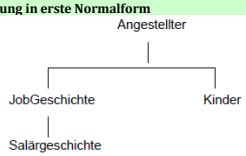
DBF - Normalisierung

Erste Normalform

Überführung in erste Normalform



- Die Segmente der im hierarchischen Baum angegebenen Typen mögen folgende Felder (Attribute) haben

```
Angestellter(AngNummer#, Name, Geburtstag,
   Jobgeschichte, Kinder)
Jobgeschichte(Jobdatum#, Titel,     SalärGe-
schichte
Salärgeschichte(Salärdatum#, Salär)
Kinder(KindName#, Geburtsjahr)
```

➔ Jobgeschichte, Kinder und Salärgeschichte sind Mehrfachattribute und entspr. jeweiligen Sub-Segmenten

- Überführung in erste Normalform besteht jetzt daraus, dass **von oben nach unten** die Schlüssel kaskadiert werden und die Mehrfachattribute weggelassen

```
Angestellter(AngNummer#, Name, Geburtstag)
JobGeschichte(AngNummer#, Jobdatum#, Titel)
SalärGeschichte(AngNummer#, Jobdatum#,
Salärdatum#, Salär)
Kinder(AngNummer#, KindName#, Geburtsjahr)
```

➔ Geht nur, wenn Primärschlüssel selber atomar, also nicht zusammengesetzt sind

Funktionale Abhängigkeiten

FD = Functional Dependency

- Sei U eine Menge von Attributen
  ➔Eine **FD über U** ist ein Ausdruck der Form X ➔ Y mit nichtleeren Teilmengen X und Y von U

Ist das Relationenformat
`Mitarbeiter(M#, Abt, Div)`
gegeben, so wären zum Beispiel
`{Abt, Div}` ➔ `{Abt}`, oder
`{M#}` ➔ `{Abt, Div}`, oder
`{Div}` ➔ `{Div}`
funktionale Abhängigkeiten über
`Attr{Mitarbeiter}={M#, Abt, Div}`

(Semantische) Gültigkeit einer FD in einer Relation

Relationenformat

Attr(R)

Rel(R)

- Sei $R(A_1, A_2, \dots, A_m)$ ein Relationenformat
- Wir bezeichnen mit $Attr(R) = \{A_1, A_2, \dots, A_m\}$ die Menge der Attribute von R und mit $Rel(R)$ die Menge der Relationen r zum Format R

- Die funktionale Abhängigkeit X ➔ Y über Attr(R) ist gültig in der relation $r \in Rel(R)$, in Zeichen:
$r \vDash X \to Y$, wenn $\forall t, s \in r(t(X) = s(X) \Rightarrow t(Y) = s(Y))$ gilt

- Hat man eine Menge F von FDs über Attr(R), so schreibt man oft $r \vDash F$, wenn alle FDs aus G gültig sind in r

Zum Beispiel des Relationenformat Mitarbeiter(M#, Abt, Div) sei die Relation

$$r = \{(1, a_1, d_1), (1, a_2, d_1), (2, a_1, d_1), (3, a_2, d_2)\}$$

gegeben. Dann gilt zum Beispiel

$$r \vDash M\# \to Div, oder$$
$$r \vDash \{M, Abt\} \to Div, aber$$
$$r \nvDash Abt \to Div$$

- Löscht man aber aus r ein Tupel, zum Beispiel dasjenige mit M# = 3, dann gilt letztere FD plötzlich: $r\{(3, a_2, d_2)\} \vDash Abt \to Div$

Erzwingen von FDs in der Datenbank

- will man, dass M# ➔Abt, Div und Abt ➔ Div stets gelten, nicht aber Abt ➔ M#, und will man die Gültigkeit der FDs durch UNIQUE Constraints erzwingen, bleibt nichts anderes übrig, als das Format Mitarbeiter(M#, Abt, Div) aufzuspalten in zwei Formate MitAbt(M#, Abt) und AbtDiv(Abt,Div) mit UNIQUE(M#) und UNIQUE(Abt). Man ersetzt also Mitarbeiter(M#, Abt, Div), {M# ➔ Abt, M# ➔ Div, Abt ➔ Div) durch MitAbt(M#, Abt), {M# ➔ Abt) und AbtDiv(Abt,Div), {Abt ➔ Div)

Herleitung einer FD aus einer Menge von FDs

- Im vorherigen Beispiel kommt die FD M# ➔ Abt nicht mehr vor, das macht nichts weil aus M# ➔ Abt und Abt ➔ Div auch M# ➔ Div folgt
  ➔Begriff der Herleitung einer FD nötig, in unserem Falle M# ➔ Div, aus einer Menge von FDs, in unserem Falle {M# ➔ Abt, Abt ➔ Div}
- Gegeben sei eine Menge F von FDs sowie eine weitere FD X ➔ Y

`X ➔ Y ist aus F herleitbar, in Zeichen`
$F \vdash X \to Y$

wenn es eine endliche Folge $(fd_k) 1 \le k \le n$ von FDs gibt, so dass $fd_n = X \to Y$ ist, und für alle k eine der Bedingungen gilt:

1. $fd_k = V \to W, mit\ V \supseteq W \neq \{\ \}(W\ nicht\ leer)$
2. $fd_k = V \cup Z \to W \cup Z, mit\ fd_i = V \to W\ für\ ein\ j < k$
3. $fd_k = V \to Z, mit\ fd_i = V \to W\ und\ fd_j = W \to Z\ für\ ein\ i < k\ und\ ein\ j < k$
4. $fd_k \in F$

Etwas griffiger lauten die Herleitungsregeln:

V → W , mit V ⊇ W ≠ {} folgt stets	(a)
aus V → W folgt V∪Z → W∪Z	(b)
aus V → W und W → Z folgt V → Z	(c)
eine $fd_k \in$ F folgt stets aus F	(d)

Damit soll als Beispiel

{ABC → DE, E → FB, G → C} ⊢ ABG → F

gezeigt werden, mit der Herleitungsfolge

$(fd_k)_{1 \le k \le 9} \equiv$ <G → C,	(1), wegen (d)
ABG → ABC,	(2), wegen (1) und (b)
ABC → DE,	(3), wegen (d)
ABG → DE,	(4), wegen (2),(3) und (c)
E → FB,	(5), wegen (d)
DE → FBD,	(6), wegen (5) und (b)
ABG → FBD,	(7), wegen (4),(6) und (c)
FBD → F,	(8), wegen (a)
ABG → F>	(9), wegen (7),(8) und (c)

Bemerkung: Der Herleitungsbegriff ist **korrekt**, das heisst, es ist nur herleitbar, was auch gültig ist:

Ist F ∪ {X → Y} eine Menge von FDs über Attr(R), so gilt

$$F \vdash X \to Y \quad \Rightarrow \quad \forall r \in Rel(R)\ (r \vDash F \ \Rightarrow\ r \vDash X \to Y)$$

(Beweis trivial, induktiv über die Länge einer Herleitung)

Abschlusssatz für FDs

- Von einer nichtleeren Attributemenge ausgehen und ermitteln, welche Attribute daraus folgen
  ➔ ermittle maximale Attributemenge V^+, für welche gilt $F \vdash V \to V^+$

Gegeben: $F = \{ABC \to DE, E \to HB, G \to C$

Man berechne: $\{ABC\}^+, \{HB\}^+, \{EG\}^+$

$Aus\ \{ABC\}^+\ folgt\ \{ABC\}$
$Aus\ \{ABC\}\ folgt\ \{ABC \to DE\} \to \{ABCDE\}$
$Aus\ \{ABCDE\}\ folgt\ \{E \to HB\} \to \{ABCDEH\}$

$Aus\ \{HB\}^+\ folgt\ \{HB\}$
$Aus\ \{HB\}\ folgt\ nichts\ weiteres \to \{HB\}$

$Aus\ \{EG\}^+\ folgt\ \{EG\}$
$Aus\ \{EG\}\ folgt\ \{E \to HB\} \to \{EGHB\}$
$Aus\ \{EGHB\}\ folgt\ \{G \to C\} \to \{EGHBC\}$

Vollständigkeitssatz für die Herleitung von FDs

Sei F ∪ {X → Y} eine Menge von FDs über Attr(R). Dann gilt

$$\forall r \in Rel(R)\ (r \vDash F \ \Rightarrow\ r \vDash X \to Y) \quad \Rightarrow \quad F \vdash X \to Y$$

(alles was gültig ist, ist herleitbar)

Beweis: Wie beim Beweis des Abschlusssatzes konstruieren wir eine Relation r ∈ Rel(R) bestehend aus zwei Tupeln, t und s, wie folgt:
t(A)=0 für alle A ∈ Attr(R),
s(A)=0 für A ∈ X⁺, und s(A)=1 für A ∈ Attr(R) \ X⁺.
Dieselbe Argumentation zeigt dass r ⊨ F, damit wegen der Voraussetzung r ⊨ X → Y, und deshalb ist Y ⊆ X⁺ , was wiederum wegen dem Abschlusssatz bedeutet F ⊢ X → Y.
Kürzt man die linke Seite

$$\forall r \in Rel(R)\ (r \vDash F \ \Rightarrow\ r \vDash X \to Y) \quad \text{ab als} \quad F \vDash X \to Y,$$

dann bedeutet der Vollständigkeitssatz und die Korrektheit des Herleitungsbegriffes zusammengenommen gerade dass

$$\mathbf{F \vDash X \to Y} \quad \Leftrightarrow \quad \mathbf{F \vdash X \to Y},$$

in Prosa, dass der **semantische Folgerungsbegriff** ⊨ der Gültigkeiten und der **syntaktische Herleitungsbegriff** ⊢ gleichwertig sind. Dies ist keine Selbstverständlichkeit und wird auch nicht mehr der Fall sein, sobald zu den FDs noch Inklusionsabhängigkeiten dazukommen.

Superschlüssel, Schlüssel

- Sei R ein Relationenformat, F eine Menge von FDs über Attr(R) und r ∈ Rel*. Eine nichtleere Teilmenge X ⊆ Attr(R) ist ein **(syntaktischer) Superschlüssel von (R,F)**, falls F ⊢ X ➔ Attr(R) (zeitunabhängig, von Tabelleninhalt unabhängig)
  X ist ein **(semantischer) Superschlüssel von r**, falls r ⊨ X ➔ Attr(R) (zeitabhängig und von Tabelleninhalt abhängig)

- In beiden Fällen ist ein **Schlüssel** ein minimaler Superschlüssel
- Jeder syntaktische Schlüssel ist zwar ein semantischer Superschlüssel, nicht hingegen notwendigerweise auch ein semantischer Schlüssel

Gegeben ist R(A,B,C,D,E) und F={AB➔CDE, CD➔AB, A➔C, AC➔E}

Gesucht sind alle Schlüssel von (R, F)

Wir suchen zuerst Superschlüssel. Es ist $\{A\}^+ = \{ACE\}, \{B\}^+ = \{B\}, \{C\}^+ = \{C\}, \{D\}^+ = \{D\}, \{E\}^+ = \{E\}$

also gibt es keinen einelementigen Superschlüssel. Weiter ist

$\{AB\}^+ = \{ABCDE\}, \{AC\}^+ = \{ACE\}, \{AD\}^+ = \{ADCEB\}$,
$\{AE\}^+ = \{AECE\}, \{BC\}^+ = \{BC\}, \{BD\}^+ = \{BD\}$,
$\{BE\}^+ = \{BE\}, \{CD\}^+ = \{CDABE\}, \{CE\}^+ = \{CE\}$,
$\{DE\}^+ = \{DE\}$

also sind {AB}, {AD} und {CD} zweielementige Superschlüssel. Weiter ist ({ABC} braucht nicht kontrolliert zu werden, da bereits {AB} als Superschlüssel erkannt ist, usw)
$\{ACE\}^+ = \{ACE\}, \{BCE\}^+ = \{BCE\}, \{BDE\}^+ = \{BDE\}$, also gibt es keinen dreielementigen Superschlüssel, der nicht bereits einen zweielementigen enthält.

Jede vierelementige (und die fünfelementige) Menge enthält einen der Superschlüssel {AB}, {AD} oder {CD} als Teilmenge, ist also uninteressant (letztlich suchen wir minimale Superschlüssel).

Weil es aber keinen einelementigen Superschlüssel gibt, sind alle drei der Superschlüssel {AB}, {AD} und {CD} minimal, das heisst, sie sind die gesuchten Schlüssel.

Verlustfreie Zerlegung

Seien R, S und U Relationenformate mit U = R∪S, und u eine Relation zum Format U. Dann gilt in jedem Fall

$$u \subseteq \pi_{R(U)} \bowtie \pi_{S(U)}.$$

(Begründung: $\pi_{R(U)} \bowtie \pi_{S(U)} = \{t \in dom(R \cup S) \mid t(R) \in \pi_{R(U)} \wedge t(S) \in \pi_{S(U)}\} \supseteq \{t \in dom(U) \mid t \in u \wedge t \in u\} = u$, weil für jedes $t \in dom(R \cup S) = dom(U)$ gilt: wenn $t \in u$, dann $t(R) \in \pi_{R(U)}$).

In $u \subseteq \pi_{R(U)} \bowtie \pi_{S(U)}$ kann i. A. "⊆" nicht durch "=" ersetzt werden

Gilt aber $u = \pi_{R(U)} \bowtie \pi_{S(U)}$, so ist $\pi_{R(U)} \bowtie \pi_{S(U)}$ eine **verlustfreie Zerlegung** von u in $\pi_{R(U)}$ und $\pi_{S(U)}$ ("lossless decomposition").

In unserem obigen Beispiel der Zerlegung von

(Mitarbeiter(M#,Abt,Div), {M# ➔ Abt, Abt ➔ Div}) in

(MitAbt(M#,Abt), {M# ➔ Abt}) und (AbtDiv(Abt,Div), {Abt ➔ Div})

gilt für jedes r ∈ Rel(Mitarbeiter) mit r ⊨ {M# ➔ Abt, Abt ➔ Div}, dass $r = \pi_{M\#,Abt}(r) \bowtie \pi_{Abt,Div}(r)$,

deshalb nennt man auch die Zerlegung von

(Mitarbeiter(M#,Abt,Div), {M# ➔ Abt, Abt ➔ Div})

selber eine **verlustfreie Zerlegung**. Jeden Informationsgehalt, den man durch (Mitarbeiter(M#,Abt,Div), {M# ➔ Abt, Abt ➔ Div}) abbilden kann, kann man auch durch (MitAbt(M#,Abt), {M# ➔ Abt}), zusammen mit (AbtDiv(Abt,Div), {Abt ➔ Div}), abbilden (aber nicht umgekehrt, dafür *redundanzfrei* ...).

Boyce-Codd Normalform (BCNF)

- Im Beispiel der Zerlegung von 'Mitarbeiter' in 'MitAbt' und 'AbtDiv' haben wir eine Zerlegung in Formate gemacht, deren FDs jeweils **durch Schlüsselbedingungen erzwungen** werden konnten. Eine solche Zerlegung in Formate in "Boyce-Codd Normalform" ist äusserst praktisch (weil sie durch UNIQUE Conditions im Datenbanksystem implementiert werden kann), aber nicht immer erreichbar
- Das Format (R, F) (das heisst R ist ein Relationenformat und F eine Menge von FDs über Attr(R)) ist in **Boyce-Codd Normalform (BCNF)** falls gilt:
$\forall X, Y \subseteq Attr(R)[(X \to Y) \in F \Rightarrow Y \subseteq X \vee F \vdash X \to Attr(R)]$

➔ **das heisst, jede FD aus F ist entweder trivial oder deren linke Seite ist Superschlüssel.**

Simon Flüeli, IT10b

Welche der folgenden Formate sind in BCNF?

1. (R(A,B,C,D,E,F), {A➔B, AB➔CDEF, C➔D, E➔F})
   ➔ nein, weil C nicht erzeugend ist
2. (R(A,B,C,D), {ABC➔D, CD➔A})
   ➔ nein, linke Seite erzeugt nicht alles
3. (R(A,B,C,D), {A➔B, B➔C, C➔D, D➔A})
   ➔ ja
4. (R(A,B,C,D), {ABC➔D, ABD➔C, ACD➔B, BCD➔A, D➔A, AC➔B, AD➔C})
   ➔ ja

Vereinfachung von Mengen von FDs

- Hat man zum Beispiel $F = \{AB \to CD, B \to C\}$ gegeben, so stellt sich zuerst die Frage der Vereinfachung, da gewisse Redundanz in den angegebenen FDs stecken kann. Wir ersetzen zuerst die Menge der FDs durch eine **äquivalente** Menge (das heisst, dass noch dieselben FDs herleitbar sind nach wie vor), deren FDs aber einfachere Form haben, nämlich **auf der rechten Seite nur Attribute**, keine Mengen von Attributen

$$\{AB \to C, AB \to D, B \to C\}$$

Dann sieht man sofort, dass in AB ➔ C das A weggelassen werden kann, und dass somit die ganze FD weggelassen werden kann, das heisst, aus den restlichen herleitbar ist. Man erhält eine neue Menge G von FDs, welche äquivalent ist zu F

$$G = \{AB \to D, B \to C\}$$

➔ Es wird alles Überflüssige weggestrichen

Minimale Überdeckung

- G heisst eine **minimale Überdeckung** (minimal cover) für F, falls G aus F durch Anwendung des folgenden Algorithmus hervorgeht

1. ersetze alle FDs durch solche der Form X ➔ A **(rechts ein einzelnes Attribut)**
2. für alle X ➔ A und B ∈ X,
   falls $(G \setminus \{X \to A\}) \cup \{X \setminus \{B\} \to A\}$ äquivalent ist zu G,
   dann ersetze X ➔ A durch X\{B} ➔ A
   ➔ dieser Schritt so lange wiederholen, bis alle X minimal sind)
3. für alle X ➔ A
   falls $G \setminus \{X \to A\}$ äquivalent ist zu G, dann streiche X ➔ A aus G

- Es ist wichtig, dass der Algorithmus so angewendet wird wie angegeben

Noch störendes in einer minimalen Überdeckung

Nun gehen wir der Frage nach, wie in einer minimalen Ueberdeckung G die FDs sich gegenseitig quasi noch stören können. Sei

$$X \to A_j \in G \text{ für } 1 \le j \le n.$$

Wir betrachten $X \cup \{A_1, A_2, \dots, A_n\}$ als Grundmenge und ein $V \to A$ darin, das heisst

$$V \to A \in G \quad \text{und} \quad V \cup \{A\} \subseteq X \cup \{A_1, A_2, \dots, A_n\}$$

Behauptung: **Dann muss V=X oder A ∈ X sein.**

Beweis: Sei V ≠ X. Dann ist V ⊂ X nicht möglich wegen Minimalität von X.

Daher ist
$V \cap \{A_1, A_2, \dots, A_n\}$ nicht leer.
Sei oBdA $V \cap \{A_1, A_2, \dots, A_n\} = \{A_1, A_2, \dots, A_m\}$.

Nun ist
$X \to A_j \in G$ für $1 \le j \le m$ und $V \to A \in G$. Wäre $A \in \{A_{m+1}, \dots, A_n\}$, dann wäre $X \to A$ redundant in G. $A \in \{A_1, \dots, A_m\}$ kommt sowieso nicht in Frage wegen der Minimalität von V.

Also bleibt (unter der Annahme V ≠ X) nur A ∈ X als Möglichkeit übrig.

Da X in der Grundmenge $X \cup \{A_1, A_2, \dots, A_n\}$ Schlüssel ist, kommt einem A ∈ X eine Sonderrolle zu (da man eben sieht, **dass solche A's i.A. unvermeidbar sind, das heisst, noch vorkommen können**).

Aus diesem Grunde ist die Definition der dritten Normalform so komisch:

Dritte Normalform (3NF)

Primattribut

- Sei R ein Relationenformat, F eine Menge von FDs über Attr(R) und $A \in Attr(R)$. **A** ist ein (syntaktisches) **Primattribut von (R, F)**, falls es einen Schlüssel von (R, F) gibt, welcher A enthält

- In der Grundmenge $X \cup \{A_1, A_2, \dots, A_n\}$ ist also für ein darin enthaltenes V ➔ A entweder V ein Superschlüssel oder A ein Primattribut. Diese Situation kann man ausgehend von einem Format (R, F) **immer erreichen**

- Das Format (R, F) (das heisst R ist ein Relationenformat und F eine Menge von FDs über Attr(R)) ist in dritter **Normalform (3NF)** falls gilt

$$\forall X, Y \subseteq Attr(R)[(X \to Y) \in F \Rightarrow \forall A \in Y \setminus X\ (A\ prim) \vee F \vdash X \to Attr(R)],$$

das heisst, alle FDs aus F sind entweder trivial oder zeigen auf Primattribute oder deren linke Seite ist Superschlüssel

- Dieselbe Bemerkung wie bei der Definition von BCNF ist hier ebenfalls am Platze: Man findet auch oft die Definition von 3NF mit $F \vdash X \to Y$ anstelle von $(X \to Y) \in F$ auf der linken Seite der Implikation. Das kommt auf das gleiche heraus, aber die Variante mit $(X \to Y) \in F$ ist viel praktischer in der Kontrolle, ob ein (R, F) in 3NR ist oder nicht

➔ Man beachte, dass aus $Y \subseteq X$ die Eigenschaft $\forall A \in Y \setminus X\ (A\ prim)$ folgt, was heisst, dass **aus BCNF die 3NF folgt**

Codd's Definition der dritten Normalform

- Dritte Normalform definiert in Begriffen der **transitiven Abhängigkeiten**
- Gegeben: Format (R, F) mit einer Menge von funktionalen Abhängigkeiten F. Sei A ein Attribut von R und X eine nichtleere Menge von Attributen von R. Dann ist **A von X transitiv abhängig** bezüglich (R, F), falls es eine nichtleere Menge Y von Attributen von R gibt, so dass $F \vdash X \to Y$ und $F \vdash Y \to A$, aber $F \nvdash Y \to X$ gilt, sowie $A \notin X \cup Y$

- (R, F) ist in 3NF, falls kein nichtprimes Attribut transitiv von einem Schlüssel abhängt

- Etwas formaler ist 3NF, falls
  ∀ A ∈ Attr(R) [¬(A prim)
  ⇒ $\neg\exists Y, K \subseteq Attr(R)$ (K Schlüssel ∧ F ⊢ K → Y ∧ F ⊢ Y → A ∧ F ⊬ Y → K ∧ A ∉ K ∪ Y)].

Äquivalenz der Definitionen (Einschub)

Man muss nach dem Umweg deshalb die Definition zuerst vereinfachen zu

∀ A ∈ Attr(R) [¬(A prim)
⇒ $\neg\exists Y, K \subseteq Attr(R)$ (K Schlüssel ∧ F ⊢ Y → A ∧ F ⊬ Y → K ∧ A ∉ Y)].

Die Aussage "∃K ⊆ Attr(R) (K ist Schlüssel ∧ F ⊬ Y → K)" ist äquivalent zur Aussage
"Y ist kein Superschlüssel", deshalb kann man die Definition noch etwas vereinfachen zu
∀ A ∈ Attr(R) [¬(A prim)
⇒ $\neg\exists Y \subseteq Attr(R)$ (¬(Y Superschlüssel) ∧ F ⊢ Y → A ∧ A ∉ Y)].

Etwas umgeformt ergibt sich äquivalent dazu

∀ A ∈ Attr(R), Y ⊆ Attr(R) [F ⊢ Y → A
⇒ A ∈ Y ∨ (A prim) ∨ F ⊢ Y → Attr(R)].

Damit ist man schon fast bei der sehr einfachen Definition, die oben gegeben ist, nämlich

∀ X,Y ⊆ Attr(R) [(X → Y) ∈ F ⇒ ∀A∈Y\X (A prim) ∨ F ⊢ X → Attr(R)].
∀ X,Y ⊆ Attr(R) [(X → Y) ∈ F ⇒ ∀A∈Y\X (A prim) ∨ F ⊢ X → Attr(R)].

Der Unterschied besteht noch darin, dass man bei letzterer nur etwas zeigen muss für alle (X → Y) ∈ F, hingegen bei der (vereinfachten) Umwegdefinition etwas für alle F ⊢ Y → A.

Die muss man aber zuerst haben, die sind gar nicht gegeben, die muss man zuerst herleiten!

Also, man vergesse die 3NF Definition mittels transitiver Abhängigkeiten.

Bemerkung: tut Codd keinen Abbruch

Welche der folgenden Formate sind in 3NF?
(R(A,B,C), {AB ➔C, C ➔ A})
(R(A,B,C), {A ➔B, B ➔C})
(R(A,B,C), {AB ➔C, C ➔B})
(R(A,B,C,D), {AB ➔C, B ➔D})
(R(A,B,C,D,E), {AB ➔CD, C ➔E, D ➔A})
(R(A,B,C,D,E), {AB ➔CD, E ➔AB, D ➔A})
(R(A,B,C,D), {A ➔B})
(R(A,B,C,D), {A ➔CD, D ➔B})
(R(A,B,C,D,E,F,G,H),
{A ➔BC, BC ➔DG, G ➔H, D ➔EF, EF ➔A, H ➔D})
(R(A,B,C,D,E), {A ➔E, B ➔E, C ➔E, D ➔E})

➔ (R(A,B,C), {AB ➔C, C ➔A})
(R(A,B,C), {AB ➔C, C ➔B})
(R(A,B,C,D,E,F,G,H),
{A ➔BC, BC ➔DG, G ➔H, D ➔EF, EF ➔A, H ➔D})

Zerlegungssatz (Satz über die Erreichbarkeit von 3NF)

- 3NF ist so definiert, **dass man es stets erreichen kann** (verlustfrei und FD-erhaltend). Das ist der Hauptsatz der Normalisierungstheorie.
- Sei (R,F) ein Format, das heisst R ein Relationenformat und F eine Menge von funktionalen Abhängigkeiten über Attr(R). Dann gibt es eine Zerlegung in Formate

(R_1,F_1) , (R_2,F_2) , ..., (R_m,F_m), sodass mit U_i=Attr(R_i) gilt

$U_1 \cup U_2 \cup \ldots \cup U_m$=Attr(R) ,

$\forall i,j\ (U_i \subseteq U_j \ \Rightarrow\ U_i = U_j)$,

$\forall k\ (F_k \subseteq \{X \to Y \mid (F \vdash X \to Y) \wedge (X \cup Y \subseteq U_k)\})$,

$F_1 \cup F_2 \cup \ldots \cup F_m$ ist äquivalent zu F (**"FD erhaltend"**) ,

$\forall r \in Rel(R)\ (r \models F \ \Rightarrow\ r = \pi_1(r) \bowtie \pi_2(r) \bowtie \ldots \bowtie \pi_m(r))$,
mit π_j =Projektion auf U_j (**"verlustfrei"**),

und für alle k ist (R_k,F_k) in 3NF.

Beweis und Konstruktion für den Zerlegungssatz

- Ausgehend von einer minimalen Ueberdeckung G für F (G ist dann äquivalent zu F) konstruieren wir der Reihe nach (wir identifizieren Attributsmengen der einfacheren Notation halber direkt mit Relationenformaten):

$GL := \{X \mid (X \to A) \in G\}$

$GV := \{V \mid \exists X \in GL\ (V = X \cup \{A \mid (X \to A) \in G\})\}$

$GW := \{W \in GV \mid \neg \exists V \in GV\ (W \subseteq V \wedge W \neq V)\}$

$Z(G) := \{(W,H) \mid W \in GW \wedge H = \{(X \to A) \in G \mid X \cup \{A\} \subseteq W\}\}$

und schliesslich (Λ ist die leere Menge)

$ZK(G) := Z(G)$ falls $\exists W \in GW$ (W superkey für U bezüglich G), und
$:= Z(G) \cup \{(K,\Lambda)\}$ sonst, für einen Schlüssel K für U (bezgl G)

(die Tatsache, dass nach Konstruktion mindestens ein Glied einen Superschlüssel von (R,F) enthält, sorgt für die erste Bedingung, dass kein Attribut verloren geht sowie für die zweitletzte, dass die Zerlegung verlustfrei ist). Man beachte, dass das Ergebnis dieser Zerlegungskonstruktion nicht eindeutig ist

Gegeben (R(A,B,C,D), {A → B, AB → CD, C → D}).

Zuerst konstruieren wir eine minimale Ueberdeckung (schrittweise):

F = {A → B, AB → C, AB → D, C → D} (rechte Seiten einattributig)

{A → B, A → C, A → D, C → D} (linke Seiten minimieren)

G = {A → B, A → C, C → D} (überflüssige FDs weg)

G ist dann eine minimale Ueberdeckung von F

Dann folgt die Zerlegungskonstruktion:

GL = {{A}, {C}} (die linken Seiten)

GV = {{A,B,C}, {C,D}} (mit den Abschlüssen)

GW = GV (die maximalen Mengen in GV)

Z(G) = {(ABC, A → BC), (CD, C → D)} in etwas nachlässiger Schreibweise

ZK(G) = Z(G) , da ABC Superschlüssel in G

Mit anderen Worten ist

$(R_1(A,B,C), \{A \to BC\})$, $(R_2(C,D), \{C \to D\})$

eine Zerlegung des gegebenen Formates in 3NF Formate mit allen Eigenschaften des Zerlegungssatzes.

$R(A,B,C,D,E,F);\ \{D \to EF, F \to AB, B \to CD\}$

1. BCNF (weil D, F, B Superschlüssel sind)

$R(A,B,C,D,E,F);\ \{ABC \to DEF, B \to EF\}$

1. Kein BCNF (B ist kein Superschlüssel)
   Kein 3NF (B ist kein Superschlüssel und EF nicht prim)
2. **Rechte Seiten einattributig**
   $F = \{ABC \to D, ABC \to E, ABC \to F, B \to E, B \to F\}$
3. **Linke Seiten minimieren**
   nicht möglich
4. **Überflüssige FDs weg**
   $G = \{ABC \to D, B \to E, B \to F\}$

Zerlegungskonstruktion

5. **Linke Seiten**
   $GL = \{\{ABC\}, \{B\}\}$
6. **Abschlüsse**
   $GV = \{\{A,B,C,D\}, \{B,E,F\}\}$
7. **Maximale Mengen in GV**
   $GW = GV$

$Z(G) = \{\{ABC \to D\}, \{B \to EF\}\}$

$ZK(G) = Z(G)$ (da ABC Superschlüssel ist)

$\Rightarrow R_1(A,B,C,D), \{ABC \to D\}$
$\Rightarrow R_2(B,E,F), \{B \to EF\}$

$R(A,B,C,D,E,F);\ \{BC \to D, E \to D\}$

Superschlüssel: ABCEF

1. Kein BCNF, Kein 3NF
2. **Rechte Seite einattributig** → schon erledigt
3. **Linke Seite minimieren** → nicht möglich
4. **Überflüssige FDs weg** → nicht möglich
   $G = F \leftarrow minimale\ Überdeckung$

Zerlegungskonstruktion

5. **Linke Seiten**
   $GL = \{\{BC\}, \{E\}\}$
6. **Abschlüsse**
   $GV = \{\{B,C,D\}, \{E,D\}\}$
7. **Maximale Mengen in GV**
   $GW = GV$

$Z(G) = \{(BCD, BC \to D), (ED, E \to D)\}$
$ZK(G) = Z(G) \cup \{\{ABCEF, \{\}\}$
$ZK(G) = \{(BCD, BC \to D), (ED, E \to D), (ABCEF, \{\}\}$

$\Rightarrow R_1(B,C,D); \{BC \to D\}$
$\Rightarrow R_2(E,D); \{E \to D\}$
$\Rightarrow R_3(A,B,C,E,F); \{\ \ \}$

$R(A,B,C,D,E,F), \{C \to D, D \to E, E \to F, F \to C\}$

Superschlüssel: $\{ABC\}, \{ABD\}, \{ABE\}, \{ABF\}$

→ 3NF, aber nicht BCNF

$R(A,B,C,D,E,F);\ \{ABCD \to EF, B \to C, E \to F\}$

Superschlüssel: $\{ABD\}$

1. Kein BCNF ($\{B\}$ und $\{E\}$ sind keine Superschlüssel)
   Kein 3NF, weil $\{B\}$ und $\{E\}$ nicht Superschlüssel und $\{F\}$ nicht prim
2. **Rechte Seiten einattributig**
   $F = \{ABCD \to E, ABCD \to F, B \to C, E \to F\}$
3. **Linke Seite minimieren**
   $\{\{ABD \to E\},\ \{ABD \to F\}, \{B \to C\}, \{E \to F\}\}$
4. **Überflüssige FDs weg**
   $G = \{\{ABD \to E\}, \{B \to C\}, \{E \to F\}\}$ →minimale Überdeckung

Zerlegungskonstruktion

5. **Linke Seiten**
   $GL = \{\{ABD\}, \{B\}, \{E\}\}$
6. **Abschlüsse**
   $GV = \{\{A,B,D,E\}, \{B,C\}, \{E,F\}\}$
7. **Maximale Mengen in GV**
   $GW = GV$

$Z(G) = \{(ABDE, ABD \to E), (BC, B \to C), (EF, E \to F)\}$
$ZK(G) = Z(G)$ (weil Superschlüssel ABD ist in ABDE enthalten)

$\Rightarrow R_1(ABDE), \{ABD \to E\}$
$\Rightarrow R_2(BC), \{B \to C\}$
$\Rightarrow R_3(EF), \{E \to F\}$

$R(A,B,C,D,E)\{AB \to CD, A \to B, C \to D\}$

Superschlüssel: $\{AE\}$

1. Kein BCNF, Kein 3NF
2. **Rechte Seiten einattributig**
   $F = \{AB \to C, AB \to D, A \to B, C \to D\}$
3. **Linke Seite minimieren**
   $\{\{A \to C\}, \{A \to D\}, \{A \to B\}, \{C \to D\}\}$
4. **Überflüssige FDs weg**
   $G = \{\{A \to C\}, \{A \to B\}, \{C \to D\}\}$

Zerlegungskonstruktion

5. **Linke Seiten**
   $GL = \{\{A\}, \{C\}\}$
6. **Abschlüsse**
   $GV = \{\{ABC\}, \{CD\}\}$
7. **Maximale Mengen in GV**
   $GW = GV$

$Z(G) = \{(ABC, A \to BC), (CD, C \to D)\}$
$ZK(G) = \{(ABC, A \to BC), (CD, C \to D), (AE, \{\ \ \})\}$

$\Rightarrow R_1(ABC), \{A \to BC\}$
$\Rightarrow R_2(CD), \{C \to D\}$
$\Rightarrow R_3(AE), \{\ \ \}$

$R(A,B,C,D,E);\ \{A \to CD, CD \to E, E \to A\}$

Superschlüssel: $\{AB\}, \{EB\}, \{CDB\}$

1. Kein BCNF, aber 3NF (weil rechts alles Primattribute)

$R(A,B,C,D,E);\ \{A \to BCDE, BC \to DE, D \to E\}$

Superschlüssel: $\{A\}$

1. Kein BCNF, Kein 3NF
2. **Rechte Seite einattributig**
   $F = \{A \to B, A \to C, A \to D, A \to E, BC \to D, BC \to E, D \to E\}$
3. **Linke Seite minimieren** → nicht möglich
4. **Überflüssige FDs weg**
   $G = \{A \to B, A \to C,\ BC \to D, D \to E\}$
5. **Linke Seiten**
   $GL = \{\{A\}, \{BC\}, \{D\}\}$
6. **Abschlüsse**
   $GV = \{\{A,B,C\}, \{B,C,D\}, \{D,E\}\}$
7. **Maximale Mengen in GV**
   $GW = GV$

$Z(G) = \{(ABC, A \to BC), (BCD, BC \to D), (DE, D \to E)\}$
$ZK(G) = Z(G)$ (weil Superschlüssel A in ABC enthalten)

$\Rightarrow R_1(ABC), \{A \to BC\}$
$\Rightarrow R_2(BCD), \{BC \to D\}$
$\Rightarrow R_3(DE), \{D \to E\}$

$R(A,B,C,D,E);\ \{ABCD \to E, E \to A\}$

Superschlüssel: $\{ABCD\}$

1. Kein BCNF, aber 3NF

$R(A,B,C,D,E);\ \{ABC \to D, AB \to E\}$

Superschlüssel: $\{ABC\}$

1. Kein BCNF, Kein 3NF
2. **Rechte Seiten einattributig**
   $F = \{ABC \to D, AB \to E\}$
3. **Linke Seite minimieren**
   nicht möglich
4. **Überflüssige FDs weg**
   keine

Zerlegungskonstruktion

5. **Linke Seiten**
   $GL = \{\{ABC\}, \{AB\}\}$
6. **Abschlüsse**
   $GV = \{\{A,B,C,D\}, \{A,B,E\}\}$
7. **Maximale Mengen in GV**
   $GW = GV$

$Z(G) = \{(ABCD, ABC \to D), (ABE, AB \to E)\}$
$ZK(G) = Z(G)$ (weil Superschlüssel {ABC} in {ABCD} vorhanden

$\Rightarrow R_1(ABCD), \{ABC \to D\}$
$\Rightarrow R_2(ABE), \{AB \to E\}$

$R(A,B,C,D,E);\ \{\ \ \}$

→ BCNF, also auch 3NF

$R(A,B,C,D,E);\ \{A \to B\}$

Superschlüssel: $\{ACDE\}$

1. Kein BCNF, Kein 3NF
2. Rechte Seite einattributig → ist schon so
3. Linke Seite minimieren → nicht möglich
4. Überflüssige FDs weg → nicht möglich

Zerlegungskonstruktion

5. Linke Seiten
   $GL = \{\{A\}\}$
6. Abschlüsse
   $GV = \{\{A,B\}\}$
7. Maximale Mengen in GV
   $GW = GV$

$Z(G) = \{(AB, A \to B)\}$
$ZK(G) = \{(AB, A \to B), (ACDE, \{\ \ \})\}$

$\Rightarrow R_1(AB), \{A \to B\}$
$\Rightarrow R_2(ACDE), \{\ \ \}$

$R(A,B,C,D,E);\ \{A \to B, B \to A\}$

Superschlüssel: $\{ACDE\}, \{BCDE\}$

1. Kein BCNF aber 3NF

Abschlüsse

Gegeben: $FuncDep = \{BF \to ABC, D \to E, BF \to AF, A \to B, C \to AB\}$

Gesucht:

$\{AB\}^+$

$Aus\ \{AB\}^+\ folgt\ \{AB\} \to \{AB\}$

$\{ADF\}^+$
$Aus\ \{ADF\}^+\ folgt\ \{ADF\} \to \{ADF\}$
$Aus\ \{ADF\}\ folgt\ \{A \to B\} \to \{ADFB\}$
$Aus\ \{ABDF\}\ folgt\ \{D \to E\} \to \{ADFBE\}$
$Aus\ \{ADFBE\}\ folgt\ \{BF \to ABC\} \to \{ADFBEC\}$

$\{ABC\}^+$
$Aus\ \{ABC\}^+\ folgt\ \{ABC\} \to \{ABC\}$

$\{CF\}^+$
$Aus\ \{CF\}^+\ folgt\ \{CF\} \to \{CF\}$
$Aus\ \{CF\}\ folgt\ \{C \to AB\} \to \{CFAB\}$

Gegeben: $R(A,B,C,D,E,F,G,H,K)$,
$\{BF \to ABC, D \to E, BF \to AF, A \to B, C \to AB\}$

Gesucht:

$\{ABK\}^+$
$Aus\ \{ABK\}^+\ folgt\ \{ABK\} \to \{ABK\}$

$\{ABCG\}^+$
$Aus\ \{ABCG\}^+\ folgt\ \{ABCG\} \to \{ABCG\}$

$\{ADFHK\}^+$
$Aus\ \{ADFHK\}^+\ folgt\ \{ADFHK\} \to \{ADFHK\}$
$Aus\ \{ADFHK\}\ folgt\ \{D \to E\} \to \{ADFHKE\}$
$Aus\ \{ADFHKE\}\ folgt\ \{A \to B\} \to \{ADFHKEB\}$
$Aus\ \{ADFHKEB\}\ folgt\ \{BF \to ABC\} \to \{ADFHKEBC\}$

$\{CFGHK\}^+$
Aus {CFGHK}$^+$ folgt {CFGHK} → {CFGHK}
Aus {CFGHK} folgt $\{C \to AB\} \to \{CFGHKAB\}$

(Syntaktische Schlüssel)

Gegeben: Relationenformat mit funktionalen Abhängigkeiten
Gesucht: alle syntaktischen Schlüssel, bezogen auf die Abhängigkeiten

a) $R(A,B,C,D,E);\{ABC \to DE, D \to E\}$
   Schlüssel: $\{A,B,C\}$

b) $R(A,B,C,D,E,F,G);\{ABC \to DE, D \to E\}$
   Schlüssel: $\{A,B,C,F,G\}$

c) $R(A,B,C,D,E);\{A \to BC, C \to D, D \to AE\}$
   Schlüssel: $\{A\}, \{C\}, \{D\}$

Simon Flüeli, IT10b

d) $R(A,B,C,D,E,F,G,H);\{A \to BC, C \to D, D \to AE\}$
   Schlüssel: $\{A,F,G,H\}, \{C,F,G,H\}, \{D,F,G,H\}$

e) $R(A,B,C,D);\{AB \to C, C \to B\}$
   Schlüssel: $\{A,B,D\}, \{A,C,D\}$

f) $R(A,B,C,D,E);\{BD \to AC, C \to DE\}$
   Schlüssel: $\{B,D\}, \{B,C\}$

g) $R(A,B,C,D);\{A \to B, B \to C, C \to A\}$
   Schlüssel: $\{A,D\}, \{B,D\}, \{C,D\}$

h) $R(A,B,C,D,E,F);\{A \to B, B \to C, C \to A\}$
   Schlüssel: $\{A,D,E,F\}, \{B,D,E,F\}, \{C,D,E,F\}$

(Semantische Schlüssel)

Gegeben ist die Relation r (A B C D E)

A	B	C	D	E
0	1	0	0	0
0	1	2	0	1
0	2	0	3	2
0	1	3	0	3

a) Gesucht sind alle im Moment in r gültigen semantischen Schlüssel: $\{E\}, \{B,C\}, \{C,D\}$

b) Wie viele semantische Schlüssel kann r (mit anderen Tupeln) maximal haben? Beispiel? 10 → im folgenden Beispiel sind genau alle zweielementigen Attributmengen Schlüssel:

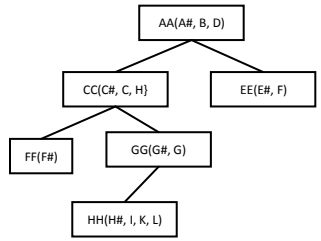
A	B	C	D	E
0	0	1	2	3
4	0	0	5	6
7	8	0	0	9
10	11	12	0	0
0	13	14	15	0

c) Welches wären die semantischen Schlüssel, wenn r nur ein Tupel hätte?
→ die einelementigen Attributmengen
→ $\{A\}, \{B\}, \{C\}, \{D\}, \{E\}$

Erste Normalform Beispiele

Führe die folgende hierarchische Struktur in die erste Normalform

$AA(A\#, B, CC, D, EE)$
$CC(C\#, C, FF, GG, H)$
$EE(E\#, F)$
$FF(F\#)$
$GG(G\#, G, HH)$
$HH(H\#, I, K, L)$



$AA(A\#, A, N, BB, CC, DD)$
$KK(K\#, O, LL, MM)$
$BB(B\#, K, EE)$
$CC(C\#, FF, B, GG)$
$DD(D\#, C, D)$
$LL(L\#, F)$
$MM(M\#, G, H)$
$EE(E\#, I, K)$
$FF(F\#)$
$GG(G\#, HH, L, II)$
$HH(H\#, M)$
$II(I\#, N, P)$

DBF - Concurrency

- Theorie der Gleichzeitigkeit

Einführungsbeispiel

- Herr und Frau Meier in verschiedenen Filialen derselben Bank
- Der Wunsch von Frau Meier führt zur Transaktion T1
R1: Lesen Konto Meier
W1: Schreiben Konto Meier mit Saldo := Saldo + 100
- Der Wunsch von Herr Meier führt zur Transaktion T2
R2: Lesen Konto Meier
W2: Schreiben Konto Meier mit Saldo := Saldo - 100
→ in abstrakter Darstellung:

```
T1begin, R1(x), W1(x), T1end
T2begin, R2(x), W2(x), T2end
```

R1(x): Transaktion T1 **liest** Objekt x
W2(x): Transaktion T2 **schreibt** Objekt x etc.

Konflikte bei gleichzeitigem (concurrent) Ablauf

- laufen die Transaktionen

```
T1begin, R1(x), W1(x), T1end, T2begin,
   R2(x), W2(x), T2end,
oder T2begin, R2(x), W2(x), T2end, T1begin,
   R1(x), W1(x),T1end
```

hintereinander ab, **serialisiert**,

```
T1begin, R1(x), W1(x), T1end, T2begin,
   R2(x), W2(x), T2end,
oder T2begin, R2(x), W2(x), T2end, T1begin,
   R1(x), W1(x), T1end
```

ergeben sich keine Probleme, am Schluss der beiden enthält das Meier Konto wieder gleich viel wie vorher
- laufen die Transaktionen aber "gleichzeitig" ab, zum Beispiel

```
T2begin, R2(x), T1beginn, R1(x), W2(x),
   T2end, W1(x), T1end
```

dann hat am Schluss das Meier Konto 100 zu wenig, weil die Einzahlung von Frau Meier verloren gegangen ist ("**Lost Update**")

Kategorisierung von Konfliktsituationen

```
...R1(x)...R2(x)...T1end...  no problem
...R1(x)...W2(x)...T1end...  non repeatable
read
...W1(x)...R2(x)...T1end...  dirty read
...W1(x)...W2(x)...T1end...  dirty write
```

das Ende einer Transaktion, T1end, kann ein **Commit** sein, oder ein **Abort**, gefolgt von **Rollback**
- im Falle des Rollback muss das System alle Schreiboperationen (insert, update, delete) rückgängig machen, deshalb redet der Standard gar nicht vom Fall

```
...W1(x)...W2(x)...T1end...  dirty write
```

(würde man dirty write zulassen, müsste im Falle von T1Abort auch W2(x) rückgängig gemacht werden, **cascading aborts** nennt man das, und jedes System vermeidet das von vorneherein)

Dirty Read kann man haben

- den Fall

```
...W1(x)...R2(x)...T1end...  dirty read
```

kann man im Standard haben mit

SET TRANSACTION ... ISOLATION LEVEL READ UNCOMMITTED

→ Transaktion muss dann read-only sein
→ Read Uncommitted ist nicht in allen Systemen möglich
→ Falls möglich, dann zu.U. auch auf Einzelabfrage Ebene

```
SELECT ... FROM ... WHERE ... WITH UR
```

- Dirty Read mag bei statistischen Abfragen Sinn machen, in denen es nicht auf 100% Genauigkeit ankommt

Non Repeatable Read

- Der interessanteste Fall ist

```
...R1(x)...W2(x)...T1end...  non repeatable
read
```

Beispiele

Transaktion T1

```
SELECT Lohn FROM Pers WHERE empNo = 17
```

Transaktion T2

```
UPDATE Pers SET Lohn = Lohn * 1.04 WHERE
empNo = 17
```

Transaktion T3

```
SELECT Lohn + Bonus ... WHERE empNo = 17
```

→ Transaktion T1 kann also das Lesen des Lohnes nicht wiederholen, das heisst, kriegt nicht mehr dasselbe Resultat wie beim ersten Lesen

Das Phantom Problem

Transaktion T1

```
SELECT SUM(Lohn) FROM Pers
WHERE komplizierteBedingung
```

Transaktion T2

```
INSERT INTO Pers (...empNo...) VALUES
(...18...)
```

Transaktion T3

```
SELECT SUM(Lohn+Bonus) FROM Pers
WHERE komplizierteBedingung
```

wobei empNo = 18 die komplizierte Bedingung erfüllt

→ Dieser **Phantom Effekt** kann auftauchen, selbst wenn repeatable read für Einzelobjekte gewährleistet ist

Phantom ist non repeatable read für Prädikate P statt für Objekte x:
...R1(P)...W2(x in P)...T1end... **Phantom**

Isolation Levels im SQL Standard

- Standard verhindert mit

```
SET TRANSACTION ... ISOLATION LEVEL <isolati-
on level>
```

die dargestellen Phänomene gemäss:

	dirty read	non repeatable read	Phantom
SERIALIZABLE	X	X	X
REPEATABLE READ	X	X	-
READ COMMITTED	X	-	-
READ UNCOMMITTED	-	-	-

- Standard ist dem **Locking** (Sperrungsverfahren) nachempfunden

Locks

- Idee: zum Beispiel vor W(x) einen Lock, **X-Lock**(x) (**Write-Lock**) zu setzen
- Beispiel:

```
...W1(x)...W2(x)...T1end...  dirty
write
```

→ Um dirty write zu vermeiden, setzen die Systeme vor dem Schreiben einen X-Lock, einen **Exclusive Lock**, der bis zum (kurz nach) Commit gehalten wird

- Besitzt eine Transaktion einen Exclusive Lock auf einem Objekt, dann kriegt keine andere Transaktion Zugriff auf das Objekt
- Damit das System Locks in eine interne Datenstruktur unterbruchsfrei schreiben kann (nur das Schreiben selber), braucht das System auf einer tieferen Ebene weitere Parallelisierungshilfen, **Latches**

Latches (Semaphores)

- Wenn ein Prozess mitten in der Berechnung des Zinses für das Konto der Meiers unterbrochen wird, macht das nichts, er kann Bruchteile von Sekunden später, wenn er wieder CPU Ressourcen kriegt, an genau derselben Stelle weiterfahren, an der er unterbrochen wurde
- Gewisse heikle System-Verwaltungsprozesse, wie eben das Schreiben von Lock Eintragungen (und viele andere), dürfen aber nicht unterbrochen werden, da sonst leicht Inkonsistenzen entstehen

- Deshalb wurden **Semaphore** entwickelt
→ einfache gekapselte Datenstrukturen, die z.B. nur 0, 1 oder 0, 1, 2, 3, ... enthalten können und als Schnittstelle für Prozesse Lese- und Schreiboperationen zur Verfügung stellen, welche **nicht unterbrochen werden können** (z.B. "lies n und schreibe anschliessend n+1")

- Semaphore kann man nur implementieren, wenn schon die CPU eine nichtunterbrechbare Anweisung "**test and set**" / "**compare and swap**" besitzt, welche garantiert ohne Unterbrechung ein Flag lesen und je nach Inhalt ein eventuell anderes so oder so schreiben kann

- In der Datenbankwelt nennt man Semaphore oft **Latches**, und der Lockmanager (Buffermanager), Logmanager (Recovery) etc., haben Unmengen davon

Read-Lock = Share-Lock
Write-Lock = Exclusive-Lock

- Wir haben gesehen, dass das System zur Vermeidung von

```
...W1(x)...W2(x)...T1end...  dirty
write
```

einen Exclusive-Lock (X-Lock) setzt (die Transaktion T1 in diesem Fall, da sie zuerst auf das Objekt x greifen möchte)
- Der Fall

```
...R1(x)...W2(x)...T1end...  non
repeatable
read
```

kann dadurch verhindert werden, dass die Transaktion T1 vor R1(x) einen S-Lock, einen **Share-Lock** setzt (Read-Lock), und ihn bis (kurz nach) Transaktionsende behält (im Falle von Rollback bis nach Ende des Rollbacks)
→ Dies entspricht Isolation Level Repeatable Read

- Weil Share-Lock und Exclusive-Lock (verschiedener Transaktionen) miteinander unverträglich sind, wird dann die Transaktion T2 den Write-Lock, den sie vor W2(x) haben müsste, nicht erhalten (T2 wird warten müssen, und wenn das zu lange dauert, wird sie vom System abgeschossen werden

- Bei Isolation Level **Read Committed** würden zwar Read-Locks gesetzt, aber wieder losgelassen, wenn der Lesevorgang fertig ist, auch wenn die Transaktion als ganze noch nicht zu Ende ist

- Bei Isolation Level **Read Uncommitted** werdenkeine Locks gesetzt (Latches schon, eine Write Operation einer anderen Transaktion auf eine Page kann nicht unterbrochen werden)

Update-Locks

Lock Promotion

- Gewisse Systeme kennen U-Locks (Update-Locks) um den Grad der Gleichzeitigkeit zu erhöhen
- Die Idee des U-Lock(x) ist, dass andere Transaktionen das Objekt x zwar noch lesen dürfen, aber nicht mehr schreiben, weil die das U-Lock verlangende Transaktion Interesse daran zeigt, das Objekt x eventuell später auch zu ändern (wenn das dann der Fall ist, muss die Transaktion das U-Lock zu einem X-Lock erweitern → **Lock Promotion**

- Der typische Fall für diese Situation ist

```
CURSOR...SELECT...FOR UPDATE OF...
```

(wenn System U-Locks nicht kennt, werden direkt Write-Locks gesetzt)

Lock Compatibility Matrix

- Für Page und Row Locks haben wir mittlerweile drei Lock-Mode (Share-Lock, Update-Lock, Exclusive-Lock)
- Wenn von zwei verschiedenen Transaktionen die eine einen S-Lock(x) hat, so ist das verträglich damit, dass die andere gleichzeitig einen U-Lock(x) auf demselben Objekt x hat (aber unverträglich, wenn die zweite einen X-Lock(x) hätte)

Page und Row Locks Compatibility Matrix

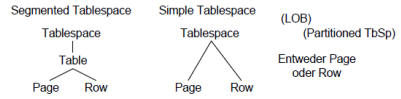
	S	U	X
S	y	y	n
U	y	n	n
X	n	n	n

- Page und Row Locks der Stärke nach geordnet
S → U → X

wobei ein Lockmodus B stärker ist als ein Lockmodus A, A ≤ B, wenn für alle Lockmodi C gilt:
wenn compatible(C,B), dann compatible(C,A)

Locking Granularities

- Die Art und Grösse der Objekte, die gesperrt werden können, kann innerhalb einer Transaktion variieren, zum Beispiel **Locksizes** oder Locking Granularities



- Locksize Row verbessert gegenüber Locksize Page oder gar Locksize Table den Concurrency Level, aber gleichzeitig erhöht es den Systemaufwand während Runtime

- In einem gewissen System kann man durch

```
CREATE TABLESPACE...LOCKSIZE table
```

(oder Locksize tablespace etc.) die Locking Granularity erzwingen für alle Transaktionen bezüglich dieses Tablespace

- Wenn eine Transaktion auf einem Table eine gewisse Anzahl Low Granularity Locks desselben Modus überschreiten will (Systemparameter), versucht die Transaktion, die nächsthöhere Granularity zu sperren im selben Modus, und dafür alle entsprechenden Locks auf kleineren Grössen aufzugeben (**Lock Escalation**)

Zusammenspiel der Locking Granularities

Intent Share Lock (IS-Lock)
Intent Exclusive Lock (IX-LOCK)
Share Intent Exclusive (SIX-Lock)

- Besteht ein S-Lock auf einer Tabelle, dann bedeutet das implizit, dass auch ein S-Lock besteht auf allen Pages und allen Rows der Tabelle
→ gleiches gilt für U-Locks und X-Locks

- Umgekehrt: Will eine Transaktion einen S-Lock auf einer Row oder Page, so muss sie zuerst einen **IS-LOCK (Intent Share Lock)** auf der zugehörigen Tabelle erwerben, und will sie einen U-Lock oder X-Lock auf der Page oder Row, muss sie zuerst einen **IX-Lock (Intent Exclusive Lock)** auf der Tabelle haben
- Auf Table und Tablespace Ebene gibt es daher neben S, U, X-Locks auch noch IS- und IX-Locks

- Hat eine Transaktion also einen S-Lock auf einer Row, darf keine andere Transaktion einen X-Lock auf der Tabelle haben.
→ Deshalb müssen IS-Lock und X-Lock **unverträglich** sein
→ Dasselbe gilt für IX-Lock und S-Lock, sowie für IX-Lock und U-Lock

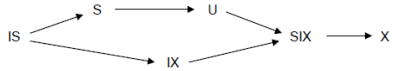
- Des Weiteren möchte man, dass es zu je zwei Locks einen weiteren Lock gibt, der genau gleich stark ist wie die beiden gegebenen zusammen, wegen der Lock Promotion. Dies für zu einem weiteren Lockmodus auf Tabelle und Tablespace**, SIX (Share Intent Exclusive)**, der genau gleich stark ist wie S und IX zusammen

Lock Compatibility für Table(space) Locks

- Table und Tablespace Locks Compatibility Matrix

	IS	IX	S	U	SIX	X
IS	y	y	y	y	y	n
IX	y	y	n	n	n	n
S	y	n	y	y	n	n
U	y	n	y	n	n	n
SIX	y	n	n	n	n	n
X	n	n	n	n	n	n

- Table und Tablespace Locks der Stärke nach geordnet



- Zu allen Lockmodi A, B gibt es einen Modus C, so dass für alle Modi D gilt:
compatible(D,C) genau dann, wenn compatible(D,A) und compatible(D,B)

Deadlock

- Wir nehmen die Repeatable Read Variante der Meier Transaktionen, die etwa gleichzeitig beginnen sollen. Dann könnten sie folgenden Ablauf haben

```
T1begin
IS-Lock(Kontotabelle);
   T2begin
S-Lock1(betr. Page(s)); (evtl. mehrere Pages)
   IS-Lock2(Kontotabelle);
T1 Lesen Konto Meier;
   S-Lock2(betr. Page(s)); (dieselben Pages)
IX-Lock1(Kontotabelle); (Lock Promotion IS zu IX)
   T2 Lesen Konto Meier;
X-Lock1(betr. Page(s)); (evtl. mehrere Pages)
```

- An dieser Stelle, wo T1 den X-Lock verlangt, muss sie warten, weil bereits ein S-Lock auf derselben Page von einer anderen Transaktion, nämlich T2, besteht. Herr Meiers Transaktion kann aber weiterfahren

```
IX-Lock2(Kontotabelle); (Lock Promotion IS zu
IX
X-LOCK2(betr. Page(s))
```

- Jetzt kommt auch T2 ins Stocken, denn das System muss ihr den gewünschten X-Lock verweigern, da auf demselben Objekt bereits ein S-Lock einer anderen Transaktion, nämlich T1, besteht

→ Somit hat man nun die Situation, dass die Transaktion T1 darauf wartet, bis T2 den Lock freigibt, damit sie ihren gewünschten selber erhalten kann, und umgekehrt. Das heisst, T1 und T2 warten gegenseitig aufeinander
→ Deadlock

Wie wehrt sich das System gegen Deadlocks?

Deadlock Detection Algorithmen
Timeout

- Es gibt Algorithmen für periodisches Nachprüfen, ob irgendwo ein Deadlock vorliegt (wenn ja schiessen sie meist die jüngste Transaktion ab oder diejenige, die weniger geschrieben hat). Ein Deadlock kann auch aus einem längeren Zyklus bestehen als wie in unserem Falle aus einem Zweierzyklus, d.h. es können drei, vier oder mehr Transaktionen gegenseitig aufeinander warten. Nun ist es aber so, dass **Deadlock Detection Algorithmen** aufwendig sind und bei verteilten Systemen fast nicht zu realisieren

- Viele reale Systeme begnügen sich mit **Timeout**, das heisst, es wird grundsätzlich vom System jede Transaktion abgeschossen, die länger als eine bestimmte Zeit auf eine Ressource wartet. Die Wartezeitdauer kann als Systemparameter definiert oder bis zu einem bestimmten Grade vom Programm selber definiert werden

- Das Programm kann natürlich die Deadlockwahrscheinlichkeit auch beeinflussen. Als Beispiel nehme man das Beispiel 3 im Falle von Isolation Level Serializable. So wie die Lösung jetzt steht, ist Deadlock zwar ebenfalls möglich. Würde aber von Anfang an ein X-Lock verlangt, käme ein Deadlock sicher nicht vor (hingegen wäre es trotzdem möglich, dass die Transaktion abgeschossen wird, wenn z.B. tausend andere vor ihr auch noch darauf warten, einen X-Lock zu erhalten, und der Timeout Parameter zu klein ist)

- In den Anfängen gab es viele Deadlocks, in welche Indexes involviert waren, aber wie die Systeme unterdessen gemerkt haben, **ist es (fast) gar nicht nötig, auf Indexes Locks zu setzen**

Two Phase Locking Protokoll (2PL)

- Eine Transaktion erfüllt das 2PL, wenn sie ihre Locks quasi in zwei Phasen handhabt, in der ersten Phase aquiriert sie bloss neue Locks (oder verstärkt bereits erhaltene), und in der zweiten Phase lässt sie diese wieder los

- In der meist realisierten Variante des **strict 2PL**, wo alle Locks erst nach dem Ende der Transaktion losgelassen werden, das heisst nach dem Commit Zeitpunkt oder nach zu Ende geführtem Abort Prozess, werden ja gleich auch noch cascading aborts vermieden

Ein durch 2PL kontrollierter Ablauf ist serialisierbar

- 2PL garantiert nur den Isolation Level Repeatable Read, nicht aber die Vermeidung des Phantom Problems

- Der SQL Standard hingegen vermeidet bei Isolation Level Serializable auch das Phantom Problem. Deshalb klaffen die Serialisierbarkeitsbegriffe der klassischen Theorie und des Standards auseinander

Serialisierbarkeit (Definition)

- Betrachte einen Schedule

```
...R1(x) R2(y) W3(x) c1 W2(y)...
```

- Der **Serialisierbarkeitsgraph** eines Schedule von committed Transaktionen ist ein gerichteter Graph mit den beteiligten transaktionen als Knoten und Verbindungen Ti → Tj, falls eine Operation pi in Ti in Konflikt steht mit einer Operation pj in Tj so dass pi im Schedule vor pj erscheint

Ein Schedule ist genau dann serialisierbar, wenn sein Serialisierbarkeitsgraph azyklisch ist (keine Zyklen enthält

R1(x) R2(x) W1(y) c1 W2(x) c2 ist serialisierbar
R1(y) R2(x) W1(x) c1 W2(y) c2 ist nicht serialisierbar

Grundsatzbemerkungen

- Der Begriff der Serialisierbarkeit beinhaltet die Vorstellung, dass der Effekt parallel laufender Transaktionen irgendwie denselben Effekt auf die Datenbank haben sollen wie wenn sie seriell, hintereinander, laufen würden
→ Zweifelhaft, weil viele Transaktionen Timestamps in der DB hinterlassen

- Besser ist die Vorstellung (des Programmierers, der die Transaktion gestaltet), dass jede Transaktion sich vor den möglichen Effekten parallel laufender Transaktionen (evtl. anderen Typs) selber schützen soll. Es soll nicht verhelht werden, dass zu diesem Zwecke eine gewisse Kenntnis der Concurrency Theorie und Praxis der Systeme von Vorteil ist

Simon Flüeli, IT10b

Multiversion Concurrency Control

- Eine **Version** einer DB ist ein **Snapshot** zur Zeit eines Transaktions Commits, der das Resultat dieser Transaktion enthält, sowie die Resultate aller Transaktionen die früher committed haben

- In Multiversion DBMS werden mehrere DB Versionen bereitgehalten. Realisiert werden multiple Versionen der Datenbank, indem einfach mehrere Versionen der Tabellenzeilen bereitgehalten werden (nur von Zeilen, auf denen Update Aktivitäten stattgefunden haben)

- Daher kann jedes SQL SELECT... aus einer (möglichst aktuellen) Version lesen, und sogar jede Transaktion kann aus einer Version lesen
  → keine Read-Locks nötig
  → Readers müssen nicht auf Writers warten und Writers müssen nicht auf Readers warten (nur auf andere Writers)

- Die Anzahl erhältlicher Versionen ist beschränkt durch den zur Verfügung stehenden Platz, deshalb können unter Umständen auch lange Read-only Transaktionen abortet werden, weil gewisse zur Version gehörende Row Versionen nicht mehr da sind

- Abstrakte Darstellung eines Schedule

```
R1(x0) R2(y0) R3(z0) W1(x1) c1 W2(y2) c2
```

hier ist nach c2 von x, y und z in Current Version x1, y2 und z0

- Die Versionen derselben Row x sind z.B. x0, x1, x19, x23, usw. upgedatet durch die Transaktionen T1, T19, T23, etc.

- Die Rows, die von committed Transaktionen geschrieben worden sind, kriegen eine Nummer attached (spätestens vom nächsten Reader), welche in der zeitlichen Reihenfolge dem Commitpunkt der Transaktion entspricht

Read Consistency Multiversion Concurrency Control

Transaction Level Read Consistency
Statement Level Read Consistency

- Read-only Transaktionen (SET TRANSACTION READ ONLY) lesen einen Snapshot ("Transaction Level Read Consistency"), und bei Read/Write Transaktionen setzen Writers einen long duration write lock (bis nach Commit, die anderen warten), und Reads lesen aus der aktuellsten committed Version (genauer: "Statement Level Read Consistency", d.h. **alle** Reads aus **einem** SQL Statement lesen aus derselben Version

→ Read Consistency entspricht in etwa ANSI Read Committed

Snapshot Isolation Multiversion Concurrency Control

optimistic concurrency control

- Alle (!) Reads nehmen vom Snapshot des ersten Read der Transaktion
- Für Writes gilt eine "first committer wins" Regel (die anderen werden abgeschossen, dies ist eine echte "**optimistic concurrency control**")

(- praktische Systeme haben evtl. eine pragmatisch verbesserte Variante davon, mt einer "first updater wins" Regel, und brauchen deshalb Write-Locks auf Rows, die Looser werden abgeschossen, falls der erste Write-Locker committet)

→ Snapshot Isolation entspricht leider nur fast ANSI Serializable

Problem mit Snapshot Isolation

- Beispiel **write skew**

```
R1(x0) R1(y0) R2(x0) R2(y0) W1(x1) W2(y2) c1 c2
```

- Wenn nun eine Bedingung wie z.B. x+y ≥ 0 dazukommt (die Summe zweier Konten einer Person darf nicht negativ sein), dann kann sie durch write skew verletzt werden, obwohl der Schedule unter Snapshot Isolation laufen kann

- Anderes Beispiel

```
R1(x0) R2(x0) R2(y0) W1(x1) c1 R3(x1) R3(y0) W3(z3) c3 W2(y2) c2
```

- Dieser Schedule ist sogar ohne Zusatzbedingungen **nicht serialisierbar**. der Serialisierbarkeitsgraph ist
  T3 → T2 → T1 → T3, also zyklisch, obwohl er **erlaubt ist bei Snapshot Isolation**

- Das heisst, Snapshot Isolation verhindert alle ANSI Phänomene (auch als "anomaly serializable" bekannt), aber der Standard ANSI 1992 verlangt eigentlich zusätzlich zur Abwesenheit der Phänomene, dass Serializable must provide "what is commonly known as fully serializable execution)
  → obiger Schedule wäre in Repeatable Read nicht erlaubt

Gibt es eine Rettung für Snapshot Isolation?

Fekete2000 hat gezeigt, dass in jedem nicht serialisierbaren Snapshot
Isolation Schedule der Zyklus im Serialisierbarkeitsgraph besteht aus
mindestens zwei read/write Konflikten paralleler Transaktionen und
möglicherweise einigen read/write, write/read oder write/write Konflikten in nicht parallel laufenden Transaktionen
- In Fekete2005, Making Snapshot Isolation Serializable, ACM TODS, geht es darum, Kriterien aufzustellen, mit denen getestet werden kann, ob eine Anwendung unter Snapshot Isolation serializable ist oder nicht. Der Artikel führt beispielhaft aus wie es aussieht beim TPC-C benchmark und weist nach, dass die Transaktionen von TPC-C serializable sind unter Snapshot Isolation

Schlussbemerkungen:

Falls der Programmierer nicht sicher ist, ob die ihm zur Verfügung stehenden Systemparameter Serialisierbarkeit oder die Vermeidung des
Phantom Problems auch wirklich garantieren, kann er auch (in den meisten?) Systemen mit Multiversion Concurrency Control auf Tabellenebene Sperren (Locks) selber setzen, auch Read-Locks. Die entsprechenden Lockmodi mögen dann halt zum Beispiel RS(row share), RX(row exclusive), S(share), SRX(share row exclusive), und X(exclusive) heissen (statt IS, IX, S, SIX und X). Die Compatibility Matrix ist mit geänderten Namen dieselbe wie gehabt (ohne U-Lock). Weil im Falle von Referential Integrity auch der Concurrency Schutz auf weitere Tabellen übertragen werden muss, sollte man keine Datenbank
Struktur nehmen, die ein *Referential Integrity Festival* enthält (leider alles schon vorgekommen).

Aufgaben Concurrency

1) Im Meier Konto Beispiel mit

   Frau Meiers Transaktion

   R_1: Lesen Konto Meier
   W_1: Schreiben Konto Meier mit Saldo:=Saldo+100

   und Herr Meiers Transaktion

   R_2: Lesen Konto Meier
   W_2: Schreiben Konto Meier mit Saldo:=Saldo - 100

   zähle man alle möglichen Varianten von Abläufen im System ohne Isolierschutz auf (zB $<R_1, R_2, W_1, W_2>$ etc), und verfolge den Saldo.

Lösung Aufgabe 1:

Weil die Reihenfolge $<R_1,W_1>$ der Aktionen der Transaktion T_1 unter sich feststeht und analog für die Aktionen von T_2, sind nur die folgenden gemischten Abläufe beider Transaktionen zu untersuchen: $<R_1,W_1,R_2,W_2>$, $<R_1,R_2,W_1,W_2>$, $<R_1,R_2,W_2,W_1>$, $<R_2,R_1,W_1,W_2>$, $<R_2,R_1,W_2,W_1>$, $<R_2,W_2,R_1,W_1>$.
In den Fällen $<R_1,W_1,R_2,W_2>$ und $<R_2,W_2,R_1,W_1>$ sind die Transaktionen T_1 und T_2 serialisiert, da ist alles in Ordnung, und der Schlusssaldo gleich b, dem Anfangssaldo.
Den Fall $<R_1,R_2,W_1,W_2>$ haben wir schon besprochen, da war der Schlusssaldo b-100.
Es verbleiben $<R_1,R_2,W_2,W_1>$, $<R_2,R_1,W_1,W_2>$ und $<R_2,R_1,W_2,W_1>$. In allen diesen drei Fällen bestehen die ersten beiden Aktionen aus dem Lesen von S_1 und S_2 (als b). Es kommt also nur darauf an, wer zuletzt schreibt, somit ist bei $<R_1,R_2,W_2,W_1>$ und $<R_2,R_1,W_2,W_1>$ der Schlusssaldo b+100 und bei $<R_2,R_1,W_1,W_2>$ ist er b - 100.
Zusammengefasst kann man also sagen, dass in den Fällen $<R_1,R_2,W_2,W_1>$ und $<R_2,R_1,W_1,W_2>$ die Bank zu kurz kommt und in den Fällen Fällen $<R_1,R_2,W_1,W_2>$ und $<R_2,R_1,W_1,W_2>$ die Meiers. Die anderen beiden Fälle $<R_1,W_1,R_2,W_2>$ und $<R_2,W_2,R_1,W_1>$ laufen korrekt.

2) Programme A und B ändern zwei Grössen x und y wie folgt:

   A: x := x - 1 ; y := y + 1
   B: x := 2 • x; y := 2 • y

   Es soll ein Integrity Constraint "x + y = 0" aufrecht erhalten werden. Beide Programme sind für sich gesehen korrekt, das heisst, führen korrekte Zustände in korrekte über. Man gebe ein Beispiel eines korrekten Anfangszustandes sowie einen vermischten Ablauf der Programme, der in einem System ohne Isolierschutz einen inkorrekten Zustand erzeugt.

Lösung Aufgabe 2:

Zum Beispiel der Ablauf x := x - 1; x := 2 • x; y := 2 • y; y := y + 1 führt den konsistenten Zustand (x = -1, y = 1) in den inkonsistenten (x = -4, y = 3) über.

3) Man gebe weitere Beispiele für das Phantom Problem.

Lösung Aufgabe 3:

Jedes SQL INSERT, UPDATE oder DELETE kann ein Phantom Problem erzeugen, wenn gleichzeitig eine andere Lese Transaktion die Tabelle liest.

4) Gegeben ist die Compatibility Matrix für die Lock Modi IS, S, IX, X:

	IS	S	IX	X
IS	y	y	y	n
S	y	y	n	n
IX	y	n	y	n
X	n	n	n	n

Für Lockmodi A, B bedeute Comp(A,B), dass A und B verträglich sind ("y" in Matrix).

Ist Comp transitiv? (für alle Modi A,B,C (Comp(A,B) and Comp(B,C) → Comp(A,C))

Zeige, dass es keinen Lock Modus A gibt, der genau gleich stark ist wie die beiden Modi S und IX zusammen, das heisst so, dass für alle Modi Z gilt:

Comp(Z,A) ↔ Comp(Z,S) und Comp(Z,IX).

Lösung Aufgabe 4:

Transitivität für Comp gilt nicht, weil zum Beispiel Comp(IX,IS) und Comp(IS,S) gilt, nicht aber Comp(IX,S).

Würde für einen Lockmodus A gelten, dass für alle Modi Z gilt:

Comp(Z,A) ↔ Comp(Z,S) und Comp(Z,IX),

dann müsste insbesondere

für alle Modi Z (¬Comp(Z,S) or ¬Comp(Z,IX) → ¬Comp(Z,A)) gelten,

d.h. ein solches A müsste überall, wo S oder IX ein "n" hat, ebenfalls ein "n" haben. Dafür käme aber nur A = X in Frage. Nun ist aber Comp(IS,S) und Comp(IS,IX), aber nicht Comp(IS,X).

5) Man gebe ein Beispiel für das **Lost Update Problem** anhand zweier gleichzeitig ablaufender Instanzen eines Programmes mit den Anweisungen

   temp := counter + 1; counter := temp;

   in einer Umgebung ohne Isolierschutz.

Lösung Aufgabe 5:

A1: temp := counter + 1;
A2: temp := counter + 1;
A1: counter := temp;
A2: counter := temp;

A2 verliert den Update.

6) Begründe, dass Two Phase Locking, 2PL, Repeatable Read garantiert, nicht aber die Vermeidung des Phantom Problems.

Lösung Aufgabe 6:

Wollte eine Transaktion eine Entität zum zweiten Male lesen, nachdem sie begonnen hat, Locks loszulassen, müsste sie erneut einen Lock verlangen, was aber gemäss Definition von 2PL nicht geht.
Bestehen in einer 2PL Transaktion die Locks nicht auf grösstmöglicher Granularität (von Table), kann jederzeit das Phantom hereinfunken nach bereits besprochenem Muster.

7) Was müsste man im Locking Paradigma vorkehren, um Deadlocks vollständig zu vermeiden?

Lösung Aufgabe 7:

Entweder hat man eine vollständige Uebersicht über die Logik aller relevanten Programme, und kann garantieren, dass die Logik Deadlocks verhindert (das ist allerdings nicht einfach, weil vor allem in älteren Systemen auch die Indexe in Betracht gezogen werden müssen), oder die Programme verlangen alle Locks in grösstmöglicher Granularität (was übrigens meist auch in Multiversion Systemen möglich ist). Das letztere ist aber nur bedingt empfehlenswert, weil es Concurrency massiv hindern kann.

8) Der Schedule

   $R1(x_0)$ $R2(x_0)$ $R2(y_0)$ $W1(x_1)$ c_1 $R3(x_1)$ $R3(y_0)$ $W3(z_3)$ c_3 $W2(y_2)$ c_2

   ist, wie wir gesehen haben, nicht serialisierbar, obwohl er erlaubt ist bei Snapshot Isolation.

   Warum wäre er im Locking Paradigma bei Repeatable Read nicht erlaubt?

Lösung Aufgabe 8:

Es ginge dann um den Schedule

$R1(x)$ $R2(x)$ $R2(y)$ $W1(x)$ c_1 $R3(x)$ $R3(y)$ $W3(z)$ c_3 $W2(y)$ c_2

ohne multiple Versionen der Entitäten x,y und z. W1(x) könnte nicht ausgeführt werden, weil Transaktion 2 an dieser Stelle noch einen S-Lock auf x hätte.

9) Kann Lost Update in Read Consistency Multiversion Concurrency Control vorkommen?

Lösung Aufgabe 9:

Nein, wegen der trotz multipler Versionen vorhandenen Write-Locks, obwohl Repeatable Read nicht garantiert ist bei Read Consistency Multiversion Concurrency Control.

10) Man überlege sich die Rolle des Phantom Problems bei Read Consistency Multiversion Concurrency Control und bei Snapshot Isolation Multiversion Concurrency Control.

Lösung Aufgabe 10:

Bei Snapshot Isolation und für Readonly Transaktionen in Read Consistency Multiversion bildet das Phantom kein Problem, solange die Business Vorlage nicht erwartet, dass zwischenzeitliche Writes anderer Transaktionen in ersterer berücksichtigt werden sollten (in beiden Fällen wird ein Snaphot gelesen).
Bei Read/Write Transaktionen unter Read Consistency Multiversion hingegen kann das Phantom schon problematisch sein (siehe Buchungsbeispiel).

Simon Flüeli, IT10b

Aufgaben Recovery

1 Wäre es eine gute Idee, als LSN (log sequence number) einen geeigneten Timestamp zu nehmen (zB Timestamp zum Zeitpunkt des Erstellens des Log-Records)?
→ Nicht unbedingt, da die System Clock von aussen beeinflusst werden kann, und dadurch die Monotonie der LSN gefährdet sein kann (aufsteigende log sequence numbers). Sehr einfach als Wahl der LSN ist eine RBA, eine relative byte address, innerhalb des linearen Logspace (der Log beginnt bei Byte 0 und hört vielleicht auf bei Byte zwei hoch achtundvierzig minus eins, wenn die RBA aus sechs Bytes besteht)

2 ARIES (Algorithm for Recovery and Isolation Exploiting Semantics, Mohan 1992, das im Unterricht besprochene Modell) überlässt dem Buffer Manager ziemlich viele Freiheiten (ausser zum Beispiel, dass WAL eingehalten werden muss usw). Bei welchen Gelegenheiten würde man dem Buffer Manager empfehlen, Pages hinauszuschreiben (auch ohne dass eine durch Transaktionen verursachte Notwendigkeit gegeben ist wie zum Beispiel dass der Platz gebraucht wird)?
→Zum Beispiel bei 'hot spot pages' ab und zu, das sind Pages, die immer wieder von Transaktionen gebraucht werden und ansonsten die Tendenz hätten, selten oder nie hinausgeschrieben zu werden (je länger sie nicht auf festen Diskspace hinausgeschrieben werden, desto mehr Log Records müssen im Recover Fall angewendet werden). Zweitens sollte Buffermanager darauf achten, dass er dirty pages nicht auf ewig bei sich behält. Man könnte sich zum Beispiel vorstellen, dass er dafür sorgt, dass jede geschriebene Page pro Tag oder pro Stunde einmal hinausgeschrieben wird (kann asynchron passieren). Das ewige Behalten ohne Flush könnte im Extremfall dazu führen, dass der aufbewahrte Log und die fuzzy image copies nicht mehr ausreichen für media oder system crash recovery (bei nonfuzzy image copies sollte dies allerdings nicht passieren)

3 Wie werden cascading rollbacks vermieden? (man versteht nach diesem Kapitel, was für einen immensen Aufwand dies bedeuten würde für das System)
→ Durch Locking mit strict 2PL. Siehe Kapitel Concurrency

4 Man erkläre, wann es möglich ist, auch eine Wiederherstellung einer einzelnen Page zu gewährleisten
→ Wenn man sich im klaren ist darüber, bis zu welchem Punkt die Wiederherstellung gehen soll, das heisst zum Beispiel bis zu welchem LSN, oder aber dass man nur die Effekte von LURs haben möchte, welche Commited sind, oder ähnlich. Utilities stellen diese Möglichkeit unter Umständen zur Verfügung, aber die Sache ist etwas heikel. Das System selber macht eventuell interne recover page (bis zum Zeitpunkt des Entdeckens dass die page 'broken' ist), ohne dass der Benutzer etwas merkt

5 Wir nehmen an, dass in einem System zum End of LUR Processing ein Bufferflush aller diese LUR betreffenden Daten Pages stattfinden würde. Vorteile? Nachteile? Konsequenzen?
→ Beginnen wir mit den Konsequenzen. Das Logging von EndLUR müsste aus zwei Teilen bestehen, Log(BeginEndLUR), welches den logischen Zeitpunkt definieren würde, ab welcher die LUR als Commited gilt. Dann Bufferflush und anschliessend Log(EndEndLUR). Der Vorteil wäre, dass Recovery Processing nur REDO machen müsste für LURs, bei denen Log(BeginEndLUR) da ist aber Log(EndEndLUR) nicht (analog bei rollbacked...). Der Nachteil wäre eine Einbusse in der Performance, vor allem bei von hot spot pages betroffenen Transaktionen

6 Was kann vom System getan werden, damit bei Restart das Ende des Log File klar erkenntlich ist?
→ Das System kann abstürzen mitten im Schreiben eines Logrecords. Bei Restart können aber nur vollständig geschriebene Logrecords verwendet werden. Logrecords können sich über Pagegrenzen oder gar Logfilegrenzen erstrecken. Das Ende jeden Logrecords kann markiert sein, der Anfang eines Logrecords muss markiert sein, für das System erkenntlich. Das Ende des letzten ganzen Logrecords wäre dann vor der letzten Beginnmarkierung

7 Diskutiere Möglichkeiten, Batch Programme beliebig restartable zu machen (damit sie zu jedem Zeitpunkt durch Operator Invention abgeschossen werden können, und später einfach wieder gestartet werden können, ohne dass etwas verlorengeht oder zweimal gemacht wird)
→ Beliebt ist, die Inputrecords für das Programm in Tabellen zu stellen, und nach der Verarbeitung jeden Records (das Verarbeitete in eine andere Tabelle zu schreiben und) den oder die Input Records zu löschen, und sagen wir, nach je 50 Records einen Commit vom Programm aus zu verlangen. Rollback nach Programm Abschuss stellt dann Input und Outputrecords bis zum letzten Commit zurück. Restart setzt dort wieder auf. Es gibt aber noch andere Möglichkeiten ...

8 Viele Systeme gestatten "Recover tablefile to LSN" oder gar "Recover .. to Timestamp". Was muss das System tun und worauf muss der Benutzer achten?
→ Das System muss REDO anwenden bis zur betreffenden LSN, und UNDO für alle LURs, das Tablefile betreffend, die beim LSN noch nicht committet waren (es wird anschliessend einen Checkpoint Record schreiben).
Der Benutzer muss sich im Klaren sein, dass dadurch Updates verloren gehen können. Gerne macht das kein System Administrator, nur im Notfall.

9 Ist es möglich, Log Buffer zu füllen bevor flush out auf Logfile? (ein paar Pages gleichzeitig hinauszuschreiben ist billiger, das heisst insgesamt performanter)
→ Im Prinzip ja. Es braucht aber einen Dämon, der zum Beispiel alle 100ms den Buffer flushed (Inhalt des Log Buffers auf File hinausschreibt), weil es sonst eine Art Deadlock geben könnte. Ein Beispiel wäre ein Batch Programm, das gerade einen Commit gemacht hat, und warten muss, bis der Commit im System realisiert ist (Commit Log Record auf File draussen), sonst nichts los ist, und der Log Buffer Manager warten würde, bis der Buffer voll wäre. Dann warten Programm und Log Buffer Manager aufeinander.

10 Man beschreibe genauer, warum bei indoubt Transaktionen nach System Absturz eventuell von Hand eingegriffen werden muss.
→ Der blöde Fall ist der, dass ein Slave in der zweiten Phase des 2PC Protokolls Commit gemeldet hat und keine Antwort vom Master erhalten hat, die er noch im eigenen Logfile festhalten konnte vor seinem Absturz. Dann besteht bei Restart des Slave keine Möglichkeit, zu entscheiden, ob der Master für alle Commit befohlen hat oder für alle Abort. Solche Fälle sind in der Praxis mühsam, kommen aber selten vor

11 Ist es zwingend nötig, dass bei Pseudokonversationeller Programmierung (typischerweise der Fall bei Internet Programmen) Ueberbuchungen zB von Flügen stattfinden müssen?
→ Nein. Es könnte logisch gelockt werden. Das Problem dabei ist, dass das System nicht weiss, wie lange der Benutzer für eine Antwort braucht und ob er überhaupt antwortet. Die Fluggesellschaft will aber nicht eine Unterbesetzung, weil die Benutzer die Business Transaktionen nicht abschliessen, verzichtet deshalb auf logical locking ("flag setzen"), was die Möglichkeit von Ueberbuchung einschliesst (weil auch noch eine Kreditkarten Buchung oder Ueberprüfung dabei ist). Da aber viele Leute dann im letzten Moment doch noch absagen, ist das Problem nicht so gross. Aus astrologischen Gründen ist es übrigens nicht empfehlenswert, bei einem Flug dabei zu sein, bei dem im letzten Moment sehr viele Leute abgesagt haben