional abhängig von K, weil $MatrNr \longrightarrow Name$ und $|K \setminus \{MatrNr\}| \neq 0$.

- **Multi-Valued-Dependency (MVD):** $\alpha \twoheadrightarrow \beta$ gilt, genau dann wenn:

$\exists t_1, t_2$ (Tupel) s.d. $t_1.\alpha = t_2.\alpha$

$\Longrightarrow \exists t_3, t_4$ s.d. $t_1.\alpha = t_2.\alpha = t_3.\alpha = t_4.\alpha \;\wedge$

$t_1.\beta = t_3.\beta \wedge t_2.\beta = t_4.\beta \;\wedge$

$t_2.\gamma = t_3.\gamma \wedge t_1.\gamma = t_4.\gamma$

M.a.W.: 2 (oder mehr) Attribute hängen beide von einem dritten Attribut ab.

Beispiel:

	α MatrNr	β Fach	γ Hobby
t_1	1	ERA	Linux-Zeitschriften lesen
t_2	1	GBS	Programmieren
t_3	1	ERA	Programmieren
t_4	1	GBS	Linux-Zeitschriften lesen
	2	ANAL	Analysieren

MatrNr $\twoheadrightarrow$ Fach

MatrNr	Fach-Möglichk.	Rest-Mögl.
1	ERA	Linux-Z.
	GBS	Programmieren

Für gleiche MatrNr gibt es alle Kombinationen aus Fach und [Rest] falls MatrNr $\twoheadrightarrow$ Fach

Komplementregel:

Sei $R := \{A, B, \ldots\}$

Falls $A \twoheadrightarrow B$, dann auch $A \twoheadrightarrow R \setminus B$

- **Anomalien** durch unexzellentes Datenbank-Design:
  - **Insert-Anomalie**: Daten können nicht eingefügt werden weil Primary-Key-Attribute unbekannt sind. Im obigen Beispiel: man kann keine neue Person einfügen, ohne zu wissen, was sie spielt.
  - **Update-Anomalie**: Das Aktualisieren eines Tupels kann zu inkonsistenten Daten führen. Im obigen Beispiel: Wenn Chad sich umbenennt in Chaddington, müssen alle Tupel, wo Chad vorkommt, aktualisiert werden, um Inkonsistenzen zu vermeiden.
  - **Delete-Anomalie**: Durch das Löschen eines Datensatzes gehen Informationen verloren. Im obigen Beispiel: Wenn Virginia verbergen möchte, dass sie Fortnite spielt, geht auch die Information verloren, dass ihr Name zur MatrNr 2 gehört.

Normalformen

1. Normalform (1NF)

Alle Attribute sind **atomar**.

Beispiel:

Preis
15 €
12 $

⟼

Preis	Währung
15	EUR
12	USD

2. Normalform (2NF)

Non-Prime Attribut abh. von einem Teil eines Kandidatenschlüssels

1 NF **und**:

Jedes nicht-Schlüsselattribut muss **voll funktional abhängig von jedem Kandidatenschlüssel** sein. M.a.W.: es gibt keine partial dependencies.

Beispiel:

abh. von einem Teil des PK

MatrNr → Name		Spielt
1	Chad	Minecraft
2	Virginia	Fortnite
1	Chad	CS:GO

⟼

MatrNr	Name
1	Chad
2	Virginia

MatrNr	Spielt
1	Minecraft
1	CS:GO
2	Fortnite

Primary Key: (MatrNr, Spielt)

aber MatrNr → Name

3. Normalform (3NF)

2 NF **und**:

Für alle FDs $(\alpha \rightarrow \beta)$ gilt:

trivial $(\beta \subseteq \alpha)$ ∨ α ist Super Key ∨ β ist Teil eines Kandidatenschlüssels

(M.a.W.: kein nicht-Schlüsselattribut hängt von einem nicht-Schlüsselattribut ab)

Beispiel:

prim | nicht-prim

ID	Name	PLZ →	Bundesland
1	Chad	80800	Bayern
2	Chad	80801	Bayern
3	Kevin	66111	Saarland

⟼

ID	Name	PLZ
1	Chad	80800
2	Chad	80801
3	Kevin	66111

PLZ	Bundesland
80800	Bayern
66111	Saarland
80801	Bayern

Bemerkung: mit dem Synthesealgorithmus kann man Relationen in 3NF bringen.

Boyce-Codd-Normalform (BCNF)

3 NF und:

Für alle FDs $(\alpha \rightarrow \beta)$ gilt:

trivial $(\beta \subseteq \alpha)$ ∨ α ist Super-Key

Beispiel:

Rechnung	ArtNr	ArtName	Anzahl
10	1	Artikel	69
10	2	Martikel	420
11	1	Artikel	3

⟼

Rechnung	ArtNr	Anzahl
10	1	69
10	2	420
11	1	3

ArtNr	ArtName
1	Artikel
2	Martikel

Schlüsselkandidaten:

(Rechnung, ArtNr), (Rechnung, ArtName)

ArtNr → ArtName aber ArtNr ist kein Super-Key

Bemerkung: mit dem Dekompositionsalgorithmus kann man Relationen in BCNF bringen.

4. Normalform (4NF)

BCNF und :

Für alle MVDs ($\alpha \twoheadrightarrow \beta$) gilt (Sei R die Relation):

trivial ($\beta \subseteq \alpha \vee \alpha \cup \beta = R$) $\vee$ α ist Super-Key

Beispiel:

MatrNr	Fach	Hobby
1	ERA	Linux-Zeitschriften lesen
1	GBS	Programmieren
1	ERA	Programmieren
1	GBS	Linux-Zeitschriften lesen
2	ANA2	Analysieren

⟼

MatrNr	Fach
1	ERA
1	GBS
2	ANA2

MatrNr	Hobby
1	...
1	...
2	...

mehrwertige Abhängigkeit

MatrNr $\twoheadrightarrow$ Fach
MatrNr $\twoheadrightarrow$ Hobby

kein Schlüsselkandidat

Relationale Entwurfstheorie

Kanonische Überdeckung

= nicht reduzierbare Menge von FDs.

Wird benötigt für den Synthesealgorithmus.

Gegeben eine Menge F von FDs.

Die kanonische Überdeckung zu F erhält man wie folgt:

1) FDs der Form $\alpha \longrightarrow \beta$ mit $\beta = \{\beta_1, \ldots, \beta_n\}$ aufteilen in

   $\alpha \longrightarrow \beta_1, \ldots, \alpha \longrightarrow \beta_n$ (z.B. $AB \longrightarrow CD$: $AB \longrightarrow C$, $AB \longrightarrow D$)

2) Linksreduktion

   Bei FDs, die auf der linken Seite mehrere Attribute haben, prüfen ob eines (oder mehrere) überflüssig

3) Redundante FDs entfernen

Beispiel:

F:

$A \longrightarrow B$			$A \longrightarrow B$
$AB \longrightarrow C$		1)	$AB \longrightarrow C$
$D \longrightarrow AC$		$\longmapsto$	$D \longrightarrow A$
$D \longrightarrow E$			$D \longrightarrow C$
			$D \longrightarrow E$

→ 2): Betrachte $AB \longrightarrow C$

Ist A überflüssig? D.h., komme ich mit den anderen FDs (über mehrere Schritte) von B nach A?

$B \longrightarrow \lightning$ $\Longrightarrow$ nein

B überflüssig?

$A \longrightarrow B$ (erste FD) $\Longrightarrow$ ja.

$A \longrightarrow B$		$A \longrightarrow B$
~~AB~~ $\longrightarrow C$	2) $\longmapsto$	$A \longrightarrow C$
$D \longrightarrow A$		$D \longrightarrow A$
$D \longrightarrow C$		$D \longrightarrow C$
$D \longrightarrow E$		$D \longrightarrow E$

$\longrightarrow$ 3): Betrachte jede FD einzeln.

$\alpha \longrightarrow X$ ist redundant, wenn $X \in \alpha^+$ (in $F \setminus \{\alpha \longrightarrow X\}$)

Betrachte $A \longrightarrow B$:

A^+: C $\Longrightarrow B \notin A^+ = \{A, C\}$ $\Longrightarrow$ nicht redundant

Betrachte $A \longrightarrow C$:

A^+: B $\Longrightarrow C \notin A^+ = \{A, B\}$ $\Longrightarrow$ nicht redundant

Betrachte $D \longrightarrow A$:

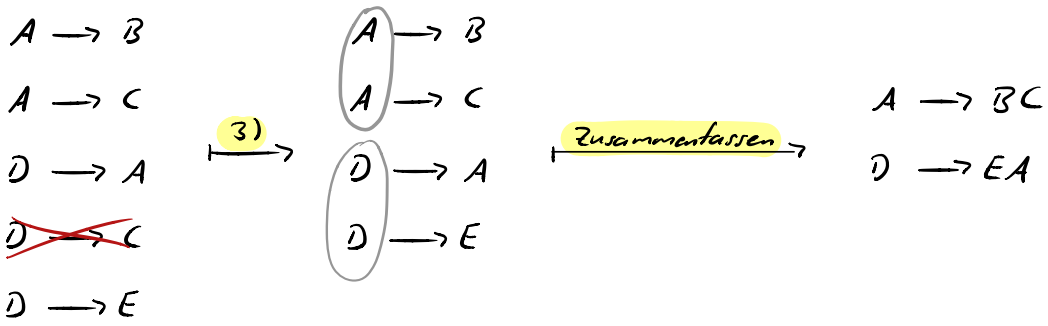
D^+: $D \nearrow C$, $D \searrow E$ $\Longrightarrow A \notin D^+ = \{D, C, E\}$ $\Longrightarrow$ nicht redundant

Betrachte $D \longrightarrow C$:

D^+: $D \nearrow E$, $D \searrow A \nearrow B$, $A \searrow C$ $\Longrightarrow C \in D^+ = \{A, B, C, D, E\}$ $\Longrightarrow$ redundant

Betrachte $D \longrightarrow E$:

D^+: $D \nearrow A \longrightarrow B$, $D \searrow C$ $\Longrightarrow E \notin D^+ = \{A, B, C, D\}$ $\Longrightarrow$ nicht redundant



Syntheseal gorithmus

Bringt eine Relation R mit einer Menge an FDs F in 3NF.

Vorgehen:

1) Kanonische Überdeckung F' von F bestimmen
2) $\forall_{(\alpha \to \beta) \in F'}$: Definiere $R_\alpha := \alpha \cup \beta$
3) Sei K Kandidatenschlüssel von R. Falls $\nexists R_i$ s.d. $K \subseteq R_i$, definiere $R_K := K$
4) Falls für zwei erhaltene Relationen R_i, R_j gilt $R_i \subseteq R_j$: Eliminiere R_i, und 4)

Obiges Beispiel:

durch 1) erhalten wir $A \to BC$, $D \to EA$

→ 2): $R_1 := \{\underline{A}, B, C\}$

$R_2 := \{\underline{D}, E, A\}$

→ 3): $K = \{D\}$ ist der einzige Kandidatenschlüssel.

$K \subseteq R_2 \implies$ tu nichts

$\longrightarrow$ 4): $R_1 \not\subseteq R_2 \;\wedge\; R_2 \not\subseteq R_1 \Longrightarrow$ tu nichts

Ergebnis: $R_1 := \{\underline{A}, B, C\}$, $R_2 := \{\underline{D}, E, A\}$ } Erfüllt 3NF

Dekompositionsalgorithmus nicht abhängigkeitsbewahrend

Bringt eine Relation R mit einer Menge an FDs F in BCNF. bzw. 4NF

Vorgehen:

1) $M := \{R\}$
2) Terminiere, falls $\nexists R_i \in M$ s.d. R_i ist nicht in BCNF bzw. 4NF
3) Für alle nicht-triviale ($\alpha \longrightarrow \beta$) bzw. ($\alpha \twoheadrightarrow \beta$) $\in R_i$ mit

   $\alpha \cap \beta = \varnothing \;\wedge\; \alpha \cup \beta \subset R_i \;\wedge\; \alpha \not\longrightarrow R_i$:

   Definiere $R_1 := \alpha \cup \beta$, $R_2 := R_i - \{\beta\}$

   Füge R_1 und R_2 in M ein, entferne R_i aus M. $\longrightarrow$ 2)

Beispiel:

$R = \{[A, B, C, D]\}$ $M = \{R\}$

$\{A, B\} \longrightarrow \{D\}$: $\{A,B\} \cap \{D\} = \varnothing$, aber $\{A,B\} \longrightarrow R \Longrightarrow$ OK

$\{B\} \longrightarrow \{C\}$: $\{B\} \cap \{C\} = \varnothing$, $\{B\} \not\longrightarrow R \Longrightarrow$ nicht OK

$\{C\} \longrightarrow \{B\}$

Betrachtete FD: $\{B\} \longrightarrow \{C\}$

$R_1 := \{[\underline{B}, C]\}$

$R_2 := \{[\underline{A}, \underline{B}, D]\}$ $M = \{R_1, R_2\}$

R_1 und R_2 sind in BCNF (jeweils gilt: $\forall_{(\alpha \rightarrow \beta)}$: trivial $\beta \subseteq \alpha$ oder α ist Superschlüssel) $\Longrightarrow$ terminiere.

Indexstrukturen

B-Bäume

1 Knoten ⟷ 1 Page im Speicher

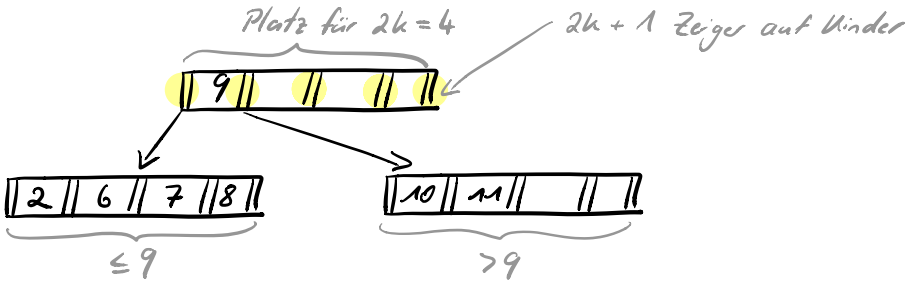
Max. Anzahl der Seitenzugriffe ⟷ Höhe des Baums

B-Baum mit Grad k:

- jeder Knoten: mind. k, höchstens $2k$ Einträge (Wurzel 1 bis $2k$)
- Einträge innerhalb Knoten sortiert
- Knoten mit n Einträgen hat $n+1$ Kinder (außer Blätter)
- Kinder links von einem Key k: $\leq k$, rechts: $> k$

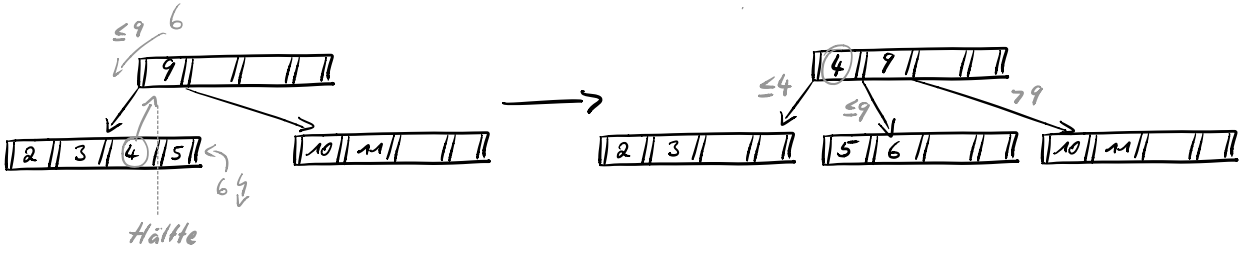
Beispiel:

$k = 2$



Falls beim Einfügen der Knoten voll ist: teilen

Beispiel:



Falls Wurzel voll: neue Wurzel, teilen analog

Key k löschen:

- k in Blatt ? ⟹ einfach löschen
- k in innerem Knoten ? ⟹ mit nächst-kleinerem k' tauschen (aus Blatt), k dann dort (im Blatt) löschen
- Blattknoten danach unterbelegt ?

  ⟹ Nachbar auch unterbelegt ? ⟹ mit Nachbar und zugehörigen Parent-Key verschmelzen

  Sonst ⟹ Inhalte mit Nachbar ausgleichen (beide Inhalte neu verteilen)

Ggf. muss dann im Vaterknoten die gleiche Fehlerbehandlung stattfinden

Ausführlich: siehe GAD - TR: DR → (a,b)-Bäume

Erweiterbares Hashing

Keys invers binär, z.B. $2_{10} = 10_2 \longmapsto 01_2$

Starte mit globaler Tiefe $t = 1$, d.h. unterscheide Keys nur anhand des ersten Bits, und lokaler Tiefe $t' = 0$ für die Buckets mit Kapazität b:

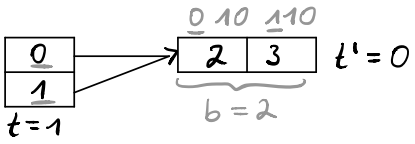
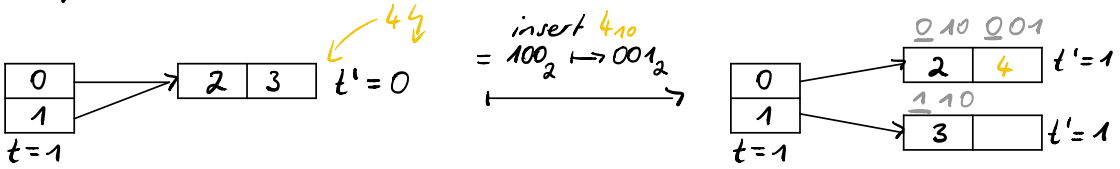


Tabelle (Hauptspeicher) Buckets (Disk)

Falls ein Key nicht eingefügt werden kann, weil der jeweilige Bucket voll ist, unterscheide 2 Fälle:

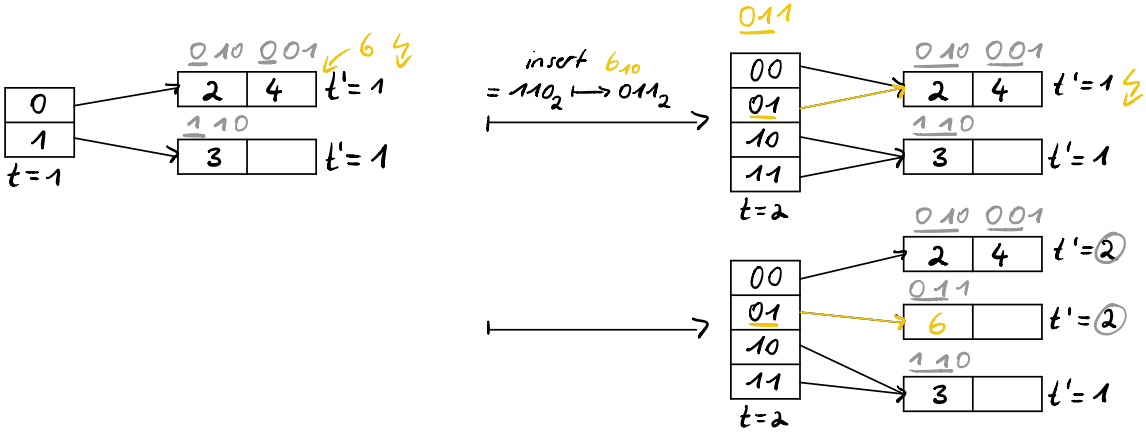
- $t' < t$: erhöhe t', erzeuge neuen Bucket und teile neu auf

Beispiel:



- $t' = t$: verdopple die Kapazität der Tabelle, und füge dann ein

Beispiel:



Anfrageoptimierung

Äquivalenzerhaltende Transformationen

1) **Konjunktion in Selektion aufbrechen**

$$\sigma_{a \wedge b}(R) \equiv \sigma_a(\sigma_b(R))$$

2) **σ ist kommutativ**

$$\sigma_a(\sigma_b(R)) \equiv \sigma_b(\sigma_a(R))$$

3) **Projektion-Kaskaden.** Falls $\alpha_1 \subseteq \alpha_2 \subseteq \ldots \subseteq \alpha_n$, dann:

$$\pi_{\alpha_1}(\ldots(\pi_{\alpha_n}(R))) \equiv \pi_{\alpha_1}(R)$$

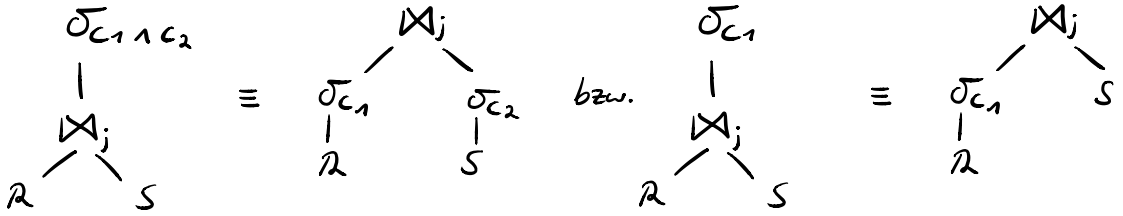
4) **Selektion und Projektion vertauschen.** Falls c sich auf $A_1, \ldots, A_n$ bezieht, dann:

$$\pi_{A_1,\ldots,A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1,\ldots,A_n}(R))$$

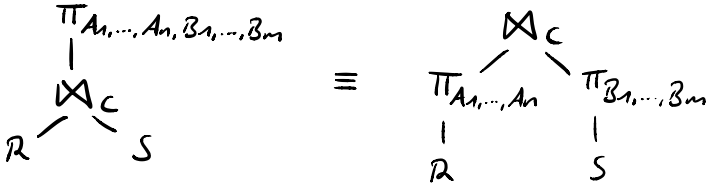
5) $\times$, $\cup$, $\cap$, $\bowtie$ sind kommutativ:

$$R \bowtie_c S \equiv S \bowtie_c R$$

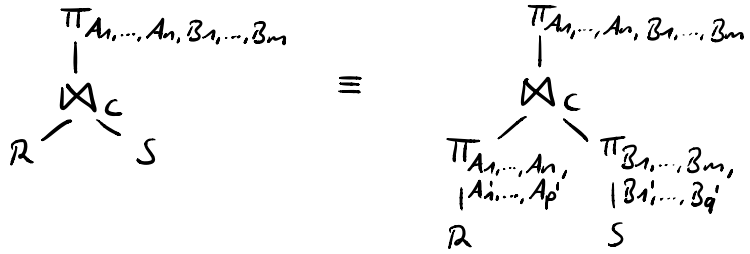
6) Selektion und Join vertauschen. Falls c_1 sich auf Attribute von R bezieht, und c_2 auf Attribute von S, dann:



7) Projektion und Join vertauschen. Seien A_i Attribute aus R, B_i Attribute aus S, und $\mathcal{L} := \{A_1, \dots, A_n, B_1, \dots, B_m\}$ die Projektionsliste. Falls c sich nur auf Attribute in $\mathcal{L}$ bezieht, dann:



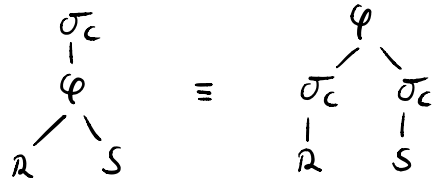
Bezieht c sich auf weitere Attribute $A_1', \dots, A_p', B_1', \dots, B_q' \notin \mathcal{L}$, dann:



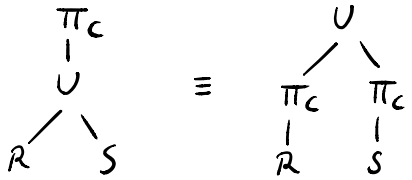
8) Assoziativität von $\bowtie$, $\times$, $\cup$, $\cap$. Sei $\varphi \in \{\bowtie, \times, \cup, \cap\}$. Dann:

$$(R \varphi S) \varphi T \equiv R \varphi (S \varphi T)$$

9) **Distributivität von σ mit $\cup, \cap, -$.** Sei $\varphi \in \{\cup, \cap, -\}$. Dann:



10) **Distributivität von π mit $\cup$.**



11) **De Morgan** (für Join- & Selektionsprädikate)

$$\neg(a \wedge b) \equiv \neg a \vee \neg b$$

$$\neg(a \vee b) \equiv \neg a \wedge \neg b$$

12) **Kreuzprodukt mit Selektion zu Join.** Sei A ein Attribut von R, B ein Attribut von S. Dann:

Joins

https://de.wikipedia.org/wiki/Joinalgorithmen

- **Nested Loop:**

  Betrachte $R \bowtie_* S$:

  $\forall r \in R$:
  $\quad \forall s \in S$:
  $\quad\quad r * s$?

  Gut für Kreuzprodukte, z.B.: select * from R, S

  $\Rightarrow$ Ergebnismenge zwangsläufig Kardinalität $|R \times S|$

  $\Rightarrow$ Overhead (vgl. mit anderen Joins) sparen

- **Blockwise Nested Loop:**

  Wie Nested Loop, aber R wird gruppiert, und statt über alle $r \in R$ zu iterieren, wird nur noch für jede Gruppe aus R über jedes $s \in S$ iteriert.

- **Index-Join:**

  Es existiert eine Indexstruktur (z.B. B^+-Baum, Hashtable) für bestimmte Attribute. Anzahl d. Vergleiche abh. von Indexstruktur. Der Primary Key ist immer indiziert.

  Gut für range queries (z.B. „alle die 18 oder älter sind") ($\longrightarrow$ Baum), oder Equi-Joins (z.B. „select * from R, S where $x = y$) ($\rightarrow$ Hashtable).

  Trade-off: Speicher-Overhead

- **Sort-Merge-Join:**

  R, S beide nach dem Join-Attribut sortiert (bereits vor dem Join).

  Ergebnis durch einmaliges Scannen beider Relationen.

  Nur geeignet für Equi-Joins und Natural Joins.

Beispiel:

```
select *
from R, S
where x=y
order by x
```

⇒ Es muss danach sortiert werden, also kann die Sortierung auch bereits vor dem Join durchgeführt werden.

- **Hash Join:**

Hashtabelle mit Join-Attribut von R. Einträge $s \in S$ werden nur mit Einträgen $r \in R$ verglichen, falls $h(s) = h(r)$.

Beispiel:

```
select S.Name, P.Phone
from Student S, PhoneNums P
where
    S.MatrNr = P.MatrNr
```

(Student.MatrNr indiziert)

Trade-Off: Speicher, Hashing (→ Tabelle vergrößern, Buckets teilen, ...)

Kostenmodelle

Selektivität $sel_p := \frac{|\sigma_p(R)|}{|R|}$

Selektivität bei Joins: $sel_{i,j} := \frac{|R_i \bowtie_{p_{i,j}} R_j|}{|R_i \times R_j|}$

Beispiel:

$sel_{i,j} = 0{,}5$, $|R_i| = 10$, $|R_j| = 20$

$\Rightarrow |R_i \bowtie_{p_{i,j}} R_j| = 10 \cdot 20 \cdot 0{,}5 = 100$

Abschätzung der Selektivität:

- $sel_{R.A=c} = \frac{1}{count(A)}$
- Falls A Schlüssel von R ist: $sel_{R.A=c} = \frac{1}{|R|}$
- Bei Equi-Joins, falls $R.A$ Fremdschlüssel auf $S.B$ ist: $sel_{R.A=S.B} = \frac{1}{|S|}$

Transaktionsverwaltung

Transaktion: Folge von elementaren read/write Operationen

ACID

Transaktionen sollen die ACID-Kriterien erfüllen:

- **Atomicity**: „alles oder nichts"

  Falls Transaktionen nicht atomar sind: Inkonsistenzen möglich.

  Beispiel: Überweisung besteht aus 1) Kontostand verringern und 2) Kontostand erhöhen. Zustand inkonsistent, falls nur 1) oder nur 2) gespeichert wird

- **Consistency**:

  Konsistente Daten. Ohne Consistency: viele Probleme möglich

  Beispiel: Ein Student, mehrere Matrikelnummern (fehlerhaft) ⟹ Note welcher Matrikelnummer zuordnen?

- **Isolation**:

  Transaktionen müssen korrekt voneinander isoliert sein.

  Beispiel: zwei parallele Überweisungen könnten gleichzeitig Kontostände ändern, und somit kann der Kontostand nachher inkorrekt sein

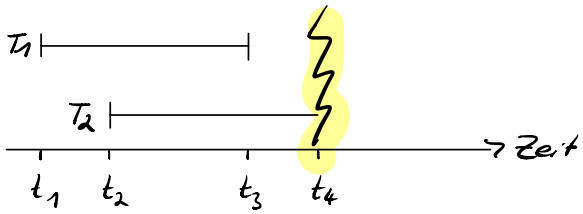
- **Durability**:

  Dauerhaftigkeit garantieren. Man muss darauf vertrauen können, dass ein Commit eine Transaktion festschreibt.

  Beispiel: Geldautomat soll erst Geld auszahlen, wenn die Datenbank garantieren kann, dass die Abbuchung festgehalten wurde.

Recovery *Atomacity, Durability gewährleisten*

Folgendes Szenario:



T_i: Transaktionen
t_i: Zeitstempel

t_4: **Absturz**, während T_2 nicht fertig

Aufgabe der Recovery: Zustand herstellen, s.d. T_1 fertig, und T_2 nicht begonnen hat

- **Redo**: Protokollierte, aber nicht in die Datenbasis aufgenommene (Teil)-Transaktion wiederholen
- **Undo**: Protokollierte, nicht vollständig übernommene Teil-Transaktion rückgängig machen

Fehlerbehandlung:

Änderungen an der **Datenbasis** (Hintergrundspeicher) erfordern einlagern der jew. Seiten in den **Datenbank-Puffer** (Hauptspeicher), wo die Änderungen vorgenommen werden.

- **steal**: Seiten können wieder (auf die Datenbasis) ausgelagert werden, auch wenn eine Transaktion im Puffer noch nicht abgeschlossen ist
- **¬steal**: Transaktionen hinterlassen vor dem Commit keine Spuren auf der Datenbasis $\Rightarrow$ rollback einer aktiven Transaktion muss nicht im Hintergrundspeicher ausgeführt werden.

Kleine Sperrgranulate: Eine Seite kann mehrere Transaktionen enthalten, deren

Daten einzeln gesperrt werden können (Änderungen von anderen Transaktionen verhindern) ⟹ ggf. hat die Seite sowohl abgeschlossene, als auch nicht-abgeschlossene Transaktionen

- **force**: Fertige Transaktionen werden direkt vom Puffer in die Datenbasis geschrieben
- **¬force**: Fertige Transaktionen können zu einem späteren Zeitpunkt in die Datenbasis übernommen werden, und werden protokolliert (Log-Datei), um Redo-Recovery zu ermöglichen.

	force	**¬force**
¬steal	· benötigt kein Redo · benötigt kein Undo	· benötigt Redo · benötigt kein Undo
steal	· benötigt kein Redo · benötigt Undo	· benötigt Redo · benötigt Undo

benötigt WAL

- **update-in-place**: geänderte Seite A' wird dorthin geschrieben, wo die ursprüngliche Seite A vorher war (⟹ alter Zustand geht verloren)
- **twin-block**: zwei Seiten P_{A1}, P_{A2}; bei Änderung im Puffer: überschreiben einer Seite ⟹ alter Zustand gesichert

Write-Ahead-Logging (WAL)

Benötigt bei steal ∧ ¬force (∧ update-in-place):

- Vor Commit einer Transaktion müssen alle Logs (aus dem Puffer) in die Datenbasis geschrieben werden
- Bevor eine modifizierte Seite ausgelagert wird, müssen alle Log-Einträge zu der Seite im Puffer- und im Hintergrundspeicher-Log sein

Logs

Transaktionen:

[LSN, TA, PageID, Redo, Undo, PrevLSN]

- LSN: Log Sequence Number, fortlaufend
- TA: Betroffene Transaktion
- PageID: Betroffene Page
- Redo: Was wurde gemacht?
- Undo: Wie kann ich das rückgängig machen?
- PrevLSN: LSN der vorherigen zur TA gehörigen Transaktion

Compensation Log Record (CLR):

<LSN, TA, PageID, Redo-Info, PrevLSN, UndoNextLSN>

- Redo-Info: Was wurde gemacht?
- UndoNextLSN: LSN der Aktion, die als Nächstes un-done werden muss

Beispiel-Log:

[#1, T_1, BOT, 0] ← Begin of Transaction (BOT) hat kein Undo / Redo, oder (offensichtlich) PrevLSN

[#2, T_1, P_X, X+=1, X-=1, #1]

[#3, T_3, BOT, 0]

[#4, T_1, COMMIT, #2] ← Kein Undo / Redo bei COMMIT

[#5, T_3, P_X, X*=10, X/=10, #3]

[#6, T_2, BOT, 0]

[#7, T_3, COMMIT, #5]

[#8, T_2, P_Y, Y*=2, Y/=2, #6]

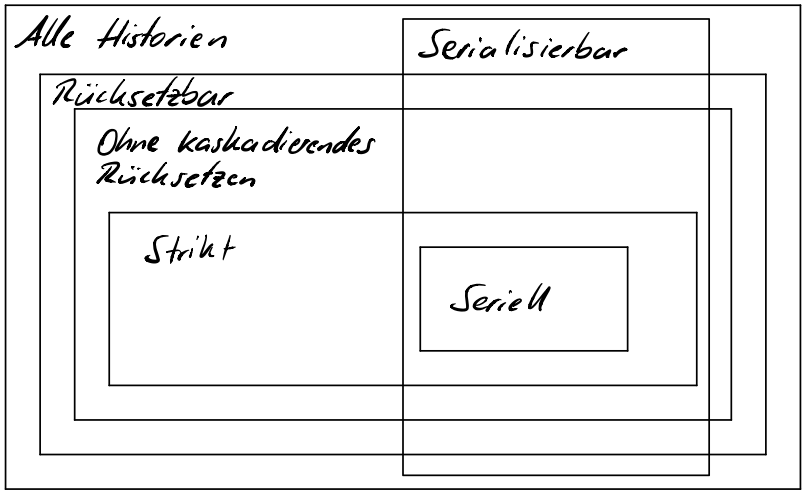
Ab hier Recovery

<#8', T_2, P_Y, Y/=2, #9', #7>

<#7', T_2, -, -, #8', 0>

- Undo eines BOT

Historien



$w_i(A)$: T_i schreibt in A
$r_i(A)$: T_i liest aus A
c_i : T_i commit
a_i : T_i abort

- Historien H, H' heißen äquivalent ($H \equiv H'$), falls sie Konfliktoperationen in der selben Reihenfolge ausführen.
- H heißt serialisierbar, wenn $H \equiv H_s$ (mit H_s seriell, d.h. die T_i sind nicht verzahnt)

Im Folgenden schreibt T_j in A, und T_i liest / schreibt danach A.

- H serialisierbar $\iff$ $SG(H)$ azyklisch (Serialisierbarkeitsgraph)
- H rücksetzbar: $\forall T_i : c_j <_H c_i$
- H ohne kaskadierendes Rücksetzen: $\forall T_i : c_j <_H r_i(A)$
- H strikt: $\forall T_i : c_j <_H (w/r)_i(A)$ bzw. $a_j <_H (w/r)_i(A)$

Serialisierbarkeitsgraph zu einer Historie:

Seien $p_i, q_j \in H$ (z.B. $r_i(A)$, $w_j(B)$, ...)

Falls $p_i <_H q_j$, sieht die Historie so aus:

$H = p_i \longrightarrow q_j$

Und der Serialisierbarkeitsgraph:

$SG(H) = T_i \longrightarrow T_j$ falls p_i und q_j Konfliktoperationen sind
z.B. $r_i(A)$, $w_j(A)$

Datenbank-Scheduler

Locks:

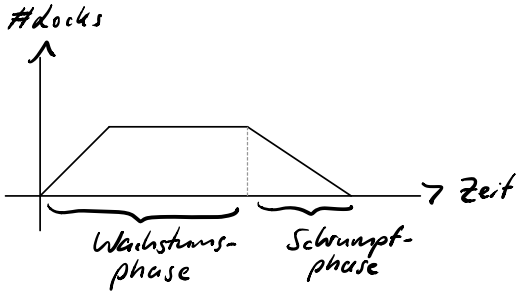
S: Lesesperre, shared (mehrere S Locks auf einem Objekt möglich)

X: Schreibsperre, exklusiv

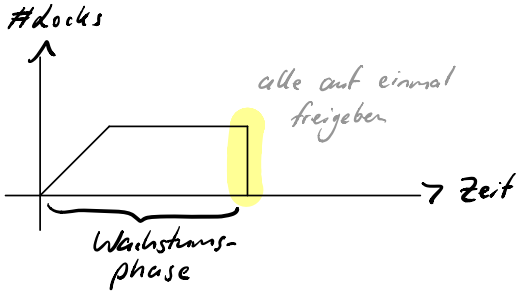
angeforderte Sperren \ existierende Sperren →	-	S	X
S	✓	✓	X
X	✓	X	X

2-Phasen-Sperrprotokoll (2PL):

- Falls T_i Objekt benutzt, muss es gesperrt werden (falls noch nicht gemacht)
- Wenn Sperre illegal (→ s. Tabelle oben), wartet T_i
- T_i durchläuft Wachstumsphase & Schrumpfphase
  - Wachstumsphase: Sperren anfordern, keine freigeben
  - Schrumpfphase: Sperren freigeben, keine anfordern
- Bei Transaktionsende muss T_i alle Sperren freigeben



2PL



Strikes 2PL

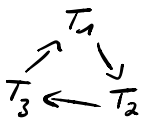
⟹ vermeidet kaskadierendes Zurücksetzen & Reihenfolge der T_i entspricht äquivalenter serieller Reihenfolge (garantiert)

Wartegraph (Deadlocks erkennen):

Falls T_i auf Freigabe der Sperre von T_j wartet: $T_i \longrightarrow T_j$

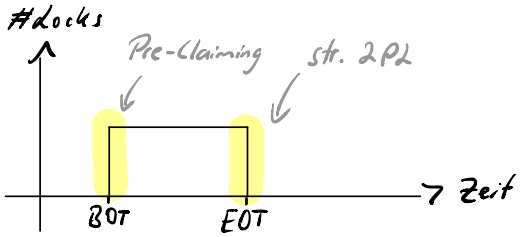
Deadlock ⟺ Wartegraph hat Zyklus

Beispiele:



$T_1 \longleftrightarrow T_2$

Preclaiming (Verklemmungen vermeiden):



⚠ Nicht praxistauglich: i.A. weiß man nicht im Vorhinein alle Objekte, auf die T_i zugreifen will.

Zeitstempel (Verklemmungen vermeiden):

Im Folgenden fordert T_1 die Sperre an, und muss auf T_2 warten

- wound-wait: T_1 älter als T_2 $\Rightarrow$ T_2 wird aborted, T_1 läuft weiter
  sonst $\Rightarrow$ T_1 wartet auf T_2
- wait-die: T_1 älter als T_2 $\Rightarrow$ T_1 wartet darauf, dass T_2 Sperre freigibt
  sonst $\Rightarrow$ T_1 wird aborted

Garantiert verklemmungsfrei

Nachteil: i.A. werden zu viele Transaktionen zurückgesetzt, die eigentlich ok wären