Datenbanken

Professor Dr. Georg Lausen
Institut für Informatik
Universität Freiburg

Universität Freiburg
Kursvorlesung Datenbanken und Informationssysteme

Universitärer Lehrverbund Informatik
Lehreinheit Datenbank– und Informationssysteme

Gliederung

weitere Informationen

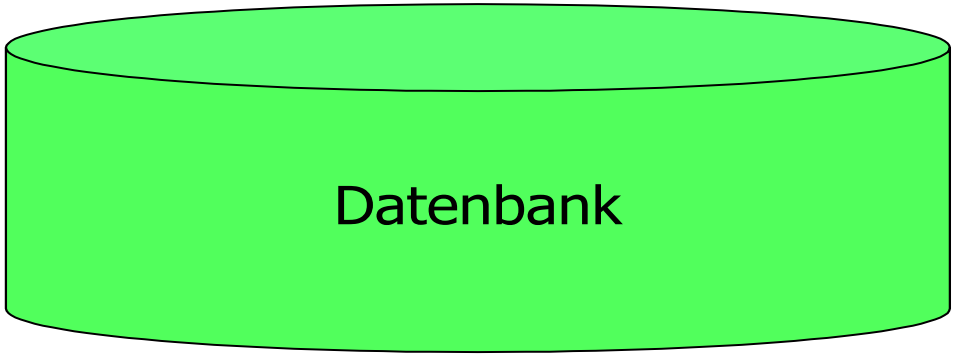
- **Übungsbetrieb:** siehe Lehrstuhlseiten.
- **Klausur:** 18. Februar 2003, 9:00 Uhr.

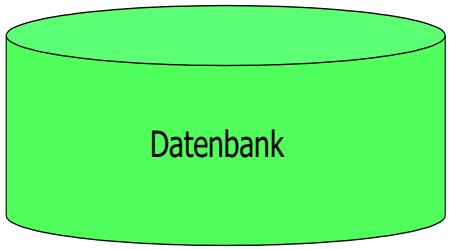
nehmen Sie aktiv an den Übungsgruppen teil!

empfohlene begleitende Literatur

- A. Kemper, A. Eickler.
  Datenbanksysteme – Eine Einführung.
  Oldenbourg, 4. Auflage 2001.
- G. Vossen.
  Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme.
  Oldenbourg, 4. Auflage 2000.
- R. Ramakrishnan, J. Gehrke.
  Database Management Systems.
  WCB/McGraw-Hill, 1998.

Einführung: Grundbegriffe



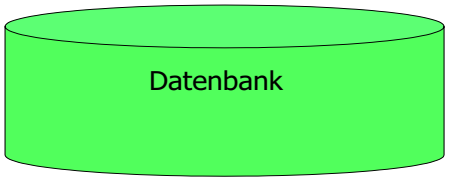


Studierenden-Datenbank

Hans Eifrig hat die Matrikelnummer 1223. Seine Adresse ist Seeweg 20. Er ist im zweiten Semester.

Lisa Lustig hat die Matrikelnummer 3434. Ihre Adresse ist Bergstraße 111. Sie ist im vierten Semester.

Maria Gut hat die Matrikelnummer 1234. Ihre Adresse ist Am Bächle 1. Sie ist im zweiten Semester.



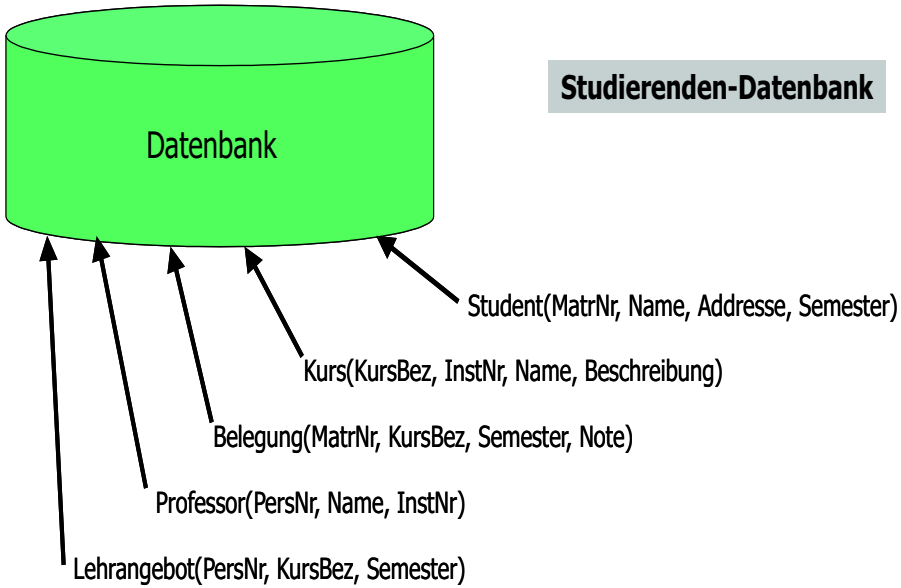
Studierenden-Datenbank

Hans Eifrig hat die Matrikelnummer 1223. Seine Adresse ist Seeweg 20. Er ist im zweiten Semester.

Lisa Lustig hat die Matrikelnummer 3434. Ihre Adresse ist Bergstraße 111. Sie ist im vierten Semester.

Maria Gut hat die Matrikelnummer 1234. Ihre Adresse ist Am Bächle 1. Sie ist im zweiten Semester.

Student	MatrNr	Name	Adresse	Semester
	1223	Hans Eifrig	Seeweg 20	2
	3434	Lisa Lustig	Bergstraße 111	4
	1234	Maria Gut	Am Bächle 1	2
	. . .	. . .	. . .	. . .



Studierenden-Datenbank

Was ist eine Datenbank (DB)?

- Eine Datenbank ist eine strukturierte Menge von dauerhaften (persistenten) Daten.

  Typischerweise verwendet man als Strukturierungskonzept **Relationen** und spricht von einer **relationalen Datenbank**.
- Eine Datenbank repräsentiert Informationen über einen Zustand aller relevanten **Objekte** und **Beziehungen** eines interessierenden Anwendungsbereiches (Miniwelt).

  In einer relationalen Datenbank werden die Objekte und Beziehungen durch die einzelnen Elemente der Relationen, den sogenannten **Tupeln**, repräsentiert.

Was ist ein Objekt, was eine Beziehung?

- Ein **Objekt** ist ein wohlunterscheidbares physisches oder gedankliches Konzept der betrachteten Miniwelt.
- Eine **Beziehung** ist ebenfalls ein wohlunterscheidbares physisches oder gedankliches Konzept der betrachteten Miniwelt, das jedoch – im Unterschied zu einem Objekt – über Objekten definiert ist.

  Die Unterscheidung zwischen Objekten und Beziehungen ist häufig subjektiv.
- Beispiele für Objekte sind die einzelnen Studierenden, oder auch Kurse; Beispiele für Beziehungen sind Belegungen von Kursen durch Studierende, oder das Lehrangebot.
- In einer relationalen Datenbank werden gleichartige Objekte und Beziehungen jeweils in **Relationen** zusammengefaßt.

Was ist eine Anfrage (query)?

- Ein mengenwertiger, deklarativer Ausdruck zur Extraktion von Daten.

```
SELECT P.Name
   FROM Professor P
   WHERE P.InstNr = '15-1'
```

```
SELECT S.Name, S.MatrNr, K.Name
   FROM Student S, Belegung B, Kurs K
   WHERE S.MatrNr = B.MatrNr AND B.KursBez = K.KursBez
```


Sind auch Änderungen, Einfügungen und Löschungen möglich?

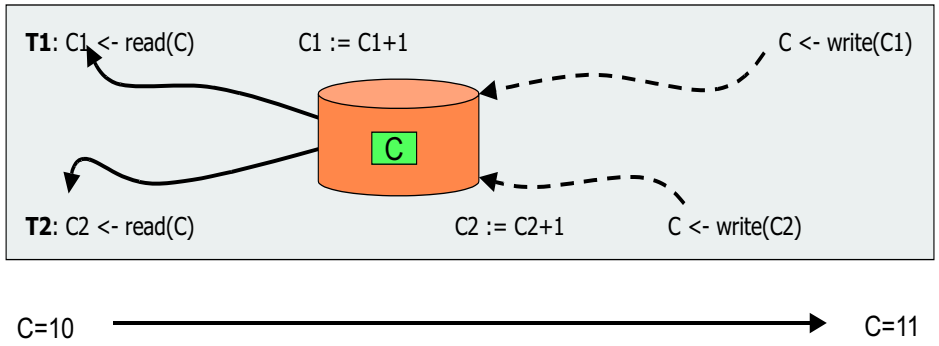
... na klar.

Sie werden ähnlich mengenwertig, deklarativ formuliert. Wir werden dies später kennen lernen.

Was ist eine Transaktion?

- Im allgemeinen ein Programm, das einem in sich abgeschlossenen Verarbeitungsschritt innerhalb der Anwendungen der betreffenden Miniwelt entspricht.
- Beispiele:
  - Erstelle für jeden Studierenden eine Liste der belegten Kurse.
  - Trage die Noten für die belegten Kurse ein.
  - Wechsel zwischen Kursen.
  - usw. usw.

Ausführungen von Transaktionen müssen kontrolliert werden!



Nicht jede parallele (zeitlich verzahnte) Transaktionsausführung ist zulässig!

Welche Eigenschaften müssen Transaktionen haben?

Atomicity: Eine Transaktion ist (logisch) eine nicht weiter zerlegbare Einheit; die durch sie bewirkten Änderungen eines DB-Zustandes sind atomar und werden entweder vollständig oder gar nicht vorgenommen (*alles-oder-nichts*-Prinzip).

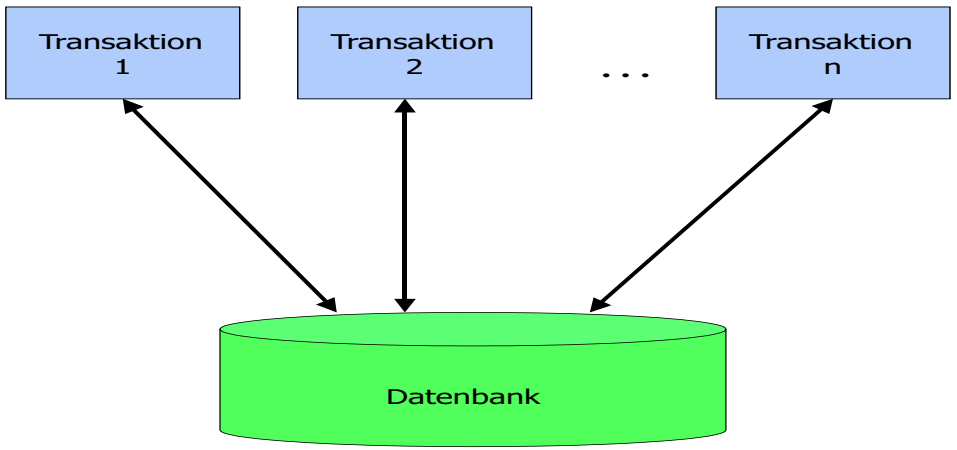
Concistency: Eine Transaktion bewirkt einen konsistenten Zustandsübergang in der DB. Inkonsistente Zwischenzustände können im allgemeinen nicht ausgeschlossen werden und sind somit zulässig.

Isolation: Im allgemeinen wird eine Menge von Transaktionen gleichzeitig in ihrer Ausführungsphase sein (Mehrbenutzerbetrieb). Obwohl somit die DB-Zugriffe der einzelnen Transaktionen potentiell verzahnt ablaufen, bleibt dies für den Benutzer verborgen (*simulierter* Einbenutzerbetrieb).

Durability: Wenn eine Transaktion erfolgreich ihr Ende erreicht hat, dann sind alle von ihr verursachten Änderungen dauerhaft (*persistent*), d.h. sie überdauern alle möglichen Fehlersituationen der DB.

Wie können die ACID-Eigenschaften erreicht werden?

So geht es nicht!



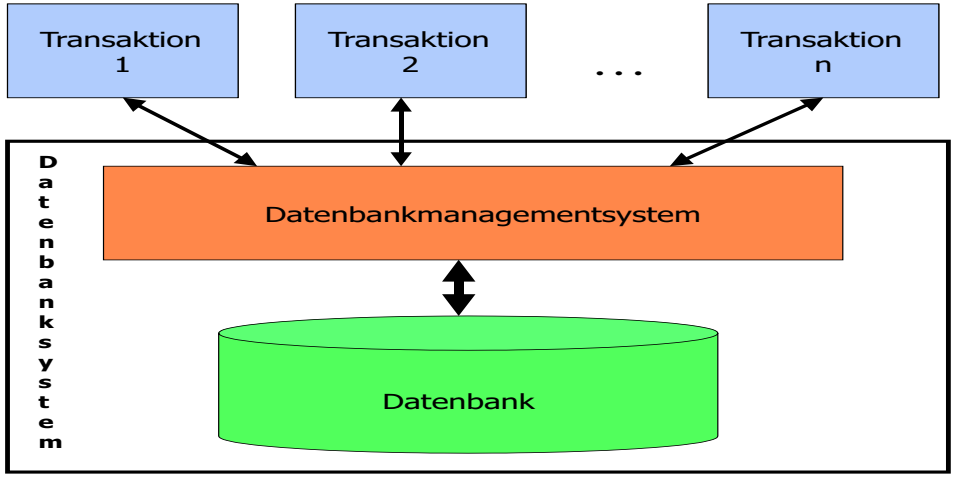
. . . nämlich ...

DB brauchen ein Datenbankmanagementsystem (DBMS).

- Eine Menge von Programmen, die auf einer DB eine bequeme, effiziente und sichere Abspeicherung, Änderung und Wiedergewinnung von Informationen ermöglicht,
- die eine Datenbank kapselt, d.h., Zugriff zur Datenbank sind nur über spezielle Schnittstellen zugelassen,
- die ein Modell der betreffenden Miniwelt implementiert.

Eine DB zusammen mit einem DBMS nennen wir ein **Datenbanksystem (DBS)**.

Basisarchitektur eines DBS



Was sind die charakteristischen Eigenschaften eines DBS?

- integrierte Verwaltung (unter Umständen sehr großer) persistenter Datenbestände mit gewährleisteter Integrität der Daten,
- weitgehende Unabhängigkeit zwischen Verarbeitung der Daten und ihrer Abspeicherung,
- deklarative, mengenorientierte Anfragesprachen ergänzt um einen Anfrageoptimierer,
- kontrollierter Mehrbenutzerbetrieb,
- Datensicherheit bei Fehlerfällen,
- Schutz vor Mißbrauch der Daten.

Was sind die Anwendungsgebiete für Datenbanken?

- Wirtschaft,
- Verwaltung,
- Technik,
- allgemein überall dort, wo persistente Datenmengen mit einem hohen Qualitätsanspruch verarbeitet werden müssen.

Was sind die Herausforderungen heute?

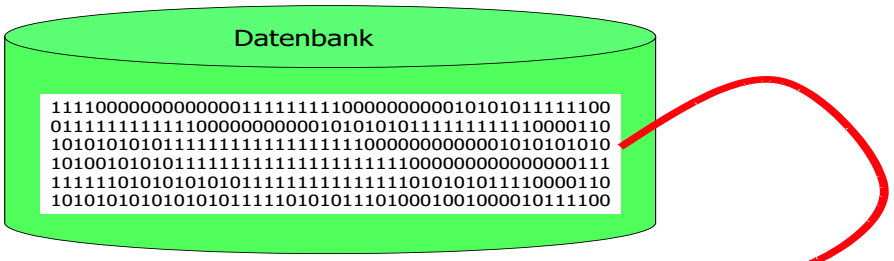
- das **Internet** (World-Wide-Web), mit seiner verteilten Datenhaltung und seinen heterogenen Informationssystemen,
- die Unterstützung von kompletten, lang laufenden **Geschäftsprozessen** (workflows),
- **mobile** Informationssysteme,
- die fortschreitende Digitalisierung informationstragender Medien (**Multimedia**).

Einführung: Datenmodell

Warum brauchen Datenbanken ein Datenmodell?

- Daten sind Folgen von Bits - aber wer will damit arbeiten? Um Daten zu verarbeiten werden **Abstraktionen** benötigt.
- Eine Datei ist eine erste Abstraktion einer Menge von gleichartigen, zusammengehörenden Daten.
- Basiert die Verarbeitung der Daten direkt auf den Formaten der Dateien, so kann dies zu schwerwiegenden Problemen führen: Y2000-Problem!

Von Daten zu Relationen: das relationale Datenmodell.



Student	MatrNr	Name	Adresse	Semester
	1223	Hans Eifrig	Seeweg 20	2
	3434	Lisa Lustig	Bergstraße 111	4
	1234	Maria Gut	Am Bächle 1	2
	. . .	. . .	. . .	. . .

In einer Datenbank werden 3 Abstraktionsebenen unterschieden.

Die Datentypen einer Abstraktionsebene werden in einem **Schema** zusammengefaßt.

physikalische Ebene/Schema

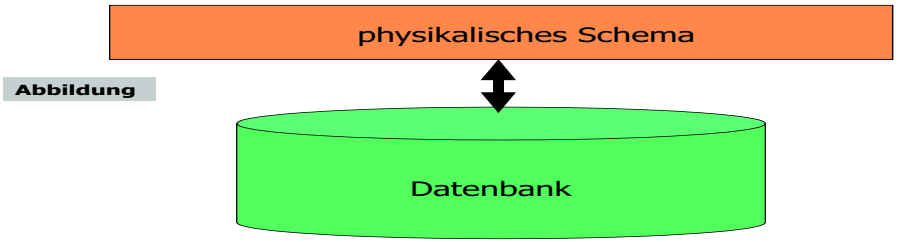
konzeptuelle Ebene/Schema

externe Ebene/Schemata

physikalische Ebene

physikalische Ebene/Schema: Es wird festgelegt, wo und wie die Daten auf den Speichermedien physisch abgelegt werden und welche Hilfsstrukturen zur Effizienzsteigerung zur Verfügung gestellt werden sollen.

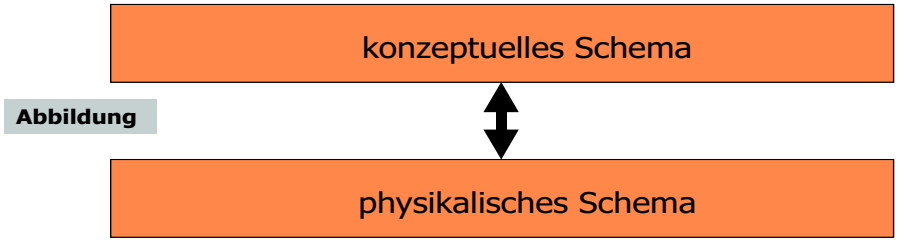
Die angebotenen Datentypen sind die Grundlage für die Implementierung der Datentypen der konzeptuellen Ebene.



konzeptuelle Ebene

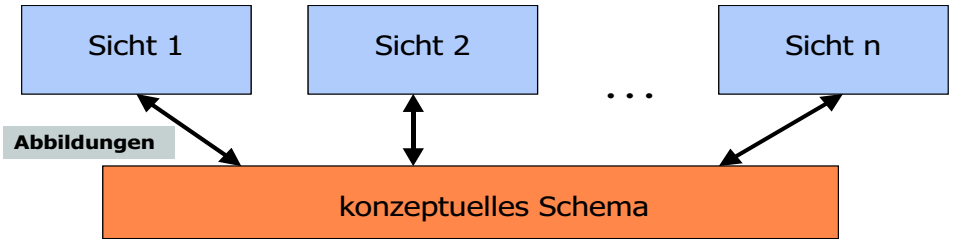
konzeptuelle Ebene/Schema: Die Datentypen werden unabhängig von den Details einer physischen Implementierung definiert. Ziel ist eine möglichst adäquate Repräsentation der relevanten Objekte der Miniwelt.

Die festgelegten Datentypen sind die Grundlage für die Anwendungsprogrammierung. Sie definieren das zugrundeliegende globale Gesamtmodell der Miniwelt.

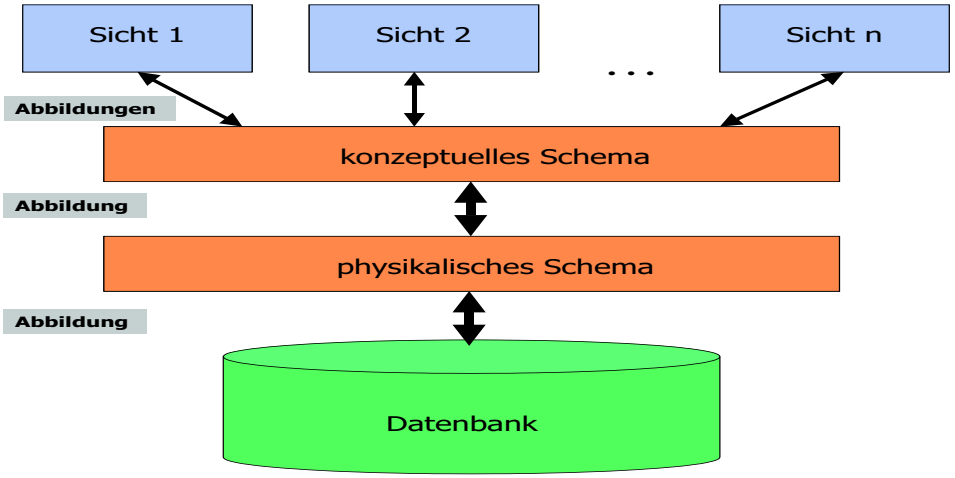


externe Ebene

externe Ebene/Schema: Für einzelne Benutzergruppen werden speziell für deren Belange geeignete Datentypen definiert. Es werden **Sichten** auf das konzeptuelle Schema definiert. Diese Sichten basieren auf den Datentypen des konzeptuellen Schemas.



Architektur einer DB nach ANSI/SPARC:



Datenunabhängigkeit: der entscheidende Vorteil einer mehrschichtigen Architektur

physische Datenunabhängigkeit:

Die Anwendungen arbeiten nicht direkt auf den Dateien, sondern auf den Datentypen des konzeptuellen Schemas.

Das DBMS bildet Operationen der konzeptuellen Ebene automatisch in Operationen des Dateisystems ab.

Änderungen der physischen Repräsentation bewirken keine Änderungen in den Anwendungsprogrammen.

Datenunabhängigkeit: der entscheidende Vorteil einer mehrschichtigen Architektur (fortgesetzt)

logische Datenunabhängigkeit:

Die Anwendungen arbeiten nicht direkt auf den Datentypen des konzeptuellen Schemas, sondern auf den Datentypen ihres externen Schemas.

Das DBMS bildet Operationen der externen Ebene automatisch in Operationen der konzeptuellen Ebene ab.

Änderungen der konzeptuellen Repräsentation bewirken keine Änderungen in den Anwendungsprogrammen.

Ein Datenmodell

besteht aus Sprachen und Tools zur Definition von

konzeptuellen und externen Schemata. Schemata werden mittels einer **Datendefinitionssprache** (DDL) abgefaßt.

Operationen. Die Operationen auf den mittels der DDL definierten Datentypen werden in einer **Datenmanipulationssprache** (DML) abgefaßt.

Typische Operationen sind `select, insert, delete` und `update` auf Instanzen der Datentypen.

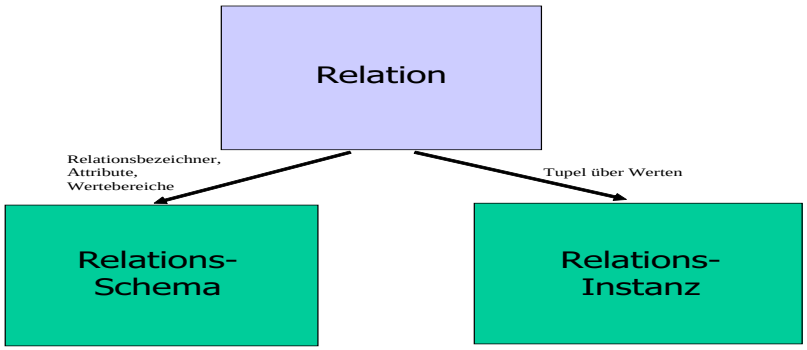
Bedingungen (Constraints). Die Daten in der Datenbank müssen alle Bedingungen erfüllen. Bedingungen werden als Teil der DDL unter Zuhilfenahme der DML abgefaßt.

Bedingungen können statisch, d.h. einen festen Zustand der DB, oder dynamisch, d.h. einen Zustandsübergang betreffend sein. Man bezeichnet sie auch als **Integritätsbedingungen (integrity constraints)**.

Eine Bemerkung zur Effizienz.

Ein DBS besitzt als Teil seiner DDL eine **Speicherdefinitionssprache** (SDL), mittels der die physische Speicherung der Daten beeinflußt und somit die Performance gesteigert werden kann.

Einführung: Das Relationale Datenmodell



Relationsschemata

```
Student(MatrNr:INTEGER,Name:STRING,Adresse:STRING,Semester:INTEGER)
Kurs(KursBez:STRING,InstNr:INTEGER,Name:STRING,Beschreibung:STRING)
Belegung(MatrNr:INTEGER,KursBez:STRING,Semester:INTEGER,Note:INTEGER)
Professor(PersNr:INTEGER,Name:STRING,InstNr:INTEGER)
Lehrangebot(PersNr:INTEGER,KursBez:STRING,Semester:INTEGER)
```

Relationsbezeichner: Student, Kurs, Belegung, Professor, Lehrangebot

Attribute: MatrNr, Name, Adresse, Semster, KursBez, InstNr, ...

Wertebereiche: INTEGER, STRING

Instanzen zu den Schemata: Relationen

Student	MatrNr	Name	Adresse	Semester
	1223	Hans Eifrig	Seeweg 20	2
	3434	Lisa Lustig	Bergstraße 111	4
	1234	Maria Gut	Am Bächle 1	2
	...	...	...	...

Kurs	KursBez	InstNr	Name	Beschreibung
	DB	15-1	Datenbanken	Grundlagen von DB
	DB&I	15-1	Datenbanken und Internet	Internettechnologien, wie XML, ...
	MST	15-2	Mikrosystemtechnik	Einführung
	...	...	...	...

Belegung	MatrNr	KursBez	Semester	Note
	1223	DB	WS2001/02	2
	3434	DB	WS2000/01	1
	3434	DB&I	WS2001/02	2
	...	...	...	...

(Integritäts-)Bedingungen

- **Strukturelle Bedingungen** beschränken in erster Linie die Struktur der Daten:
  - Typ-Bedingungen,
  - Schlüssel-Bedingungen,
  - Fremdschlüssel-Bedingungen.
- **Semantische Bedingungen** sind in erster Linie bedingt durch die konkreten Anwendungen auf der Datenbank.

strukturelle Bedingungen

- Die **Typ-Bedingungen** ergeben sich aus den Attributen und Wertebereichen eines Relationen-Schemas.
  Eine Instanz **erfüllt** die Typ-Bedingungen, wenn jedes Tupel der Instanz über der Menge der Attribute definiert ist und die Attributwerte aus den zugehörigen Wertebereichen sind.

Beispiel:

```
Professor(PersNr: INTEGER, Name: STRING, InstNr: INSTITUTE)
Belegung(MatrNr: INTEGER, KursBez: KURSE,
   Semester: SEMESTERS, Note: STRING)
```

`INSTITUTE, KURSE, SEMESTERS` sind benutzerdefinierte Wertebereiche. □

strukturelle Bedingungen (fortgesetzt)

- Eine **Schlüssel-Bedingung** definiert eine Teilmenge K der Menge der Attribute eines Relations-Schemas. K heißt **Schlüssel** (key).
  Eine Instanz **erfüllt** die Schlüsselbedingung, wenn sie keine von einander verschiedenen Tupel enthält, die bzgl. K dieselben Werte besitzen.
- Schlüssel sind minimale Mengen mit der die zulässigen Instanzen in der gewünschten Weise einschränkenden Eigenschaft.
- Existieren zu einer Relation mehrere Schlüssel, so bezeichnet man die Schlüssel als **Schlüsselkandidaten**. Ein ausgewählter Kandidat ist dann der **Primärschlüssel** der Relation.

Beispiel:

```
Professor(PersNr: INTEGER, Name: STRING, InstNr: INSTITUTE)
   KEY: {PersNr}
Belegung(MatrNr: INTEGER, KursBez: KURSE, Semester: SEMESTERS,
   Note: STRING) KEY: {MatrNr, KursBez, Semester}
```

□

strukturelle Bedingungen (fortgesetzt)

Eine **Fremdschlüssel-Bedingungen** bezieht sich auf zwei Relationen-Schemata R, S einer Datenbank. Sei K Schlüssel von R und sei K' Teilmenge der Attributmenge von S. Wir setzen eine bekannte Bijektion zwischen K und K' vor.

Eine Instanz einer Datenbank **erfüllt** die Fremdschlüsselbedingung "K' `REFERENCES` K", wenn für die entsprechenden Instanzen r und s die folgende Eigenschaft gilt:

> Zu jedem Tupel $\mu' \in s$ existiert ein Tupel $\mu \in r$, so dass μ und μ' bzgl. K und K' dieselben Werte haben.

K' heißt **Fremdschlüssel**.

Beispiel:

```
Belegung(MatrNr: INTEGER, KursBez: KURSE, Semester: SEMESTERS,
   Note: STRING) KEY: {MatrNr, KursBez, Semester}
(KursBez, Semester) REFERENCES Lehrangebot(KursBez, Semester)
```

□

semantische Bedingungen

Semantische Bedingungen definieren mittels logischer Ausdrücke die zulässigen Instanzen einer Relation. Eine Instanz **erfüllt** eine semantische Bedingung, wenn der zugehörige Ausdruck den Wahrheitswert `true` besitzt.

```
Belegung(MatrNr: INTEGER, KursBez: KURSE,
   Semester: SEMESTERS, Note: STRING)
   CHECK(Note IN ('A', 'B', 'C', 'D', 'F'))
   CHECK(MatrNr > 0 AND MatrNr < 100000000)
```

semantische Bedingungen (fortgesetzt)

Zur Formulierung eines semantischen Bedingung wird häufig auf die DML zurückgegriffen.

```
Professor(PersNr: INTEGER, Name: STRING, InstNr: INSTITUTE)
   CHECK (0 < (SELECT COUNT(*) FROM Student))
   CHECK ((SELECT COUNT(*) FROM Professor) <
      (SELECT COUNT(*) FROM Student))
```

Enthält die Bedingung einen Bezug auf eine andere Relation, so ist eine eigenständige Formulierung der Bedingung vorzuziehen.

```
ASSERTION
   CHECK ((SELECT COUNT(*) FROM Professor) <
      (SELECT COUNT(*) FROM Student))
```


Formalisierung der Begriffe

Attribut

Sei $X = \{A_1, \ldots, A_k\}$ eine endliche **Attributmenge**, $k \geq 1$.

Jedes $A \in X$ besitzt einen nicht-leeren **Wertebereich** dom(A).

Sei dom$(X) = \cup_{A \in X}$dom(A).

Beispiel:

$X = \{$`MatrNr, Name, Adresse, Semester`$\}$.

dom(`MatrNr`) = `INTEGER`, dom(`Semester`) = `INTEGER`,
dom(`Name`) = `STRING`, dom(`Adresse`) = `STRING`.

□

Tupel

Ein **Tupel** μ über X ist eine Abbildung

$$\mu : X \longrightarrow \text{dom}(X),$$

wobei $(\forall A \in X)\mu(A) \in$ dom(A).

Sei Tup(X) die Menge aller Tupel über X.

Beispiel:

$$\mu_1 = \{\texttt{MatrNr} \rightarrow 1223, \texttt{Name} \rightarrow \texttt{'HansEifrig'}, \texttt{Adresse} \rightarrow \texttt{'Seeweg20'}, \texttt{Semester} \rightarrow 2\}$$

$$\mu_2 = \{\texttt{MatrNr} \rightarrow 3434, \texttt{Name} \rightarrow \texttt{'LisaLustig'}, \texttt{Adresse} \rightarrow \texttt{'Bergstraße111'}, \texttt{Semester} \rightarrow 4\}$$

$$\mu_3 = \{\texttt{MatrNr} \rightarrow 1234, \texttt{Name} \rightarrow \texttt{'MariaGut'}, \texttt{Adresse} \rightarrow \texttt{'AmBächle1'}, \texttt{Semester} \rightarrow 2\}$$

□

Relation

Eine **Relation** r über X ist eine **endliche** Menge $r \subseteq$ Tup(X).

Sei Rel(X) die Menge aller Relationen über X.

Sei $r \in$ Rel(X). r ist ein **Instanz** zu X.

Beispiel:

MatrNr	Name	Adresse	Semester
1223	'HansEifrig'	'Seeweg20'	2
3434	'LisaLustig'	'Bergstraße111'	4
1234	'MariaGut'	'AmBächle1'	2

=

Name	MatrNr	Adresse	Semester
'HansEifrig'	1223	'Seeweg20'	2
'LisaLustig'	3434	'Bergstraße111'	4
'MariaGut'	1234	'AmBächle1'	2

□

Relation (fortgesetzt)

- Sei R ein **Relationsbezeichner**.

  Ein **(Relations)-Schema** zu R hat die Form $R(X)$.

  X ist hier eine endliche Attributmenge, das **Format** des Schemas.

- Anstatt $R(\{A_1, \ldots, A_k\})$ schreiben wir auch $R(A_1, \ldots, A_k)$; die so implizierte Ordnung der Attribute eines Formates ist unter Umständen von Bedeutung.

  k ist die **Stelligkeit** des Relationsbezeichners.

- Zu den Attributen kann auch der Domain angegeben werden.

$$R(A_1 : \mathsf{dom}(A_1), \ldots, A_k : \mathsf{dom}(A_k))$$

Datenbank

- Ein **(relationales) Datenbank-Schema** $\mathbf{R}$ ist gegeben durch eine (endliche) Menge von (Relations-) Schemata,

$$\mathbf{R} = \{R_1(X_1), \ldots, R_m(X_m)\},$$

  bzw. $\mathbf{R} = \{R_1, \ldots, R_m\}$.

- Eine **Instanz** $\mathcal{I}$ zu einem relationalen Datenbankschema $\mathbf{R} = \{R_1, \ldots, R_m\}$ ist eine Menge von Relationen $\mathcal{I} = \{r_1, \ldots, r_m\}$, wobei r_i Instanz zu R_i für $1 \leq i \leq m$.

  $\mathcal{I}$ betrachten wir bei Bedarf auch als Abbildung:

$$\mathcal{I}(R_i) = r_i, 1 \leq i \leq m.$$

Nullwerte

Tupel sind bisher **totale** Funktionen. Existiert zu einem Attribut bzgl. eines Tupels kein Wert, so kann ein **Nullwert** `null` verwendet werden.

Problematisch ist die Semantik eines Nullwertes: “Wert existiert, jedoch zur Zeit unbekannt” vs. “Wert existiert erst in der Zukunft” vs. “Wert prinzipiell unbekannt” vs. “Attribut nicht anwendbar” usw.

Beispiel:

MatrNr	Name	Adresse	Semester
1223	'HansEifrig'	null	2
3434	'LisaLustig'	'Bergstraße111'	4
1234	'MariaGut'	'AmBächle1'	null

□

Grundlagen von Anfragesprachen: Relationenalgebra

Was wollen wir können?

- Attribute aus Relationen herausstreichen: **Projektion**.
- Tupel aus Relationen auswählen: **Selektion**.
- Unterschiedliche Relationen miteinander verknüpfen: **Verbund**.
- Relationen wie Mengen verarbeiten: **Vereinigung, Differenz**.

Student	MatrNr	Name	Adresse	Semester
	1223	Hans Eifrig	Seeweg 20	2
	3434	Lisa Lustig	Bergstraße 111	4
	1234	Maria Gut	Am Bächle 1	2
	. . .	. . .	. . .	. . .

Projektion auf MatrNr und Name

Student	MatrNr	Name
	1223	Hans Eifrig
	3434	Lisa Lustig
	1234	Maria Gut
	. . .	. . .

Kurs	KursBez	InstNr	Name	Beschreibung
	DB	15-1	Datenbanken	Grundlagen von DB
	DB&I	15-1	Datenbanken und Internet	Internettechnologien, wie XML, ...
	MST	15-2	Mikrosystemtechnik	Einführung
	. . .	. . .	. . .	. . .

Selektion aller Tupel mit InstNr = 15-2

Kurs	KursBez	InstNr	Name	Beschreibung
	MST	15-2	Mikrosystemtechnik	Einführung

Student	MatrNr	Name
	1223	Hans Eifrig
	3434	Lisa Lustig
	1234	Maria Gut
	. . .	. . .

Verbund

Kurs	KursBez	Name
	DB	Datenbanken
	DB&I	Datenbanken und Internet
	MST	Mikrosystemtechnik
	. . .	. . .

Belegung	MatrNr	KursBez
	1223	DB
	3434	DB
	3434	DB&I
	. . .	. . .

StuKurs	MathNr	StudName	KursBez	KursName
	1223	Hans Eifrig	DB	Datenbanken
	3434	Lisa Lustig	DB	Datenbanken
	3434	Lisa Lustig	DB&I	Datenbanken und Internet

(A) Rechnen mit Relationen: Operationen auf Tupeln

Projektion

Sei $\emptyset \subset Y \subseteq X$, $X = \{A_1, \ldots, A_k\}$, und $\mu \in \mathsf{Tup}(X)$.

Der Ausdruck $\mu[Y]$ heißt **Projektion** von μ auf Y.

Es gilt:

$$\mu[Y] \in \mathsf{Tup}(Y)$$

wobei $\mu[Y](A) = \mu(A), A \in Y$.

Beispiel:

$$X = \{A, B, C, D\}.$$

$$\mu = (A \to a, B \to b, C \to c, D \to d)$$

$$\mu[B, D] = (B \to b, D \to d)$$

□

Selektion

Sei $A, B \in X$, $a \in \mathrm{dom}(A)$ und θ ein Vergleichsoperator aus $\{=, \neq, \leq, <, \geq, >\}$.

Eine (atomare) **Selektionsbedingung** α (bzgl. X) ist ein Ausdruck der Form $A\,\theta\,B$, bzw. $A\,\theta\,a$, bzw. $a\,\theta\,A$.

Ein Tupel $\mu \in \mathrm{Tup}(X)$ **erfüllt** eine Selektionsbedingung α, wenn gerade

$$\mu(A)\,\theta\,\mu(B), \text{ bzw. } \mu(A)\,\theta\,a, \text{ bzw. } a\,\theta\,\mu(A).$$

Atomare Selektionsbedingungen können mittels $\wedge$, $\vee$, $\neg$, $(,)$ zu Formeln verallgemeinert werden.

Beispiel:

$X = \{A, B, C\}$.
$\mu_1 = (A \to 2, B \to 2, C \to 1), \quad \mu_2 = (A \to 2, B \to 3, C \to 2)$
$\alpha_1 = (A = B), \quad \alpha_2 = ((B > 1) \wedge (C > 1))$

μ_1 erfüllt α_1, aber nicht α_2; μ_2 erfüllt α_2, aber nicht α_1. □

(B) Rechnen mit Relationen: Operationen auf Relationen

Seien X, Y Mengen von Attributen.

Vereinigung

Gelte $X = Y$ und seien weiter $r \subseteq \mathrm{Tup}(X), s \subseteq \mathrm{Tup}(Y)$.

$$r \cup s = \{\mu \in \mathrm{Tup}(X) \mid \mu \in r \vee \mu \in s\}.$$

Beispiel:

$$r = \begin{array}{|ccc|}\hline A & B & C \\ \hline a & b & c \\ d & a & f \\ c & b & d \\ \hline \end{array} \qquad s = \begin{array}{|ccc|}\hline A & B & C \\ \hline b & g & a \\ d & a & f \\ \hline \end{array} \qquad r \cup s = \begin{array}{|ccc|}\hline A & B & C \\ \hline a & b & c \\ d & a & f \\ c & b & d \\ b & g & a \\ \hline \end{array}$$

□

Differenz

Gelte $X = Y$ und seien weiter $r \subseteq \mathrm{Tup}(X), s \subseteq \mathrm{Tup}(Y)$.

$$r - s = \{\mu \in \mathrm{Tup}(X) \mid \mu \in r, \text{ wobei } \mu \notin s\}.$$

Beispiel:

$$r = \begin{array}{|ccc|}\hline A & B & C \\ \hline a & b & c \\ d & a & f \\ c & b & d \\ \hline \end{array} \qquad s = \begin{array}{|ccc|}\hline A & B & C \\ \hline b & g & a \\ d & a & f \\ \hline \end{array} \qquad r - s = \begin{array}{|ccc|}\hline A & B & C \\ \hline a & b & c \\ c & b & d \\ \hline \end{array}$$

□

Projektion

Sei $r \subseteq \mathrm{Tup}(X)$ und $Z \subseteq X$.

$$\pi[Z]r = \{\mu \in \mathrm{Tup}(Z) \mid \exists \mu' \in r, \text{ so dass } \mu = \mu'[Z]\}.$$

Beispiel:

$$r = \begin{array}{|ccc|}\hline A & B & C \\ \hline a & b & c \\ a & a & c \\ c & b & d \\ \hline \end{array} \qquad \pi[A, C](r) = \begin{array}{|cc|}\hline A & C \\ \hline a & c \\ c & d \\ \hline \end{array}$$

□

Selektion

Sei $r \subseteq \mathsf{Tup}(X)$ und α eine Selektionsbedingung zu X.

$$\sigma[\alpha]r = \{\mu \in \mathsf{Tup}(X) \mid \mu \in r \wedge \mu \text{ erfüllt } \alpha\}.$$

Beispiel:

$$r = \begin{array}{|ccc|}\hline A & B & C \\ \hline a & b & c \\ d & a & f \\ c & b & d \\ \hline \end{array} \qquad \sigma[B = b](r) = \begin{array}{|ccc|}\hline A & B & C \\ \hline a & b & c \\ c & b & d \\ \hline \end{array}$$

□

(natürlicher) Verbund (Join)

XY sei im folgenden eine Kurzschreibweise für $X \cup Y$.

Seien X, Y beliebig und seien weiter $r \subseteq \mathsf{Tup}(X), s \subseteq \mathsf{Tup}(Y)$.

$$r \bowtie s = \{\mu \in \mathsf{Tup}(XY) \mid \mu[X] \in r \wedge \mu[Y] \in s\}.$$

Beispiel:

$$r = \begin{array}{|ccc|}\hline A & B & C \\ \hline 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 6 \\ \hline \end{array} \quad s = \begin{array}{|cc|}\hline C & D \\ \hline 3 & 1 \\ 6 & 2 \\ 4 & 5 \\ \hline \end{array} \quad r \bowtie s = \begin{array}{|cccc|}\hline A & B & C & D \\ \hline 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 2 \\ 7 & 8 & 6 & 2 \\ \hline \end{array}$$

□

Umbenennung

Sei $X = \{A_1, \ldots, A_k\}$, $Y = \{B_1, \ldots, B_k\}$ und $\mathsf{dom}(A_i) = \mathsf{dom}(B_i)$, $1 \leq i \leq k$.

Sei $r \subseteq \mathsf{Tup}(X)$.

$$[X \to Y]r = \{\mu \in \mathsf{Tup}(Y) \mid \exists \mu' \in r, \text{ so dass } \mu(B_i) = \mu'(A_i), 1 \leq i \leq k\}$$

Beispiel: $X = \{A, B, C\}, Y = \{D, E, C\}$.

$$r = \begin{array}{|ccc|}\hline A & B & C \\ \hline a & b & c \\ d & a & f \\ c & b & d \\ \hline \end{array} \qquad [X \to Y]r = \begin{array}{|ccc|}\hline D & E & C \\ \hline a & b & c \\ d & a & f \\ c & b & d \\ \hline \end{array}$$

□

Basisoperatoren der Algebra

Vereinigung $\cup$

Differenz $-$

Projektion π

Selektion σ

Verbund $\bowtie$

Umbenennung $[\ldots]$

Weitere, mittels der Basisoperatoren ausdrückbare Operatoren

Sei $r_i \subseteq \mathsf{Tup}(X_i)$, $1 \leq i \leq n$.

Durchschnitt:

Sei $X_1 = X_2$.

$$r_1 \cap r_2 = r_1 - (r_1 - r_2).$$

Produkt:

Sei $X_1 \cap X_2 = \emptyset$.

$$r_1 \times r_2 = r_1 \bowtie r_2.$$

allgemeine natürliche Verbund:

$$\bowtie_{i=1}^{n} r_i = \{\mu \in \mathsf{Tup}(\cup_{i=1}^{n} X_i) \mid \mu[X_i] \in r_i\}.$$

θ-Verbund:

Sei $X_1 \cap X_2 = \emptyset$ und sei α eine beliebige Selektionsbedingung über $X_1 \cup X_2$.

$$r \bowtie_\alpha s = \sigma[\alpha](r \times s).$$

Enthält α ausschließlich Gleichheitsvergleiche, dann redet man von einem **Equi-Verbund**.

Division

Sei $Y \subset X$, $Z = X - Y$ und weiter $s \neq \emptyset$.

$$r \div s = \{\mu \in \mathsf{Tup}(Z) \mid \{\mu\} \times s \subseteq r\} =$$
$$\pi[Z]r - \pi[Z]((\pi[Z]r \times s) - r).$$

Beispiel:

$$r = \begin{array}{|cccc|}\hline A & B & C & D \\ \hline a & b & c & d \\ a & b & e & f \\ b & c & e & f \\ e & d & c & d \\ e & d & e & f \\ a & b & d & d \\ \hline \end{array} \qquad s = \begin{array}{|cc|}\hline C & D \\ \hline c & d \\ e & f \\ \hline \end{array} \qquad r \div s = \begin{array}{|cc|}\hline A & B \\ \hline a & b \\ e & d \\ \hline \end{array}$$

□

Division

Sei $Y \subset X$, $Z = X - Y$ und weiter $s \neq \emptyset$.

$$r \div s = \{\mu \in \mathsf{Tup}(Z) \mid \{\mu\} \times s \subseteq r\} =$$
$$\pi[Z]r - \pi[Z]((\pi[Z]r \times s) - r).$$

Beispiel:

$$r = \begin{array}{|cccc|}\hline A & B & C & D \\ \hline a & b & c & d \\ a & b & e & f \\ b & c & e & f \\ e & d & c & d \\ e & d & e & f \\ a & b & d & d \\ \hline \end{array} \qquad s = \begin{array}{|cc|}\hline C & D \\ \hline c & d \\ e & f \\ \hline \end{array} \qquad r \div s = \begin{array}{|cc|}\hline A & B \\ \hline a & b \\ e & d \\ \hline \end{array}$$

□

(C) Die Relationenalgebra als Anfragesprache

- Bisher haben wir Ausdrücke über Relationen, d.h. über einer Instanz eines Datenbank-Schemas betrachtet.
- Ersetzen wir in den Ausdrücken die Relationen durch entsprechende Relationsbezeichner, bzw. Konstante, so können wir den resultierenden Ausdruck als **Anfrage** betrachten.
- Jede Anfrage definiert zu einer gegebenen Instanz einer Datenbank eine Relation. Diese Relation ist die **Antwortmenge** zu der Anfrage bzgl. der Instanz.

  Sei Q eine Anfrage und $\mathcal{I}$ eine Instanz der Datenbank. Die Antwortmenge zu Q bzgl. $\mathcal{I}$ ist $\mathcal{I}(Q)$.

zulässige Ausdrücke der Relationenalgebra

Sei $\mathbf{R} = \{R_1(X_1), \ldots, R_k(X_k)\}$ eine Menge von Schemata. Sei $\mathcal{I}$ eine entsprechende Instanz, d.h. $\mathcal{I}(R_i) = r_i$, $r_i \subseteq \mathsf{Tup}(X_i)$.

Jeder Ausdruck Q der Relationenalgebra definiert eine Relation in Abhängigkeit einer Interpretation $\mathcal{I}$:

- Der Definitionsbereich von $\mathcal{I}$ wird auf die Menge der zulässigen Ausdrücke der Relationenalgebra ausgedehnt.
- Einem zulässigen Ausdruck Q wird ein Format X_Q zugeordnet.
- $\mathcal{I}$ weist Q eine Relation $\mathcal{I}(Q) \subseteq \mathsf{Tup}(X_Q)$ zu.

induktive Definition der zulässigen Ausdrücke und der durch sie definierten Relationen:

- Jeder **Relationsbezeichner** $R \in \mathbf{R}$ ist ein Ausdruck der Relationenalgebra. Das Format des Ausdrucks R und $\mathcal{I}(R)$ sind bereits definiert.
- Sei A ein Attribut und $a \in \mathsf{dom}(A)$ eine Konstante. Die **Relationskonstante** $\{a\}$ ist ein Ausdruck der Relationenalgebra. Das Format von $\{a\}$ ist $\{A\}$ und es gilt $\mathcal{I}(\{a\}) = \{a\}$.

Ausdrücke der Form R und $\{a\}$ sind die **atomaren** Ausdrücke der Relationenalgebra.

Seien dann $Q1, Q2$ Ausdrücke der Relationenalgebra, die jeweils Relationen $\mathcal{I}(Q1)$, $\mathcal{I}(Q2)$ definieren.

Die Menge der zulässigen Ausdrücke der Relationenalgebra ist die Menge aller Ausdrücke, die entweder atomar sind oder durch Anwendung der folgenden Regeln (1) bis (5) gebildet werden können.

Vereinigung: Gelte für die Formate $X_{Q1} = X_{Q2}$.
$Q = (Q1 \cup Q2)$ ist ein Ausdruck der Algebra. Es gilt $X_Q = X_{Q1}$ und $\mathcal{I}(Q) = \mathcal{I}(Q1) \cup \mathcal{I}(Q2)$.

Differenz: Gelte für die Formate $X_{Q1} = X_{Q2}$.
$Q = (Q1 - Q2)$ ist ein Ausdruck der Algebra. Es gilt $X_Q = X_{Q1}$ und $\mathcal{I}(Q) = \mathcal{I}(Q1) \setminus \mathcal{I}(Q2)$.

Projektion: Sei $\emptyset \subset Y \subseteq X_{Q1}$.
$Q = \pi[Y]Q1$ ist ein Ausdruck der Algebra. Es gilt $X_Q = Y$ und $\mathcal{I}(Q) = \pi[Y](\mathcal{I}(Q1))$.

Selektion: Sei α eine Selektionsbedingung zu X_{Q1}.
$Q = \sigma[\alpha]Q1$ ist ein Ausdruck der Algebra. Es gilt $X_Q = X_{Q1}$ und $\mathcal{I}(Q) = \sigma[\alpha](\mathcal{I}(Q1))$.

Verbund: $Q = (Q1 \bowtie Q2)$ ist ein Ausdruck der Algebra. Es gilt $X_Q = X_{Q1} \cup X_{Q2}$ und $\mathcal{I}(Q) = \mathcal{I}(Q1) \bowtie \mathcal{I}(Q2)$.

Umbenennung: Sei $X_{Q1} = \{A_1, \ldots, A_k\}$ und sei $\{B_1, \ldots, B_k\}$ eine Menge von Attributen.
$Q = [A_1 \to B_1, \ldots, A_k \to B_k]Q1$ ist ein Ausdruck der Algebra. Es gilt $X_Q = \{B_1, \ldots, B_k\}$ und $\mathcal{I}(Q) = \{\mu[A_1 \to B_1, \ldots, A_k \to B_k] \mid \mu \in \mathcal{I}(Q1)\}$.

- Eine Anfragesprache, die mindestens die Mächtigkeit der Algebra besitzt, heißt **relational vollständig**.
- Die Algebra ist nicht Turing-vollständig.
- Insbesondere existiert kein Ausdruck der Relationenalgebra, der für beliebige binäre Relationen r die transitive Hülle von r definiert.

Äquivalenz

Zwei Ausdrücke der Algebra Q, Q' heißen **äquivalent**, $Q \equiv Q'$, genau dann, wenn für jede Instanz $\mathcal{I}$ der Datenbank gilt: $\mathcal{I}(Q) = \mathcal{I}(Q')$.

Der Äquivalenzbegriff ist die Grundlage einer **algebraischen Optimierung**.

einige wichtige äquivalenzerhaltende Umformungen:

Sei attr(α) die in einer Selektionsbedingung α verwendete Menge von Attributen und seien $R, S, T \ldots$ Relationsbezeichner mit Formaten X, Y, Z.

- $Z_1 \subseteq Z_2 \subseteq X \Longrightarrow \pi[Z_1](\pi[Z_2]R) \equiv \pi[Z_1 \cap Z_2]R.$
- attr$(\alpha) \subseteq Z_1 \subseteq X \Longrightarrow \pi[Z_1](\sigma[\alpha]R) \equiv \sigma[\alpha](\pi[Z_1]R).$
- $R \bowtie S \equiv S \bowtie R.$
- $R \bowtie R \equiv R.$
- $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T).$
- attr$(\alpha) \subseteq X$, attr$(\alpha) \cap Y = \emptyset \Longrightarrow \sigma[\alpha](R \bowtie S) \equiv (\sigma[\alpha]R) \bowtie S.$

Grundlagen von Anfragesprachen: Relationenkalkül

- Der Relationenkalkül ist ein vom Prädikatenkalkül 1. Ordnung abgeleiteter Kalkül für relationale Datenbanken.
- Die Entwicklung eines solchen Kalküls ist naheliegend, da der einzige Strukturtyp relationaler Datenbanken Relationen sind und jede solche Relation als Instanz eines Prädikates betrachtet werden kann.
- Formate im Zusammenhang mit dem Kalkül sind immer geordnet, z.B. durch Indizierung oder durch die Reihenfolge des Hinschreibens der Attribute.

Sei dom im folgenden ein Wertebereich, der sich durch Vereinigung aller Wertebereiche der Attribute der einzelnen Formate der Schemata einer Datenbank $\mathbf{R}$ ergibt.

(A) Syntax

Die Formeln des Relationenkalküls (**R-Formeln**) werden aus Konstanten, Variablen, Relationsbezeichnern, Junktoren $\neg, \wedge, \vee$, Quantoren $\forall, \exists$ und Hilfszeichen "(", ")", "," analog zu Formeln des Prädikatenkalküls gebildet.

Beispiel: Seien Relationsschemata $R(A, B)$ und $S(B)$ gegeben.

$$\exists Y\, R(X, Y) \qquad R(X, Y) \wedge X = Y$$

$$R(X, Y) \wedge S(Y) \qquad R(X, Y) \vee (X = b \wedge S(Y))$$

$$R(X, Y) \wedge \neg(X = b \wedge S(Y)) \qquad \forall Y(S(Y) \Rightarrow R(X, Y))$$

$$\neg\exists Y(S(Y) \wedge \neg R(X, Y) \wedge \exists Z\ R(X, Z))$$

□

(1) Sei R ein Relationsbezeichner der Stelligkeit k. Seien weiter $a_1, \ldots, a_k$ Konstanten oder Variablen.

$R(a_1, \ldots, a_k)$ ist eine (atomare) R-Formel.

(2) Eine **Selektionsbedingung** ist von der Form $X\theta Y$, oder $X\theta a$, oder $a\theta X$, wobei X, Y Variablen, a eine Konstante und $\theta \in \{=, \neq, \leq, <, \geq, >\}$ ein Vergleichsoperator.

Jede Selektionsbedingung ist eine (atomare) R-Formel.

(3) Sei F eine R-Formel.

$\neg F$ ist eine R-Formel.

(4) Sei X eine Variable und F eine R-Formel, die die Variable X enthält, jedoch keinen Ausdruck der Form $\exists X$, bzw. $\forall X$. X heißt in diesem Fall **frei** in der R-Formel F und anderenfalls **gebunden** in F.

$\exists X\ F$ ist eine ($\exists$**-quantifizierte**) R-Formel.
$\forall X\ F$ ist eine ($\forall$**-quantifizierte**) R-Formel.

F ist der **Wirkungsbereich** des $\exists$-, bzw. $\forall$-Quantors.

(5) Seien F und G R-Formeln und sei $\mathcal{V}_F$, bzw. $\mathcal{V}_G$ die Menge der in F, bzw. G vorhandenen Variablen, wobei die Variablen in $\mathcal{V}_F \cap \mathcal{V}_G$ sowohl in F als auch in G frei.

Die **Konjunktion** $(F \wedge G)$ ist eine R-Formel.
Die **Disjunktion** $(F \vee G)$ ist eine R-Formel.

(B) Anfragen

Beispiel: Seien Relationsschemata $R(A, B)$ und $S(B)$ mit Instanzen r, s gegeben.

$$r = \begin{array}{|cc|}\hline A & B \\ \hline a & a \\ d & a \\ c & b \\ \hline \end{array} \qquad s = \begin{array}{|c|}\hline B \\ \hline b \\ c \\ \hline \end{array}$$

$$\exists Y R(X, Y) \quad \Longrightarrow \quad \begin{array}{|c|}\hline A \\ \hline a \\ d \\ c \\ \hline \end{array}$$

$$R(X, Y) \wedge X = Y \quad \Longrightarrow \quad \begin{array}{|cc|}\hline A & B \\ \hline a & a \\ \hline \end{array} \qquad R(X, Y) \wedge S(Y) \quad \Longrightarrow \quad \begin{array}{|cc|}\hline A & B \\ \hline c & b \\ \hline \end{array}$$

□

- Eine Anfrage Q über einem Datenbank-Schema $\mathbf{R}$ im Relationenkalkül, kurz **Kalkülanfrage**, hat die Form

$$\{(a_1, \ldots, a_n) \mid F\},$$

wobei F eine R-Formel zu $\mathbf{R}$ und $a_1, \ldots, a_n$ Variablen und Konstante.

Die Menge der Variablen unter den a_i ist gerade die Menge der freien Variablen von F.

- Soll ein Format $\{A_1, \ldots, A_n\}$ für die Menge der Antworten definiert werden, so schreiben wir

$$\{(A_1 : a_1, \ldots, A_n : a_n) \mid F\}.$$

(C) Antworten

Beispiel:

Seien Relationsschemata $R(A, B)$ und $S(B)$ gegeben.

Projektion $\pi[A]R$:	$\{X \mid \exists Y\, R(X, Y)\}$
Selektion $\sigma[A = B]R$:	$\{(X, Y) \mid R(X, Y) \wedge X = Y\}$
Verbund $R \bowtie S$:	$\{(X, Y) \mid R(X, Y) \wedge S(Y)\}$
Vereinigung $R \cup (\{1\} \times S)$:	$\{(X, Y) \mid R(X, Y) \vee (X = 1 \wedge S(Y))\}$
Differenz $R - (\{1\} \times S)$:	$\{(X, Y) \mid R(X, Y) \wedge \neg(X = 1 \wedge S(Y))\}$
Division $R \div S$:	$\{X \mid \forall Y(S(Y) \Rightarrow R(X, Y))\}$

□

Sei F eine R-Formel mit Variablenmenge $\mathcal{V}_F$. Eine **Variablenbelegung** ν zu F ist eine Funktion:

$$\nu : \mathcal{V}_F \rightarrow \text{dom}.$$

ν wird erweitert um die Identität für Konstante.

Seien $a_1, \ldots, a_n$ Variablen und Konstante. Sei $Q = \{(a_1, \ldots, a_n) \mid F\}$.

Die **Antwortmenge** zu Q bzgl. $\mathcal{I}$ ist

$$Q(\mathcal{I}) = \{(\nu(a_1), \ldots, \nu(a_n)) \mid \nu \text{ ist eine Belegung der freien Variablen in } F, \text{ so dass } F \text{ unter } \nu \text{ wahr bzgl. } \mathcal{I}\}.$$

(D) Wertebereichsunabhängigkeit

Beispiel:

(a) Sei Q eine Anfrage der Form

$$\{X \mid \neg R(X)\},$$

wobei $\mathcal{I}(R) = \{1\}$.

$Q(\mathcal{I})$ ist nicht endlich!

(b) Sei Q eine Anfrage der Form

$$\{(X, Z) \mid \exists Y(R(X, Y) \vee S(Y, Z))\},$$

wobei $\mathcal{I}(R) = \{(1, 1)\}$, bzw. $\mathcal{I}(S) = \emptyset$.

$Q(\mathcal{I})$ ist nicht endlich!

□

Forderung: Das Resultat einer Anfrage darf nur von den Konstanten in der Anfrage und den Konstanten in den Relationen der betrachteten Instanz abhängen.

Sei $Q := \{(a_1, \ldots, a_n) \mid F\}$. Sei $\mathcal{I}$ eine Instanz zu $\mathbf{R}$ und adom diejenige Menge, die gerade alle Konstanten in Q und alle Konstanten aus $\mathcal{I}$ enthält.

adom ist der **aktive** Wertebereich von Q; adom ist endlich.

Q, bzw. F, heißt **wertebereichsunabhängig**, wenn für jede beliebige Menge $D \supset$ adom gilt:

$$Q(\mathcal{I}, \text{adom}) = Q(\mathcal{I}, D).$$

Sichere Formeln

Eine R-Formel F heißt **(syntaktisch) sicher** genau dann, wenn sie die folgenden Bedingungen erfüllt:

(1) F enthält keine $\forall$-Quantoren.

(2) Wenn $F_1 \vee F_2$ Teilformel von F, dann müssen F_1 und F_2 dieselben freien Variablen besitzen.

Eine Teilformel G einer Formel F heißt **maximal konjunktiv**, wenn F keine konjunktive Teilformel der Form $H \wedge G$ oder $G \wedge H$ enthält.

(3) Sei $F_1 \wedge \ldots \wedge F_m$, $m \geq 1$, eine maximal konjunktive Teilformel von F.

Alle freien Variablen müssen **begrenzt** sein im folgenden Sinn:

Sei $1 \leq j \leq m$.

(a) Eine Variable ist begrenzt, wenn sie in einer Formel F_j frei ist, wobei F_j weder ein Vergleichsausdruck noch negiert ist.

(b) Falls F_j die Form $X = a$ oder $a = X$ hat, a eine Konstante, dann ist X begrenzt.

(c) Falls F_j die Form $X = Y$ oder $Y = X$ hat und Y ist begrenzt, dann ist auch X begrenzt.

Beispiel

- $X = Y \vee R(X, Y)$ ist nicht sicher
- $X = Y \wedge R(X, Y)$ ist sicher
- $R(X, Y, Z) \wedge \neg(S(X, Y) \vee T(Y, Z))$ ist nicht sicher, jedoch in der äquivalenten Schreibweise
$$R(X, Y, Z) \wedge \neg S(X, Y) \wedge \neg T(Y, Z)$$
- Die Formel (Division!) $\{X \mid \forall Y(S(Y) \Rightarrow R(X, Y))\}$ ist äquivalent zu
$$\{X \mid \neg\exists Y(S(Y) \wedge \neg R(X, Y))\}$$
Beide Formulierungen sind nicht sicher, jedoch die äquivalente Formulierung
$$\{X \mid \neg\exists Y(S(Y) \wedge \neg R(X, Y) \wedge \exists Z\, R(X, Z)) \ \wedge\ \exists U \exists V\, R(U, V) \ \wedge\ X = U\}$$
ist sicher. □

Sicherheit einer Formel ist somit rein syntaktisch definiert!

zur sicheren Variante der Division:

$$\{X \mid \neg\exists Y(S(Y) \wedge \neg R(X, Y) \wedge \exists Z\, R(X, Z)) \ \wedge\ \exists U \exists V\, R(U, V) \ \wedge\ X = U\}$$

Es müssen alle maximal konjunktiven Teilformeln betrachtet werden.

$$R(X, Z), \qquad R(U, V), \qquad \exists V\, R(U, V)$$

$$S(Y) \wedge \neg R(X, Y) \wedge \exists Z\, R(X, Z)$$

$$\exists Y(S(Y) \wedge \neg R(X, Y) \wedge \exists Z\, R(X, Z))$$

$$\neg\exists Y(S(Y) \wedge \neg R(X, Y) \wedge \exists Z\, R(X, Z)) \ \wedge\ \exists U \exists V\, R(U, V) \ \wedge\ X = U$$

(E) Äquivalenz von Algebra und sicherem Kalkül

Satz: Zu jedem Ausdruck Q der Relationenalgebra existiert eine äquivalente sichere Anfrage Q' des Relationenkalküls und umgekehrt; d.h. Q und Q' definieren für jede beliebige Datenbank-Instanz $\mathcal{I}$ dieselbe Relation. □

Der SQL-Standard

SQL: Structured (Standard) Query Language

Literatur: A Guide to the SQL Standard, 3rd Edition, C.J. Date and H. Darwen, Addison-Wesley 1993

Grundstruktur von Anfragen in SQL:

`SELECT` $A_1, \ldots, A_n$	(... entspricht π in der Algebra)
`FROM` $R_1, \ldots, R_m$	(... gibt die betreffenden Relationen an)
`WHERE` F	(... entspricht σ in der Algebra)

... ist äquivalent zu dem Algebraausdruck: $\pi[A_1, \ldots, A_n](\sigma[F](r_1 \times \ldots \times r_m))$

- SQL-Relationen sind **Multimengen (bags)** von Tupeln, können also Tupel mehrmals enthalten.
- Begriffe: Relation $\rightsquigarrow$ Tabelle (table)
  Tupel $\rightsquigarrow$ Zeile (row)
  Attribut $\rightsquigarrow$ Spalte (column)

Historie: Anfänge ca. 1974 als SEQUEL (IBM, System R)

SQL-86 und **SQL-89:** Schnittmenge existierender Implementationen

SQL-92 (SQL2):

- expliziter Verbund
- Integritätsbedingungen
- referentielle Integrität ...

SQL-99 (SQL3):

- aktive Regeln
- Rekursion
- objektorientierte Konzepte ...

Datenbank-Schema für die folgenden Beispiele

City

name	*country*	*population*	*longitude*	*latitude*
Rome	I	2791354	12,6	41,8
Bagdad	IRQ	3200000	44,3	33,2
Er_Riad	KSA	1250000	47	25
Beirut	RL	702000	35,31	33,55
Kabul	AFG	892000	69,5	34,35
Warsaw	PL	1655000	21	52,21
Vienna	A	158300	16,36	48,25
Trient	I	244997	-95,5	33,6
Paris	USA	98700	-76,5	37,2
...	...	...	...	...

Country

name	*code*	*pop. in Mio.*	*area*
Zambia	Z	5.8	752614
Grenada	WG	0.1	344
Syria	SYR	12	182000
Venezuela	YV	18	912050
United States	USA	242	9363000
Lichtenstein	FL	0.02	160
Luxembourg	L	0.3	2586
France	F	54	547026
Seychelles	SY	0.07	455
...	...	...	...

encompasses

country	*continent*	*percentage*
TR	Asia	68
TR	Europe	32
E	Europe	100
IND	Asia	100
ET	Africa	90
...	...	...

is_member

country	*organization*	*type*
UAE	OPEC	member
B	EU	member
IS	WEU	associate member
B	NATO	member
TCH	OAU	member
...	...	...

SQL: einfache Anfragen

Welche Städtenamen sind in der Relation `City` gespeichert?

```
SELECT name
   FROM City;
```

. . . mit Eliminierung von Duplikaten:

```
SELECT DISTINCT name
   FROM City;
```

. . . mit `WHERE`-Klausel:

```
SELECT DISTINCT name
   FROM City
   WHERE population > 1000000;
```

. . . mit abkürzender Schreibweise für *alle* Spalten:

```
SELECT *
   FROM City
   WHERE population > 1000000;
```


Teilstring-Vergleich und Sortierung

Erstelle eine Liste der Länder, die den Begriff Bund *in ihrem Namen tragen.*

```
SELECT *
   FROM Country
   WHERE Name LIKE '%Bund%';
```

Erstelle eine Liste der Länder, deren dritter Buchstabe im Attribut `code`, ein "D" ist.

```
SELECT *
   FROM Country
   WHERE code LIKE '_D%';
```

Erstelle eine Liste aller Länder, aufsteigend bzgl. Fläche und absteigend bzgl. Einwohner.

```
SELECT *
   FROM Country
   ORDER BY area ASC, population DESC;
```


Datentypen Date und Time

Beispiel Datum DATE '1978-05-28' mit Uhrzeit TIME '16:07:44.6'.

Arithmetische Vergleichsoperationen können verwendet werden.

skalare Teilanfragen

Eine geklammerte Teilanfrage kann wie eine skalare Konstante verwendet werden; sinnvoll ist dies, wenn aufgrund von Zusatzinformationen (Schlüssel) gesichert ist, daß die Teilanfrage in der Tat nur genau einen Wert liefert.

Welche Länder haben einen geringeren Anteil ihrer Fläche in Asien als die Türkei?

```
SELECT DISTINCT country, percentage
   FROM encompasses
   WHERE continent = 'Asia' AND
      percentage <
         (SELECT percentage
            FROM encompasses
            WHERE country = 'TR' AND continent = 'Asia');
```

Anfragen mit mehreren Relationen

explizite Spaltenreferenzierung, bzw. Tupelvariable/Korrelationsvariable/Alias

Bestimme alle Städte eines Landes.

```
SELECT DISTINCT City.name, Country.name
   FROM City, Country
   WHERE City.country = Country.code;
```


... unter Verwendung eines Alias für die betreffenden Relationen.

```
SELECT DISTINCT S.name, L.name
   FROM City S, Country L
   WHERE S.country = L.code;
```

... die Verwendung des Schlüsselwortes `AS` bei der Benennung eines Alias ist optional.

```
SELECT DISTINCT S.name, L.name
   FROM City AS S, Country AS L
   WHERE S.country = L.code;
```

Bestimme alle Paare von Ländern, die im selben Kontinent liegen (die Verwendung eines Alias ist notwendig).

```
SELECT E1.country first, E2.country second
   FROM encompasses E1, encompasses E2
   WHERE E1.continent = E2.continent
      AND E1.country < E2.country;
```


zur Semantik:

Der folgende Ausdruck zur Berechnung von $R \cap (S \cup T)$ liefert $\emptyset$, sofern $T = \emptyset$!

```
SELECT R.A FROM R, S, T WHERE R.A = S.A OR R.A = T.A
```

□

- **SQL-Anfrage:**

  `SELECT` $A_1, \ldots, A_n$ `FROM` R_1 `AS` t_1`, . . . ,` R_m `AS` t_m `WHERE` F

- **äquivalenter Ausdruck der relationalen Algebra:**

  $$\pi[A_1, \ldots, A_n](\sigma[F](r_1 \times \ldots \times r_m))$$

- **Algorithmus (nested-loop-Semantik):**
  FOR each tuple t_1 in relation R_1 DO
  FOR each tuple t_2 in relation R_2 DO
  . . .
  FOR each tuple t_n in relation R_n DO
  IF die WHERE-Klausel ist erfüllt nach Ersetzen der Attributnamen durch die entsprechenden Werte von $t_1, \ldots, t_n$ THEN
  werte die SELECT-Klausel bzgl. $t_1, \ldots, t_n$ aus und bilde das Antwort-Tupel.

- **Kalkül-Semantik:**
  Betrachte alle möglichen Zuweisungen von Tupeln aus den Relationen an die Tupelvariablen und für jede solche Zuweisung, bzgl. der die WHERE-Klausel erfüllt ist, bilde das Antwort-Tupel.

Schachtelung von Anfragen (Teilanfragen)

Welche Länder befinden sich auf einem Kontinent wie Russland?

```
SELECT DISTINCT country
   FROM encompasses
   WHERE continent IN
      (SELECT continent
         FROM encompasses
         WHERE country = 'RU');
```

Welche Länder haben eine größere Einwohnerzahl als ein/alle Länder in Europa?

```
SELECT name
   FROM Country
   WHERE population > ANY (ALL)
      (SELECT population
         FROM Country, encompasses
         WHERE Country.code = encompasses.country AND
            encompasses.continent = 'Europe');
```


SQL: Tabellenausdrücke und Aggregierungen

allgemeine Struktur eines SQL-Ausdrucks:

SELECT $A_1, \ldots, A_n$	Liste der Attribute
FROM $R_1, \ldots, R_m$	Liste der Relationen
WHERE F	Bedingung
GROUP BY $B_1, \ldots, B_k$	Liste der Gruppierungsattribute
HAVING G	Gruppierungsbedingung
ORDER BY H	Sortierordnung

Auswertungsreihenfolge:

FROM-Klausel vor WHERE-Klausel vor

GROUP-Klausel vor HAVING-Klausel vor

ORDER-Klausel vor SELECT-Klausel.

Bemerkungen

- Ausdrücke auf Tabellen können mittels der Operatoren UNION, EXCEPT, INTERSECT, JOIN und durch Schachtelung von SELECT gebildet werden.
- In der WHERE (und HAVING-Klausel) können Tupel mittels der üblichen arithmetischen Vergleichsoperatoren und den Operatoren NOT, IN, ANY, ALL, EXISTS und UNIQUE für die weitere Betrachtung qualifiziert, bzw. disqualifiziert werden.

- Tests auf Teilmengen, bzw. Mengengleichheit können nicht direkt ausgedrückt werden. Zwei Möglichkeiten für einen Gleichheitstest:

  1. Zwei Mengen sind gleich, wenn die Vereinigung der Mengen minus dem Schnitt der Mengen leer ist.
  2. Zwei Mengen sind gleich, wenn jedes Element der ersten Menge auch Element der zweiten Menge ist und umgekehrt.

- Mittels EXISTS kann eine Menge getestet werden, ob sie Elemente enthält. NOT EXISTS ergibt dann einen Test auf die leere Menge.

  Wegen $\forall X F$ äquivalent zu $\neg \exists X \neg F$ kann EXISTS auch verwendet werden, um alle Elemente einer Menge zu testen.

Mengenoperationen

Welche Länder sind Teil von Europa und von Asien?

```
(SELECT country
   FROM encompasses
   WHERE continent = 'Europe')
INTERSECT
(SELECT country
   FROM encompasses
   WHERE continent = 'Asia');
```


Welche Land-ID's treten in der Relation `Country` *oder (und nicht in) der Relation* `encompasses` *auf?*

```
(SELECT code                (SELECT code
   FROM Country)               FROM Country)
UNION                       EXCEPT
(SELECT country             (SELECT country
   FROM encompasses);          FROM encompasses);
```

Mittels UNION ALL, INTERSECTION ALL, EXCEPT ALL werden Duplikate der Zeilen berücksichtigt.

Hat der erste Operand n Duplikate eines Tupels und der zweite Operand m, wobei $0 \leq n, m$, dann hat das Ergebnis $n+m$, $min(n, m)$, $max(n-m, 0)$ Duplikate dieses Tupels.

Teilanfragen mit Korrelationsvariablen

Bei Verwendung von Korrelationsvariablen von übergeordneten Anfragen wird die Teilanfrage pro Wertekombination der Korrelationsvariablen einmal ausgeführt.

Mitglied in denselben Organisationen wie Andorra? (Division!)

```
SELECT DISTINCT country
   FROM is_member M
   WHERE NOT EXISTS
      ((SELECT DISTINCT organization FROM is_member
         WHERE country = 'AND')
      EXCEPT
      (SELECT DISTINCT organization FROM is_member
         WHERE country = M.country));
```


Welche Organisationen haben alle europäischen Länder als Mitglieder und nur gerade diese? (Mengengleichheit!)

```
(SELECT DISTINCT organization
   FROM is_member im1
   WHERE (NOT EXISTS
      (SELECT Country
         FROM encompasses
         WHERE encompasses.continent = 'Europe') EXCEPT
      (SELECT im2.Country
         FROM is_member im2
         WHERE im2.organization = im1.organization))
AND (NOT EXISTS
      (SELECT im2.Country
         FROM is_member im2
         WHERE im2.organization = im1.organization) EXCEPT
      (SELECT Country
         FROM encompasses
         WHERE encompasses.continent = 'Europe')));
```

Aggregierungsfunktionen

Aggregierungsfunktionen beziehen sich jeweils auf eine Spalte einer Relation; es existieren die Operatoren SUM, AVG, MIN, MAX, COUNT.

Welches ist die größte Landesfläche?

```
SELECT MAX(area)
   FROM Country;
```

Wieviele Länder sind aufgeführt?

```
SELECT COUNT(*)
   FROM Country;
```

Wieviele Organisationen sind aufgeführt?

```
SELECT COUNT(DISTINCT organization)
   FROM is_member;
```

Bemerkung: Schachtelung von Aggregierungsfunktionen wie z.B. MAX(AVG(. . .)) ist gemäß dem Standard nicht erlaubt.

Gruppierung der Zeilen einer Tabelle

Wie hoch sind die durchschnittlichen Einwohnerzahlen in den Städten der einzelnen Länder?

```
SELECT country, AVG(population)
   FROM City
   GROUP BY country;
```

Bei einer gruppierten Tabelle werden Zeilen mit gleichen GROUPING-Spalten zu einer Gruppe zusammengefaßt.

Für jede Gruppe von Zeilen mit gleichen Gruppierungswerten werden die Aggregierungsfunktionen ausgewertet.

In der SELECT-Klausel dürfen nur diejenigen Attribute nicht aggregiert auftauchen, die zu einer Gruppierungsspalte gehören.

Einschränkungen der Gruppen mit der `HAVING`-Klausel

In welchen Ländern ist die durchschnittliche Einwohnerzahl in den Städten kleiner als 0.1?

```
SELECT country, AVG(City.population)
   FROM City
   GROUP BY country
   HAVING AVG(population) < 0.1;
```


Welche sind die drei Länder mit der höchsten Einwohnerzahl?

ALternative 1:

```
SELECT MAX(A.population), MAX(B.population),
       MAX(C.population)
   FROM Country A, Country B, Country C
   WHERE (A.population > B.population)
      AND (B.population > C.population);
```

Alternative 2:

```
SELECT DISTINCT COUNT(*), A.population
   FROM Country A, Country B
   WHERE (A.population <= B.population)
      GROUP BY A.population
      HAVING COUNT(*) <= 3;
```

Was sind die Unterschiede der Ergebnisse der beiden Varianten?

Orthogonalität

Ein Ausdruck, der eine Menge definiert, sollte überall dort zulässig sein, wo ein Relationsbezeichner stehen darf:

Berechne die Gesamtzahl aller Städte und Länder.

```
SELECT COUNT(*)
   FROM ((SELECT name
             FROM City) UNION
         (SELECT name
             FROM Country));
```


Berechne die Anzahl der Menschen, die in der größten Stadt ihres Landes leben.

```
SELECT SUM(pop_biggest)
   FROM (SELECT country, MAX(population) AS pop_biggest
            FROM City
            GROUP BY country);
```

Bemerkung: Hier wird das Verbot der Schachtelung von Aggregierungsfunktionen umgangen.

Ein Ausdruck, der einen skalaren Wert definiert, sollte überall dort zulässig sein, wo ein solcher stehen darf:

```
SELECT name, population,
      (SELECT AVG(population) FROM City C1
         WHERE C1.population <= C0.population) AS interesting)
   FROM City C0;
```


Exkurs: von Algebra/Kalkül zu SQL am Beispiel der Division

zur Erinnerung: die Anfrage *Welche Länder sind Mitglied in mindestens denselben Organisationen wie Andorra?* kann mittels Division gelöst werden.

```
SELECT DISTINCT country
   FROM is_member M
   WHERE NOT EXISTS
      ((SELECT DISTINCT organization FROM is_member
         WHERE country = 'AND')
       EXCEPT
       (SELECT DISTINCT organization FROM is_member
         WHERE country = M.country));
```

nur, wo ist hier die Division?

(A) Ausgangspunkt Algebra

Relationsschemata $R(A, B), S(B)$ und $T(A)$ seien gegeben.

$$
\begin{gathered}
T = R \div S \\
= \\
\{X \mid \{X\} \times S \subseteq R\} \\
= \\
\{X \mid \{X\} \times S - R = \emptyset\} \\
= \\
\{X \mid (S - \pi[B](\sigma[A = X]R)) = \emptyset\}
\end{gathered}
$$

kann in einen SQL-Ausdruck transformiert werden:

```
SELECT DISTINCT A FROM R as X
   WHERE NOT EXISTS
      ((SELECT DISTINCT B FROM S)
      EXCEPT
      (SELECT DISTINCT B FROM R
         WHERE R.A = X.A));
```

(B) Ausgangspunkt Kalkül

Relationsschemata $R(A, B), S(B)$ und $T(A)$ seien gegeben.

$$T = R \div S$$
$$=$$
$$\{X \mid \forall Y(S(Y) \Rightarrow R(X, Y))\}$$
$$=$$
$$\{X \mid \neg\exists Y(S(Y) \wedge \neg R(X, Y))\}$$

kann nach SQL transformiert werden wie folgt:

```
SELECT DISTINCT A FROM R as X
   WHERE NOT EXISTS
      (SELECT * FROM S as Y
        WHERE NOT EXISTS
         (SELECT * FROM R
           WHERE R.A = X.A
            AND R.B = Y.B));
```

```
SELECT DISTINCT A FROM R as X
   WHERE NOT EXISTS
      (SELECT * FROM S as Y
        WHERE X.A NOT IN
         (SELECT A FROM R
            WHERE R.B = Y.B));
```


SQL: Erstellen, Löschen, Einfügen, Ändern und Sichten

Erstellen

In der einfachsten Form besteht eine Tabellendefinition nur aus der Angabe der Attribute sowie ihres Wertebereichs.

```
CREATE TABLE <table>
   (<col> <datatype>,
    :
    <col> <datatype>);
```

```
CREATE TABLE City
      ( name          VARCHAR2(35),
        country       VARCHAR2(4),
        population    NUMBER,
        longitude     NUMBER,
        latitude      NUMBER );
```


Allgemeine Form:

In einer Tabellendefinition können desweiteren Eigenschaften und Bedingungen an die jeweiligen Attributwerte formuliert werden.

Beispiel:

```
CREATE TABLE Country
  ( name         VARCHAR2(32) NOT NULL UNIQUE,
    code         VARCHAR2(4) PRIMARY KEY,
    ...          ...);
```

```
CREATE TABLE City
      ( name          VARCHAR2(35),
        country       VARCHAR2(4) DEFAULT 'D',
        population    NUMBER,
        longitude     NUMBER,
        latitude      NUMBER )
        PRIMARY KEY (longitude, latititude);
```


Löschen

```
DELETE R WHERE P;
```

Lösche die Relation City.

```
DELETE City;
```

Löschen aller Städte, deren Einwohnerzahl kleiner dem Durchschnitt sind.

```
DELETE City
   WHERE population <
      (SELECT AVG(population) FROM City);
```

Bemerkung: während des Löschens der einzelnen Tupel ändert sich im Prinzip der Durchschnitt! In SQL-Implementierungen werden deshalb vor dem Löschen alle zu löschenden Tupel markiert und danach gelöscht.

Einfügen

Einfügen eines Tupels.

```
INSERT INTO R(A_1, ..., A_n) VALUES (val_1,...,val_n)
```

Die Werte des einzufügenden Tupels müssen in der Reihenfolge der entsprechenden Attribute stehen.

Werden nicht zu allen Attributen von `R` Werte angegeben, so werden für die fehlenden Werte die vorgesehenen Default-Werte, bzw. der Nullwert `NULL` genommen.

Werden zu allen Attributen Werte angegeben, so kann auf die Auflistung der Attribute verzichtet werden.

Einfügen eines neuen Mitgliedes in die EU.

```
INSERT INTO is_member
   VALUES ('PL', 'EU', 'member');
```


Einfügen einer Menge von Tupeln.

Alle Länder die Mitglied in bestimmten Organisationen sind, die aber noch nicht in der Relation `Country` *auftreten, werden in diese Relation übernommen.*

```
INSERT INTO Country
   SELECT DISTINCT is_member.country
      FROM is_member
      WHERE NOT EXISTS (
         SELECT code
            FROM Country L
            WHERE L.code = is_member.country);
```


Ändern

Einzelne Werte eines Tupels können geändert werden.

```
UPDATE R
   SET K
   WHERE P ;
```

Im Zuge der Euro-Umstellung, werden bestimmte Angaben, die sich auf nationale Währungen beziehen wie etwa das Bruttosozialprodukt (BSP), angepasst.

```
UPDATE Country
   SET BSP=BSP*0,5;
   WHERE code = 'D';
```


Nullwerte

Was bedeutet NULL: Wert unbekannt? oder Attribut nicht anwendbar?

Mit `IS NULL, IS NOT NULL` kann auf Existenz von Nullwerten geprüft werden.

- In skalaren Ausdrücken (A+B, A+1 etc.) ist das Resultat NULL, wenn einer der Operanden NULL ist.
- Aggregierungsfunktionen ignorieren NULL in ihren Argumenten; Ausnahme COUNT(*).
- Ausdrücke mit Vergleichsoperatoren der Form A=B, A<>B, A<B, etc. haben den Wahrheitswert UNKNOWN, wenn mindestens einer der beteiligten Operanden den Wert NULL besitzt.

Dreiwertige Logik (t = TRUE, f = FALSE, u = UNKNOWN):

AND	t	u	f
t	t	u	f
u	u	u	f
f	f	f	f

OR	t	u	f
t	t	t	t
u	t	u	u
f	t	u	f

NOT	
t	f
u	u
f	t

SQL: Sichten

Benutzer(Gruppen) bekommen eine individuelle **Sicht** auf die DB zugewiesen.

Eine Sicht (view) ist eine Relation definiert mittels der Anfragesprache, hier SQL, bzgl. der in der Datenbank vorhandenen **Basisrelationen**.

```
CREATE VIEW V AS
   <E>
   [ WITH CHECK OPTION ]
```

wobei <E> ein beliebiger SQL-Anfrageausdruck.

Sichten dürfen in SQL überall stehen, wo ein Relationsname stehen darf.

virtuelle Sichten:

Soll eine Anfrage bearbeitet werden, die sich auf eine Sicht bezieht, so wird vor ihrer Ausführung der Name der Sicht durch den sie definierenden Ausdruck ersetzt (**query modification**).

Nachteil: hoher Berechnungsaufwand; dieser kann durch einen guten Optimierer verringert werden.
Vorteil: Änderungen in der DB werden nicht beeinträchtigt.

materialisierte Sichten:

Sichten werden einmal berechnet und dann permanent gehalten (**view materialization**). Ändern sich die Basisrelationen, d.h. die im Schema definierten Relationen, so werden die Änderungen der Sicht (möglichst inkrementell) nachgezogen.

Nachteil: Hohe Aufwand bei Änderungen in den Basisrelationen, großer Speicherbedarf und Bedarf an zusätzlichen Indexstrukturen.
Vorteil: Schneller Zugriff auf die Sichten möglich.

Definition von Sichten

Sicht `BigCountry`.

```
CREATE VIEW BigCountry AS
   (SELECT *
      FROM Country
      WHERE area >= 1000000 AND
         population >= 100);
```

Anfrage auf Sicht `BigCountry`.

```
SELECT *
   FROM BigCountry;
```

Sicht: Sachbearbeiter dürfen den Namen nicht sehen.

```
CREATE VIEW CountryInfo AS
   SELECT code, area, population
      FROM Country;
```


Einfügen, Löschen und Ändern in Sichten

- Sei $\mathbf{R}$ ein Datenbank-Schema.
- Eine **Datenbank-Änderung** ist eine Funktion t von der Menge der Instanzen zu $\mathbf{R}$ auf sich selbst.
- Eine **Sicht** ist ein Relations-Schema S; sei f eine Abbildung von der Menge aller Instanzen zu $\mathbf{R}$ in die Menge aller Instanzen zu S.
- Eine **Sicht-Änderung** ist eine Funktion u von der Menge aller Instanzen zu S auf sich selbst.
- Sei u eine Sicht-Änderung und t eine Datenbank-Änderung, so daß für jede Datenbank-Instanz $\mathcal{I}$ gilt:

  $$u(f(\mathcal{I})) = f(t(\mathcal{I})),$$

  dann nennen wir t eine **Transformation (Übersetzung)** von u.

Auf einer Sicht sind nur solche Änderungen zulässig, zu denen eine Transformation existiert.

Beispiele für nicht existierende Transformationen:

$$r = \begin{array}{|cc|}\hline A & B \\ \hline a & b \\ x & b \\ \hline \end{array} \qquad s = \begin{array}{|cc|}\hline B & C \\ \hline b & c \\ b & z \\ \hline \end{array} \qquad v = r \bowtie s = \begin{array}{|ccc|}\hline A & B & C \\ \hline a & b & c \\ a & b & z \\ x & b & c \\ x & b & z \\ \hline \end{array}$$

Auf v sind folgende Löschungen, Einfügungen und Änderungen nicht möglich.

- **Löschen** von (a, b, c):
  Wird in r (a, b) gelöscht, so geht in v auch (a, b, z) verloren; analog für Löschen von (b, c) in s.
- **Ändern** von (a, b, c) zu (a, b, d) analog.
- **Einfügen** von (a, b, d) analog.

Beispiel Projektionssicht:

Sicht: Sachbearbeiter dürfen den Namen nicht sehen.

```
CREATE VIEW CountryInfo AS
   SELECT code, area, population
      FROM Country;

INSERT INTO CountryInfo VALUES ('XY', 250000, 20);
```

Das Einfügen muß auf der Relation `Country` nachgeführt werden.

Jedoch welcher Wert? Für herausprojizierte Attribute können Default-Werte, bzw. `NULL` verwendet werden.

Projektionssichten sind wenig problematisch, solange die Primärschlüssel der Basisrelationen in der Sicht nicht herausprojiziert werden.

Beispiel Selektionssicht:

Sicht auf die Big Countries*.*

```
CREATE VIEW BigCountry AS
   (SELECT *
      FROM Country
      WHERE area >= 1000000 AND
         population >= 100);
```

```
UPDATE BigCountry
   SET population = 90
   WHERE code = '...';
```

Tupel der Sicht `BigCountry` **migrieren** möglicherweise in eine andere Sicht. Durch Angabe der Klausel `WITH CHECK OPTION` kann dies verhindert werden.

Beispiel Verbundsicht:

Sicht: Kontinente und die in ihnen liegenden Städte.

```
CREATE VIEW CityInfo AS
   SELECT City.name, encompasses.continent
      FROM City, encompasses
      WHERE City.country = encompasses.country;

INSERT INTO CityInfo VALUES ('Freiburg', 'Europe');
```

Einfügungen in die Basisrelationen:

`City`: `('Freiburg', NULL, NULL, NULL,NULL)`
`encompasses`: `(NULL, 'Europe', NULL)`

Jedoch die Anfrage `SELECT * FROM CityInfo` enthält nicht das Tupel (Freiburg, Europe). Es existiert keine Transformation.

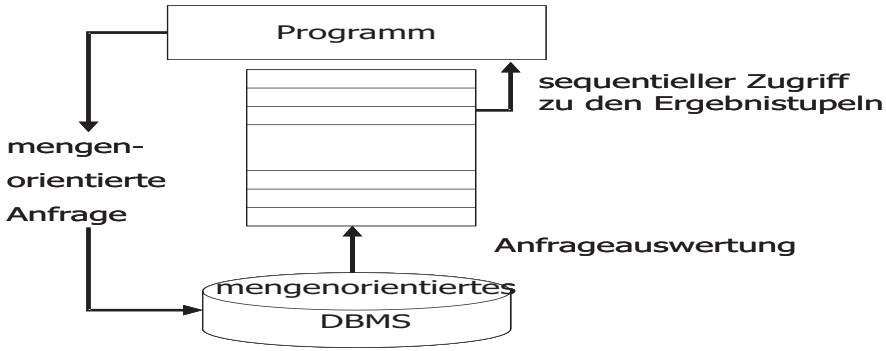
SQL: Einbettung in Hostsprachen, dynamisches SQL JDBC

- Datenbankanfragesprachen, wie SQL, sind deklarativ und mengenorientiert, jedoch nicht turing-vollständig.
- Sie geben die Turing-Vollständigkeit bewußt auf, um einer automatischen Optimierung zugänglich zu sein.
- Die Turing-Vollständigkeit kann über eine Einbettung in eine Host-Sprache (C, C++, Java, etc.), oder über Aufrufschnittstellen erreicht werden.

. . . , und natürlich ist ganz generell die Möglichkeit von einem Anwendungsprogramm aus einen DB-Zugriff vorzunehmen ein unbedingtes Muß.

Einbettung in eine Hostsprache

SQL ist mengenorientiert, Programmiersprachen sind typischerweise satzorientiert und besitzen keinen allgemeinen Datentyp `set`. Das Zusammenprallen dieser unterschiedlichen Denkweisen bezeichnet man als **Impedance Mismatch**.



Einbettungen lösen den Impedance Mismatch über ein **Cursor-**Konzept:

- Der gewünschte SQL-Ausdruck wird einem Cursor zugeordnet:

```
EXEC SQL DECLARE cityCursor CURSOR FOR
   SELECT DISTINCT City.name, Country.name
      FROM City, Country
      WHERE City.country = Country.code;
```

- Der Cursor wird geöffnet (die Anfrage wird ausgeführt) und implizit auf das erste Element der Antwortmenge positioniert:

```
EXEC SQL OPEN cityCursor;
```

- Die Antworttupel können sukzessive geholt werden:

```
EXEC SQL FETCH cityCursor INTO :cityName, :countryName;
```

- Sind die gewünschten Tupel bearbeitet, wird der Cursor geschlossen:

```
EXEC SQL CLOSE cityCursor;
```

Nachteil: Arbeiten mit einem Cursor ist umständlich; wird ein Tupel mehrmals benötigt, so muß der Cursor mehrmals geöffnet werden.

Beispiel:

```
int main() {
   EXEC SQL BEGIN DECLARE SECTION;
      char cityName[25];    /* output host var */
      int cityEinw;         /* output host var */
      char* landID = "D";    /* input host var */
      short ind1, ind2;      /* indicator var */
      char* uid = "/";
   EXEC SQL END DECLARE SECTION;  /* Verbindung herstellen */
   EXEC SQL CONNECT :uid;  /* Cursor deklarieren */
   EXEC SQL DECLARE StadtCursor CURSOR FOR
      SELECT Name, Einwohner
      FROM Stadt
      WHERE Code = :landID;
   EXEC SQL OPEN StadtCursor;  /* Cursor oeffnen */
   printf("Stadt                  Einwohner\n");
   while (1)
   {EXEC SQL FETCH StadtCursor INTO :cityName INDICATOR :ind1 ,
                                :cityEinw INDICATOR :ind2;
    if(ind1 != -1 && ind2 != -1)
    {  /* keine NULLwerte ausgeben */
       printf("%s     %d \n", cityName, cityEinw);
    }};
   EXEC SQL CLOSE StadtCursor; }
```

dynamisches SQL

Einbettungen sind **statisch**, d.h. die Anfrage muß zur Übersetzungszeit des Programms feststehen. Kann eine Anfrage als String während der Laufzeit an das DBS gereicht werden, so redet man von **dynamischem** SQL (vergl. ODBC, JDBC).

- Zunächst muß eine Verbindung zur DB aufgebaut werden.
- Anfragen werden als String übergeben:

```
ResultSet result = sql_stmt.executeQuery(
   ''SELECT DISTINCT City.name, Country.name
      FROM City, Country
      WHERE City.country = Country.code'');
```

- Zur Verarbeitung kann über die Ergebnismenge iteriert werden:

```
while(result.next()) {
   ...result.getString('' ...''); ...}
```

- Zur Effizienzsteigerung ist eine Vorübersetzung des SQL-Ausdrucks möglich, die noch eine Parameterübergabe zur Laufzeit ermöglicht.

Temporäre Verletzungen der Integrität können nicht immer ausgeschlossen werden.

Beispiel: Umbuchungen auf Konten, wobei die Summe aller Kontenstände konstant bleiben soll.

Eine **Transaktion** ist eine Folge von Datenbankoperationen, die bzgl. der Integritätsüberwachung als Einheit (atomar) betrachtet werden; verletzt eine Transaktion die Integrität, so wird sie **zurückgesetzt** (d.h. es werden die durch sie bewirkten Effekte in der Datenbank wieder entfernt).

SQL: statische Integrität

Im allgemeinen sind nur solche Instanzen einer Datenbank erlaubt, deren Relationen die der Datenbank bekannten Integritätsbedingungen erfüllen.

Integritätsbedingungen (*integrity constraints, assertions*) definieren zulässige

- Datenbankinstanzen (**statische Bedingungen**).
  Beispiele: Wertebereichseinschränkungen, Bedingung NOT NULL, Schlüsselbedingungen, referentielle Integrität.
- Zustandsübergänge, bzw. Aussagen über die Vergangenheit (**temporale Bedingungen**).
  Beispiele: Lebenszyklus von Objekten (Wechsel des Familienstandes), Gesetz "Gehälter dürfen nicht fallen", Bedingung "der aktuelle Energieverbrauch muß 5% unter dem durchschnittlichen Verbrauch der letzten 10 Jahre liegen", etc.

Integritätsbedingungen werden als Teil des Schemas abgelegt; sie können mittels `ADD` und `DROP` einem Schema hinzugefügt, oder auch aus ihm entfernt werden.

statische Integritätsbedingungen

Beispiel:

```
CREATE TABLE Lieferant
       (LNr CHAR(5),
       LName CHAR(20),
       Status DECIMAL(3) DEFAULT 0,
       Stadt StadtDomain,
       PRIMARY KEY ( LNr ))

CREATE DOMAIN StadtDomain CHAR(15),
       DEFAULT `Paris',
       CHECK (VALUE IN (`Berlin', `Paris',
                        `London', `Rom'))

CREATE TABLE Teil
       (TNr CHAR(6),
       TName CHAR(20),
       Farbe CHAR(6),
       Gewicht DECIMAL(3),
       PRIMARY KEY ( TNr ))

CREATE TABLE LT
       (LNr CHAR(5),
       TNr CHAR(6),
       Menge DECIMAL(5) CHECK (Menge > 10),
       PRIMARY KEY ( LNr, TNr ),
       FOREIGN KEY ( LNr ) REFERENCES Lieferant,
       FOREIGN KEY ( TNr ) REFERENCES Teil,
       CHECK (Menge * (SELECT Gewicht FROM Teil
                       WHERE Teil.TNr = LT.TNr)
                       <= 100))
```

- **Wertebereichsbedingungen** (*domain constraints*) ermöglichen die Definition von anwendungsspezifischen Wertebereichen.
- **Allgemeine Bedingungen** (*general constraints*) ermöglichen die Definition von Integritätsbedingungen über beliebigen Spalten beliebiger Basistabellen.
- **Basistabellenbedingungen** (*base table constraints*) ermöglichen die Definition von Integritätsbedingungen über der betreffenden Basistabelle; ihre Definition ist Teil der `CREATE`-Klausel der Basistabelle.
- **Spaltenbedingungen** (*column constraints*) sind ein Spezialfall einer Basistabellenbedingung: `NOT NULL, PRIMARY KEY, UNIQUE, FOREIGN KEY, CHECK`-Klauseln und Wertebereichsangaben.

Beispiele zu allgemeinen Bedingungen:

- *Jeder Lieferant hat einen Status von mindestens 5:*

```
CREATE ASSERTION INr13 CHECK
      ((SELECT MIN (Lieferant.Status) FROM Lieferant)
                    > 4)
```

- *Alle Teile haben ein positives Gewicht:*

```
CREATE ASSERTION INr18 CHECK
      (NOT EXISTS (SELECT * FROM Teil WHERE NOT
       (Teil.Gewicht > 0)))
```

- *Alle roten Teile müssen in London gelagert sein:*

```
CREATE ASSERTION INr99 CHECK
      (NOT EXISTS (SELECT * FROM Teil
                   WHERE Teil.Farbe = 'red' AND
                   Teil.Stadt <> 'London'))
```


- *Ein Lieferant mit Status kleiner 20 darf kein Teil in einer Menge größer als 500 liefern:*

```
CREATE ASSERTION INr95 CHECK
      (NOT EXISTS (SELECT * FROM LT
           WHERE(Menge > 500) AND
                (SELECT Status FROM Lieferant
                WHERE Lieferant.LNR = LT.LNr) < 20))
```


Beispiele zu Basistabellen- und Spaltenbedingungen:

- *Jeder Lieferant hat einen Status von mindestens 5:*

```
CHECK (Status > 4)
```

  Teil der Spaltendefinition zu `STATUS` innerhalb der `CREATE TABLE`-Klausel von Lieferant.

- *Alle roten Teile müssen in London gelagert sein:*

```
CHECK Farbe <> 'red' OR Stadt = 'London'
```

  Teil der `CREATE TABLE`-Klausel von Teil.

- *Ein Lieferant mit Status kleiner 20 darf kein Teil in einer Menge größer als 500 liefern:*

```
CHECK (LT.Menge <= 500 OR
       (SELECT Status FROM Lieferant
        WHERE Lieferant.LNr = LT.LNr)>= 20)
```

  Teil der `CREATE TABLE`-Klausel von LT.

Bemerkungen:

Eine Basistabellenbedingung zu einer Tabelle `T` der Form `CHECK (Y)` ist äquivalent zu der allgemeinen Bedingung

```
NOT EXISTS (SELECT * FROM T WHERE NOT (Y))
```

Solange `T` leer ist, ist somit jede beliebige Bedingung `Y`, sowohl beispielsweise widersprüchliche oder auch Bedingungen der Form "`T` nicht leer" immer erfüllt.

Zusammenfassung:

Typ	wo deklariert	wann überprüft	Gültigkeit
Spalten-bedingung	Attribut einer Basistabelle	bei *insert* eines Tupels in die Basistabelle, bzw. *update* der betreffenden Spalte	nicht garantiert bei Verwendung von Teilanfragen
Basistabellen-bedingung	Basistabelle	bei *insert* eines Tupels in die Basistabelle, bzw. *update* eines Tupels	nicht garantiert bei Verwendung von Teilanfragen
allgemeine Bedingung	Datenbank-schema	bei Änderung des Zustands irgendeiner Relation des Schemas	garantiert

Beispiel:

- *Jedes (gelieferte) Teil muß in mindestens 100 Einheiten geliefert werden:*

```
CHECK (100 =< ALL
      (SELECT SUM(Menge) FROM LT GROUP BY TNr));
```

  Wird diese Integritätsbedingung als Basistabellenbedingung von `LT` realisiert, so ist eine Verletzung bei Löschungen von Tupeln in `LT` möglich.

- *Jedes Teil wird auch geliefert:*

```
CHECK (TNr IN
      (SELECT TNr FROM LT));
```

  Wird diese Integritätsbedingung als Spaltenbedingung zu `TNr` bzgl. `Teil` realisiert, so ist eine Verletzung bei Änderung von `LT` möglich.

Fremdschlüsselbedingungen (referentielle Integrität)

Beispiel:

```
CREATE TABLE LT ...
       PRIMARY KEY (LNr, TNr),
       FOREIGN KEY (LNr) REFERENCES Lieferant
       FOREIGN KEY (TNr) REFERENCES Teil
```

Fremdschlüsselbedingungen sind formal Inklusionsabhängigkeiten: zu jedem von NULL verschiedenen Fremdschlüsselwert der **referenzierenden** Tabelle, der C- (child-) Tabelle, existiert ein entsprechender Schlüsselwert der **referenzierten** Tabelle, der P- (parent-) Tabelle.

- Die P- und C-Tabellen können identisch sein, bzw. es kann eine zyklische Folge von Fremdschlüsselbedingungen der Form

$$T_1, T_2, \ldots, T_k = T_1$$

existieren, wobei T_{i+1} jeweils T_i referenziert, $1 \leq i \leq k-1$.

- Zyklische Fremdschlüsselbedingungen können im allgemeinen nur mittels CREATE TABLE gefolgt von ALTER TABLE ... ADD definiert werden, da die Basistabelle zu einem Fremdschlüssel immer bereits existieren muß.
- Die Fremdschlüsselbedingungen werden jeweils bei der C-Tabelle angegeben.

Zeitpunkt der Überprüfung

Zu jeder Integritätsbedingung kann mittels IMMEDIATE und DEFERRED festgelegt werden, ob sie direkt nach Ausführung einer SQL-Anweisung, oder nach Ausführung einer Transaktion überprüft werden soll.

- Bedingungen können als DEFERRABLE oder NOT DEFERRABLE definiert werden.
- Mittels INITIALLY wird der gültige Modus für Transaktionen zu Beginn ihres Ablaufs festgelegt; ohne Angabe wird IMMEDIATE angenommen.
- Mittels der SET CONSTRAINTS Anweisung kann während der Ausführung einer Transaktion eine Bedingungen IMMEDIATE oder DEFERRED werden.

Beispiel:

Zyklische referentielle Bedingungen sind im allgemeinen bei direkter Überprüfung nicht erfüllbar.

```
CREATE TABLE T1 CONSTRAINT T1FK
FOREIGN KEY ... REFERENCES T2
INITIALLY DEFERRED

CREATE TABLE T2 CONSTRAINT T2FK
FOREIGN KEY ... REFERENCES T1
INITIALLY DEFERRED
```

SQL-Transaktion:

```
INSERT INTO T1 ( ... ) VALUES ( ...).
INSERT INTO T2 ( ... ) VALUES ( ...).
SET CONSTRAINTS T1FK, T2FK IMMEDIATE. ...
```


SQL: dynamische Integrität

Dynamische Integritätsbedingungen definieren zulässige Zustandsübergänge von Datenbankinstanzen, bzw. Aussagen über die Vergangenheit (**temporale Bedingungen**).

Beispiele:

Lebenszyklus von Objekten (Wechsel des Familienstandes), Gesetz ”Gehälter dürfen nicht fallen“, Bedingung ”der aktuelle Energieverbrauch muß 5% unter dem durchschnittlichen Verbrauch der letzten 10 Jahre liegen“, etc.

Gewährleistung der referentiellen Integrität

Referentielle Aktionen

Aufgabe **referentieller Aktionen** bzgl. einer C-Tabelle ist die Kompensierung von durch DELETE und UPDATE-Operationen auf einer zugehörigen P-Tabelle verursachten Verletzungen der Integrität.

Ist dies nicht möglich, so werden die gewünschten DELETE/UPDATE-Operationen nicht ausgeführt, bzw. zurückgesetzt.

Beispiel:

```
CREATE TABLE LT ...
       PRIMARY KEY (LNr, TNr),
       FOREIGN KEY (LNr) REFERENCES Lieferant
                   ON DELETE CASCADE ON UPDATE CASCADE,
       FOREIGN KEY (TNr) REFERENCES Teil
                   ON DELETE CASCADE ON UPDATE CASCADE)
```

Syntax der `REFERENCES`-Klausel (vereinfacht)

REFERENCES base-table [(column-commalist)]
[ON DELETE { NO ACTION | RESTRICT | CASCADE | SET DEFAULT | SET NULL }]
[ON UPDATE { NO ACTION | RESTRICT | CASCADE | SET DEFAULT | SET NULL }]

Die Klauseln ON DELETE und ON UPDATE geben die bei DELETE und UPDATE auf den P-Tabellen zur Gewährleistung der referentiellen Integrität gewünschten **referentiellen Aktionen** auf den C-Tabellen an. Fehlt die ON DELETE- oder ON UPDATE-Klausel, so ist Default NO ACTION.

Bemerkung:

- Ein INSERT bzgl. der P-Tabelle oder ein DELETE bzgl. der C-Tabelle ist immer unkritisch.
- Ein INSERT bzgl. der C-Tabelle oder ein UPDATE bzgl. der C-Tabelle, die einen Fremdschlüsselwert erzeugen, zu dem kein Schlüssel in der P-Tabelle existiert, sind immer unzulässig; anderenfalls sind sie unkritisch.
- Die references-condition definiert somit lediglich die referentiellen Aktionen für DELETE und UPDATE bzgl. der P-Tabelle.

(einige) Lösch- und Änderungs-Regeln:

NO ACTION:

Die DELETE/UPDATE-Operation auf der P-Tabelle wird zunächst ausgeführt; ob ”dangling references” in der C-Tabelle entstanden sind wird erst nach Abarbeitung aller ausgelösten referentiellen Aktionen überprüft.

RESTRICT:

Die DELETE/UPDATE-Operation auf der P-Tabelle wird nur dann ausgeführt, wenn keine ”dangling references” in der C-Tabelle entstehen.

CASCADE:

Die DELETE/UPDATE-Operation auf der P-Tabelle wird ausgeführt. Erzeugt die DELETE/UPDATE-Operation “dangling references” in der C-Tabelle, so werden die entsprechenden Zeilen der C-Tabelle ebenfalls mittels DELETE entfernt, bzw. mittels UPDATE geändert.

Ist die C-Tabelle selbst P-Tabelle bzgl. einer anderen Bedingung, so wird das DELETE/UPDATE bzgl. der dort festgelegten Lösch/Änderungs-Regel weiter behandelt.

Mehrdeutigkeiten

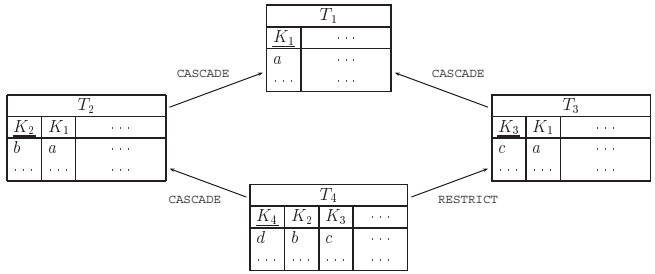
In Abhängigkeit von der Reihenfolge der Abarbeitung der FOREIGN KEY-Klauseln können in Abhängigkeit vom Inhalt der Tabellen unterschiedliche Ergebnisse resultieren.

Beispiel:

```
CREATE TABLE T1 (
       ... PRIMARY KEY K1)

CREATE TABLE T2 (                    CREATE TABLE T3 (
       ... PRIMARY KEY K2                   ... PRIMARY KEY K3
       FOREIGN KEY ( K1 )                   FOREIGN KEY ( K1 )
       REFERENCES T1 ON DELETE CASCADE )    REFERENCES T1 ON DELETE CASCADE )

CREATE TABLE T4 (
       ... PRIMARY KEY K4
       FOREIGN KEY ( K2 )
       REFERENCES T2 ON DELETE CASCADE
       FOREIGN KEY ( K3 )
       REFERENCES T3 ON DELETE RESTRICT)
```

Angenommen das Tupel in `T1` wird gelöscht und es werden die referentiellen Aktionen von `T3` vor `T4` vor `T2` betrachtet. Das Tupel in `T3` wird gelöscht (`CASCADE`-Klausel in `T3`); das Tupel in `T4` kann nicht gelöscht werden (`RESTRICT`-Klausel in `T4`). Zur Aufrechterhaltung der referentiellen Integrität wird das auslösende Löschen des Tupels in `T1` und alle dadurch implizierten Löschungen wieder rückgesetzt.

Werden die referentiellen Aktionen von `T2` vor `T4` vor `T3` betrachtet, so werden alle angegebenen Tupel gelöscht.

Ersetzt man `RESTRICT` durch `NO ACTION`, so wird unabhängig von der Reihenfolge der betrachteteten referentiellen Aktionen das Ergebnis eindeutig.

Trigger

Trigger sind ein mächtiges und flexibles Programmierungskonzept, mit dem u.a. die Integrität gewährleistet, bzw. überprüft werden kann.

Trigger sind ein Spezialfall aktiver Regeln: **E**vent**C**ondition**A**ction-Paradigma.

Beispiel:

Zu jeder Gehaltserhöhung wird noch ein Bonus von 100,-DM verrechnet.

```
CREATE TRIGGER AngestTrig
       AFTER UPDATE OF Gehalt ON Angest
       REFERENCING OLD AS oldrow NEW AS newrow
       WHEN (newrow.Gehalt > oldrow.Gehalt)
       SET newrow.Gehalt = newrow.Gehalt + 100
       FOR EACH ROW
```


Ereignis: Ein Trigger ist einer Tabelle zugeordnet. Er wird aktiviert durch das Eintreten eines Ereignisses (SQL-Anweisung): Einfügung, Änderung und Löschung von Zeilen seiner Tabelle.

Aktivierung: Der Zeitpunkt der Aktivierung ist entweder vor oder nach der eigentlichen Ausführung der entsprechenden aktivierenden Anweisung in der Datenbank. Ein Trigger kann die Ausführung der ihn aktivierenden Anweisung verhindern.

Granularität: Ein Trigger kann einmal pro aktivierender Anweisung (Statement-Trigger) oder einmal für jede betroffene Zeile (Row-Trigger) seiner Tabelle ausgeführt werden.

Transitions-Variablen: Mittels Transitions-Variablen OLD und NEW kann auf die Zeilen- und Tabellen-Inhalte vor und nach der Ausführung der aktivierenden Aktion zugegriffen werden. Im Falle von Tabellen-Inhalten handelt es sich dabei um hypothetische Tabellen, die alle betroffenen Zeilen enthalten.

Bedingung: Ein aktivierter Trigger wird ausgeführt, wenn seine Bedingung erfüllt ist.

Rumpf: Der Rumpf eines Triggers enthält die auszuführenden SQL-Anweisungen.

Trigger-Syntax (vereinfacht):

```
CREATE TRIGGER trigger-name
[ BEFORE | AFTER | INSTEAD OF] [ INSERT | DELETE | UPDATE { OF col-name }*0 ]
                                                        ON table-name
{ REFERENCING { [ OLD | NEW | OLD TABLE | NEW TABLE ] AS transition-variable }*1
{ WHEN ( trigger-condition ) } triggered-SQL-statements
[ FOR EACH STATEMENT | FOR EACH ROW ]
```

Ausführung des Triggers als:

BEFORE-Trigger: Können verwendet werden, um vor der eigentlichen Ausführung der aktivierenden Anweisung die Integrität der Werte der betreffenden Tupel zu testen, bzw. um Werte von Attributen zu berechnen, die die in der aktivierenden Anweisung zunächst vorgesehenen Werte überschreiben.

AFTER-Trigger: Können zur Gewährleistung der Integrität verwendet werden, indem entsprechende weitere Anweisungen ausgeführt werden, bzw. die aktivierende Anweisung abgebrochen und zurückgesetzt wird.

INSTEAD OF-Trigger: Können verwendet werden, um anstatt der aktivierenden Anweisung die Anweisungen des Triggers auszuführen.

Sichtbarkeit von Änderungen:

Mittels der REFERENCING-Klausel wird festgelegt unter welchen Bezeichnern die Werte der geänderten Tupel vor, bzw. nach Ausführung der den Trigger auslösenden Operation verfügbar sind.

Bei einem Statement-Trigger bezieht sich dies auf eine Tabelle (OLD TABLE, NEW TABLE), bei einem Row-Trigger auf eine Zeile (NEW, OLD).

Es muß geklärt werden, ob die Effekte der den Trigger auslösenden Operation bei der Ausführung der Trigger-Anweisung in der dem Trigger zugeordneten Tabelle sichtbar sind, oder nicht.

INSERT: Bei einem `BEFORE`- oder `INSTEAD OF`-Trigger sind die einzufügenden Tupel nicht sichtbar in der Tabelle; es kann jedoch zu ihnen mittels `NEW` oder `NEW TABLE` zugegriffen werden.

Bei einem `AFTER`-Trigger sind sie zusätzlich in der Tabelle zugreifbar.

DELETE: Bei einem `BEFORE`- oder `INSTEAD OF`-Trigger sind die zu löschenden Tupel sichtbar in der Tabelle, bei einem `AFTER`-Trigger nicht; es kann zu ihnen mittels `OLD` oder `OLD TABLE` zugegriffen werden.

UPDATE: Bei einem `BEFORE, AFTER`- oder `INSTEAD OF`-Trigger kann zu den alten und neuen Werten der zu ändernden Tupel mittels `OLD/NEW` oder `OLD TABLE/NEW TABLE` zugegriffen werden.

Bei einem `BEFORE`- oder `INSTEAD OF`-Trigger sind die Änderungen nicht sichtbar in der Tabelle, bei einem `AFTER`-Trigger jedoch.

Diskussion am Beispiel einer Angestellten-Tabelle:

```
Angest(AngNr, Name, AbtNr, Arbeitscode, Projekt, Manager,
                    MitArb, Gehalt, Bonus)
```

Before-Trigger:

Für jeden neuen Angestellten muß das Anfangsgehalt und der Bonus so sein, wie in der Tabelle `StartGehalt` *vorgesehen.*

```
CREATE TRIGGER AngestTrig2
   BEFORE INSERT ON Angest
   REFERENCING NEW AS newrow
   SET (newrow.Gehalt, newrow.Bonus) =
       (SELECT Gehalt, Bonus
       FROM StartGehalt
       WHERE Arbeitscode =
             newrow.Arbeitscode)
   FOR EACH ROW
```

Wichtige Angestellte dürfen nicht "gelöscht" werden.

```
CREATE TRIGGER AngestTrig4
       BEFORE DELETE ON Angest
       REFERENCING OLD AS oldrow
       WHEN (importance(oldrow.Arbeitscode, oldrow.Projekt) > 20)
       SIGNAL SQLSTATE '70011' ('wird noch gebraucht!')
       FOR EACH ROW
```

`SIGNAL` erzeugt hier einen speziellen Systemfehler, der rekursiv zum Abbruch mit Rücksetzen der auslösenden Anweisung führt.

Rekursive Trigger:

Trigger können selbst weitere Trigger aktivieren, wenn ein ausgelöster Trigger eine Tabelle modifiziert, über der selbst Trigger definiert sind. Eine solche Aktivierungsfolge kann zyklisch sein (**rekursive** Trigger); die Terminierung ist i.a. nicht gesichert.

Konzeptueller Datenbankentwurf: Entity-Relationship Modell I

Datenbankentwurf baut auf den Dokumenten einer Anforderungsanalyse (Pflichtenheft) auf.

Die Anforderungsanalyse beeinflußt die weiteren Entwurfsschritte:

- **konzeptueller Entwurf:** Strukturierung der Informationsanforderungen einer Miniwelt, unabhängig von den konkreten Anwendersichten. Der Entwurf wird typischerweise unter Verwendung des **Entity-Relationship-Modells** vorgenommen. Das Resultat ist das **konzeptuelle Schema**.
- **Implementationsentwurf:** Definition der Zusammenhänge eines konzeptuellen Schemas mit den Konzepten des zum Einsatz kommenden Datenbanksystems. Typischerweise wird ein relationales Datenbanksystem verwendet. Das Resultat ist das **logische Schema**. Im Unterschied zum konzeptuellen Schema, das als Spezifikationsdokument betrachtet werden kann, ist das logische Schema Teil des Datenbanksystems.

Anforderungsanalyse

Vorgehensweise:

1. Identifikation von Organisationseinheiten,
2. Identifikation der zu unterstützenden Aufgaben,
3. Anforderungs-Sammelplan: Ermittlung der zu befragenden Personen,
4. Anforderungs-Sammlung,
5. Filterung: gesammelte Information auf Verständlichkeit und Eindeutigkeit überprüfen,
6. Satzklassifikation: Information wird Objekten, Beziehungen zwischen Objekten, Operationen und Ereignissen zugeordnet,
7. Formalisierung bzw. Systematisierung: Übertragung auf Verzeichnisse, die in ihrer Gesamtheit das Pflichtenheft repräsentieren.

Bei größeren Anwendungen ist es u.U. ratsam, die Anforderungsanalyse für die einzelnen Organisationseinheiten getrennt durchzuführen.

Entity-Relationship Modell (ERM)

Die wesentlichen Strukturierungskonzepte zur Abfassung eines Schemas im ERM sind Entitäts– (entity) Typen und Beziehungs– (relationship) Typen.

- **Entitäten** sind wohlunterscheidbare physisch oder gedanklich existierende Objekte der zu modellierenden Miniwelt.
- Gleichartige Entitäten werden zu **Entitätstypen** abstrahiert.
- Entitäten können zueinander in wohlunterscheidbaren **Beziehungen** stehen.
- Gleichartige Beziehungen werden zu **Beziehungstypen** abstrahiert.

Entitätstyp: Ein Entitätstyp ist durch ein Paar $(E, \{A_1, \ldots, A_n\})$ gegeben, wobei E der Name und $\{A_1, \ldots, A_n\},\ n \geq 0$, die Menge der Attribute des Typs ist.

Attribut: Relevante Eigenschaft der Entitäten eines Typs.

Schlüssel: Zu jedem Entitätstyp ist eine Teilmenge der Attribute als Schlüssel festgelegt.

Beispiel:

(*Province,* { *name, area, population* }),
(*City,* { *name, population, longitude, latitude* }),
(*Continent,* { *name, area* })

Eigenschaften von Entitäten werden durch **Werte** aus jeweils geeigneten Wertebereichen repräsentiert.

Eine **Entität** μ eines Entitätstyps E besitzt zu jedem Attribut einen Wert.

Eine **Entitätsmenge** e eines Entitätstyps E ist eine endliche Menge von Entitäten, die die Schlüsselbedingung erfüllt.

Beispiel:

Entitätsmenge zum Entitätstyp (City, { name, population, longitude, latitude }):

{(*name: Aden, population: 250000, longitude: 50, latitude: 13*),
(*name: Katmandu, population: 393494, longitude: 85,25, latitude: 27,45*),
(*name: Ulan_Bator, population: 479500, longitude: 107, latitude: 48*)}

Beziehungstypen

Beispiel:

(*capital,* { *City, Country* }),
(*encompasses,* { *Continent, Country* }, { *percent* }),
(*belongsto,* { *Province, Country* }),
(*flowsinto,* {*tributary: River, main: River*})

Beziehungstyp: Ein Beziehungstyp ist durch ein Tripel $(B, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_n\})$ gegeben, wobei B der Name, $\{RO_1, \ldots, RO_k\},\ k \geq 2$, die Menge der sog. Rollen, $\{E_1, \ldots, E_k\}$ die den Rollen zugeordnete Entitätstypen, und $\{A_1, \ldots, A_n\}, n \geq 0$, die Menge der Attribute des Typs sind.

Die **Rollen** eines Beziehungstyps sind paarweise verschieden - die ihnen zugeordneten Entitätstypen nicht notwendigerweise. Falls $E_i = E_j$ für $i \neq j$, so liegt eine **rekursive** Beziehung vor.

Ist $k = 2$, so redet man von einem **binären** Beziehungstyp und ansonsten von einem k**-stelligen.**

k sollte nicht größer als für eine adäquate und korrekte Modellierung unbedingt notwendig gewählt werden.

Attribut: Relevante Eigenschaft der Beziehungen eines Typs.

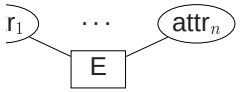
Eine **Beziehung** ν eines Beziehungstyps B ist definiert durch die beteiligten Entitäten gemäß den B zugeordneten Rollen; sie besitzt zu jedem Attribut von genau einen Wert.

Eine **Beziehungsmenge** b eines Beziehungstyps B ist eine endliche Menge von Beziehungen zu B.

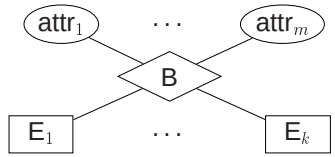
Eine Beziehungsmenge wird immer zusammen mit den Entitätsmengen der beteiligten Entitätstypen betrachtet. Jede Entität einer Beziehung existiert auch in der betreffenden Entitätsmenge (referentielle Integrität).

Graphische Darstellung (ER-Diagramm)

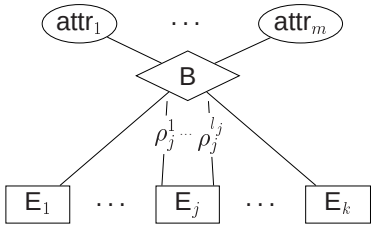
Entitätstyp E



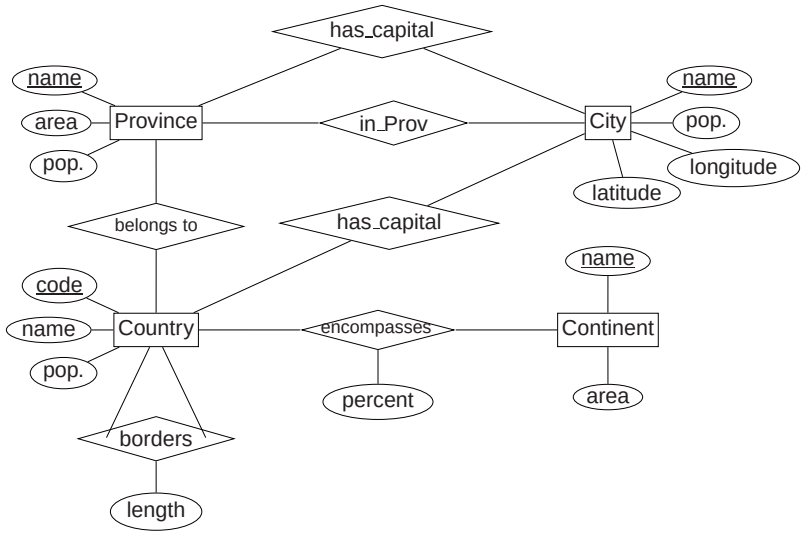
Beziehungstyp B



Beziehungstyp mit Rollen

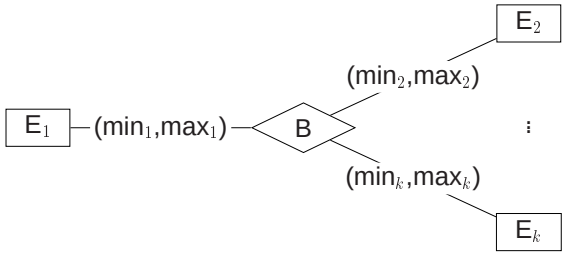


Beispiel: ER-Schema einer Geographiedatenbank



Beziehungskomplexitäten

Eine Beziehungskomplexität ist eine einem Beziehungstyp zugeordnete Integritätsbedingung; mit ihrer Festlegung wird die Mindest- und Maximalzahl von Beziehungen ausgedrückt, in denen eine Entität eines Typs unter einer bestimmten Rolle in einer Beziehungsmenge beteiligt sein darf.



Ein **Komplexitätsgrad** eines Beziehungstyps B bzgl. einer seiner Rollen RO ist ein Ausdruck der Form (min, max), bzw. $(min, *)$, wobei $0 \leq min \leq max$ und $*$ beliebig viele.

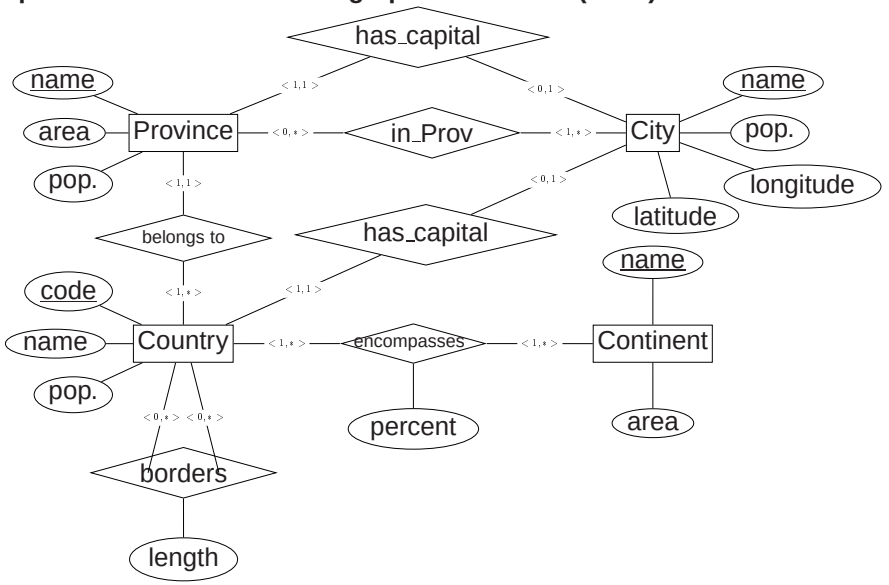
Sei B ein Beziehungstyp mit Beziehungsmenge b und E eine Entitätstyp mit Entitätsmenge e. Sei E mit Rolle RO B zugeordnet.

- b **erfüllt den Komplexitätsgrad** (min, max) der Rolle RO bzgl. e, wenn für jedes $\mu \in e$ gilt: es existieren mindestens min und maximal max Beziehungen in b, in denen μ unter der Rolle RO auftritt.
- Falls $min > 0$ redet man auch von einem **Paticipation-Constraint**.
- Falls $max = 1$ redet man auch von einem **Key-Constraint**.

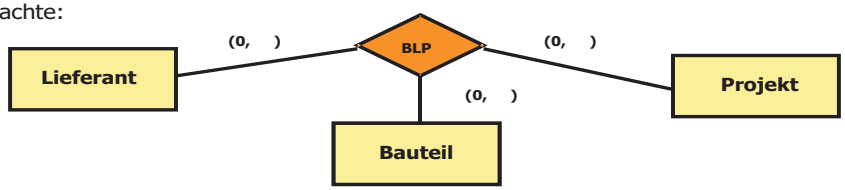
alternative Bezeichnungen für Komplexitätsgrade binärer Beziehungstypen

- Falls $max_1 = max_2 = 1$, so liegt eine $1:1$-Beziehung vor.
- Falls $max_1 > 1, max_2 = 1$, so liegt eine $n:1$-Beziehung (funktionale Beziehung) von E_2 nach E_1, bzw. eine $1:n$-Beziehung von E_1 nach E_2 vor.
- Anderenfalls eine $n:m$-Beziehung.

Beispiel: ER-Schema einer Geographiedatenbank (con't)



Betrachte:



und

- Für welche Beziehungskomplexitäten von `BLP` kann `BLP` durch `LB` und `BP` repräsentiert werden? Wie lauten dann die Beziehungskomplexitäten von `LB` und `BP`?
- Betrachte auch den Fall, dass jedes Bauteil maximal durch einen Lieferanten, jedoch möglicherweise für unterschiedliche Projekte geliefert wird.

Abbildung in das relationale Modell

Ausgehend von einem ER-Schema liefern die folgenden Schritte einen Ausgangspunkt für die Entwicklung eines logischen Schemas im Relationenmodell.

Seien E_{ER} ein Entitätstyp, B_{ER} ein Beziehungstyp und E, B Relationsschemata.

1. Entitätstypen: $(E_{ER}, \{A_1, \ldots, A_n\}) \longrightarrow E(A_1, \ldots, A_n)$,
2. Beziehungstypen:
   $(B_{ER}, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_m\}) \longrightarrow$
   $B(E_1\_K_{11}, \ldots, E_1\_K_{1p_1}, \ldots, E_k\_K_{k1}, \ldots, E_k\_K_{kp_k}, A_1, \ldots, A_m)$,
   wobei $K_i = \{K_{i1}, \ldots, K_{ip_i}\}$ Schlüssel von $E_i, 1 \leq i \leq k$.
   Falls B_{ER} Rollenbezeichnungen enthält, so kann durch Hinzunahme der Rollenbezeichnung die Eindeutigkeit der Schlüsselattribute im jeweiligen Beziehungstyp erreicht werden.
3. Hat E_{ER} bzgl. B_{ER} die Komplexität (0,1) oder (1,1), dann können E und R zu einem Entitätstyp ER zusammengefaßt werden.

Konzeptueller Datenbankentwurf: Entity-Relationship Modell II

schwache Entitätstypen

Generalisierung

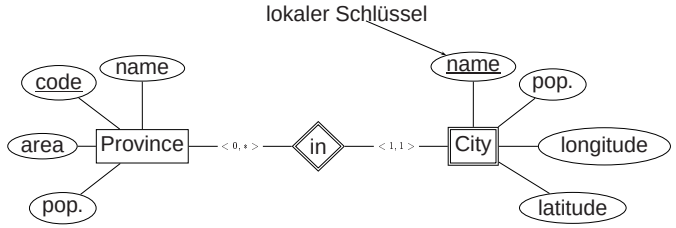
Aggregierung

mengenwertige Attribute

strukturierte Attribute

schwache Entitätstypen

Ein schwacher Entitätstyp ist ein Entitätstyp ohne (kompletten eigenen) Schlüssel.



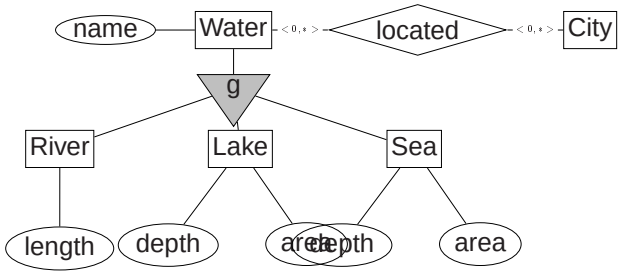
(Beispiel: Freiburg im Breisgau, Freiburg an der Oder, Freiburg (CH))

- Schwache Entitätstypen müssen mit einem (starken) Entitätstyp in einer $n:1$-Beziehung stehen.
- Sie müssen einen **lokalen** Schlüssel besitzen, d.h. Attribute, die erweitert um den Schlüssel des betreffenden (starken) Entitätstyps einen Schlüssel des schwachen Entitätstyps ergeben (**Schlüsselvererbung**).

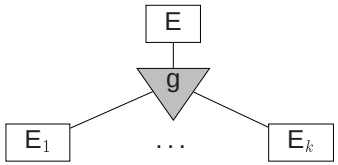
Erweiterungen des ERM: Generalisierung

Beispiel:

Flüsse, Seen und Meere bilden die Menge der Gewässer. Diese können z.B. mit einer Stadt in einer “liegt-an”-Beziehung stehen:



Generalisierung: Abstraktionsschritt, in dem Entitäten unterschiedlicher Typen zu einem gemeinsamen Typ zusammengefaßt werden. Diesem *generalisierten* Typ werden die gemeinsamen Attribute und Beziehungen der Entitäten zugeordnet.



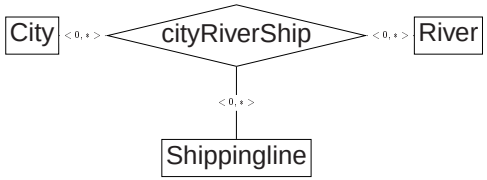
E heißt **Obertyp**, E_i heißt **Untertyp**, $1 \leq i \leq k$.

- Jede Entität eines Untertyps ist auch Entität des Obertyps. Als Konsequenz sind die Attribute und Beziehungen des Obertyps auch anwendbar auf die Untertypen (**Vererbung**).
  Vererbung ist transitiv.
- Generelle Integritätsbedingung **ISA:** ISA ist erfüllt, wenn die betreffenden Entitätsmengen der Untertypen Teilmenge der Entitätsmenge des Obertyps sind.
- Optionale Integritätsbedingung **Disjunktheit:** die Entitätsmengen der Untertypen sind disjunkt.
- Optionale Integritätsbedingung **Überdeckung:** die Vereinigung der Entitätsmengen der Untertypen überdeckt die Entitätsmenge des Obertyps.

Erweiterungen des ERM: Aggregation

Im ER-Modells dürfen Beziehungstypen nicht über Beziehungstypen definiert sein.

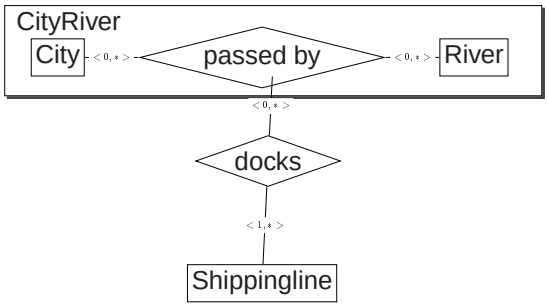
Beispiel:



Mit diesem ER-Diagramm können nur dann Beziehungen repräsentiert werden, wenn sie über allen drei beteiligten Entitätstypen definiert sind.

Naheliegender ist hier eine binäre Beziehung zwischen Städten und Flüssen, die selbst zu Schiffahrtslinien in Beziehung steht; ein solcher Zusammenhang kann im ER-Modell nicht direkt ausgedrückt werden.

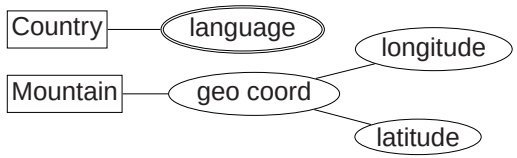
Aggregattyp:



Für den Schlüssel eines Aggregattyps sind die Primärschlüssel der an dem betreffenden Beziehungstypen beteiligten Entitätstypen Grundlage. Alle anderen Zusammenhänge wie bei Entitätstypen.

mengenwertige und strukturierte Attribute

Erweiterungen um mengenwertige und strukturierte Attribute:



Graphische Notation der verwendeten ER-Konstrukte

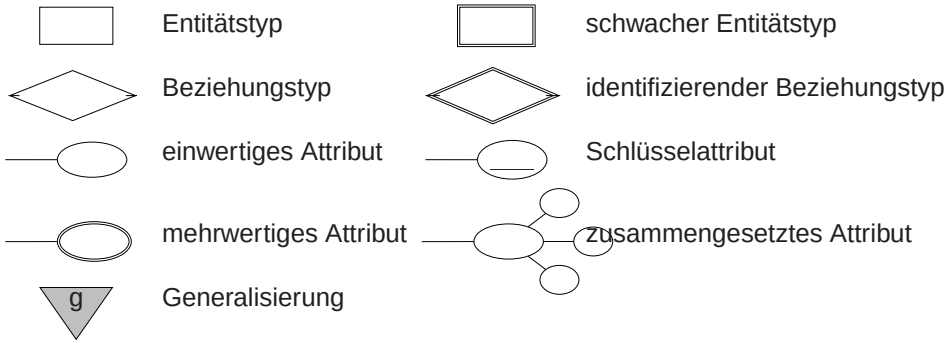
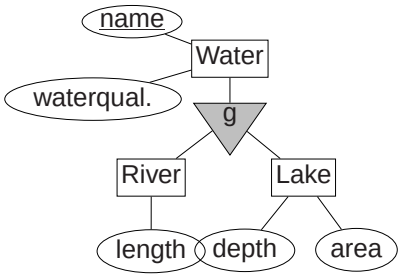


Abbildung in das relationale Modell

- Für einen schwachen Entitätstyp müssen die Schlüsselattribute der identifizierenden (starken) Entitätstypen zu dem entsprechenden Relationsschema hinzugenommen werden.
- Für Aggregattypen wird der betreffende Beziehungstyp berücksichtigt.
- Zur Darstellung von Generalisierung existieren unterschiedliche Möglichkeiten (s.u.).
- Mengenwertige Attribute können mittels einer separaten Relation repräsentiert werden. Der Schlüssel des Entitätstyps erweitert um das Attribut ergibt den Schlüssel des entsprechenden Relationsschemas.
- Strukturierte Attribute können dargestellt werden, indem der strukturierende Bezeichner den strukturierten Attributbezeichnern vorangestellt wird.

Beispiel einer Generalisierung im ERM



Water	
name	*waterqual.*
Nil	4
Aralsee	5
Elbe	2
Bodensee	1

River		
name	*waterqual.*	*length*
Nil	4	6700
Elbe	2	1144

Lake			
name	*waterqual.*	*depth*	*area*
Aralsee	5	67	62155
Bodensee	1	252	538,5

Untertypen als Sicht:

Es werden Relationsschemata für den Obertyp und die Untertypen gebildet; vom Obertyp an die Untertypen wird lediglich der Primärschlüssel vererbt.

Unterstützt Verarbeitungen auf Ebene der Obertypen. Untertypen ergeben sich als Sicht: Die fehlenden Attributwerte können mittels Verbund ergänzt werden.

Water(name, waterqual.)
River(name, length)
Lake(name, depth, area)

Water	
name	*waterqual.*
Nil	4
Aralsee	5
Elbe	2
Bodensee	1

River	
name	*length*
Nil	6700
Elbe	1144

Lake		
name	*depth*	*area*
Aralsee	67	62155
Bodensee	252	538,5

Obertypen als Sicht:

Es werden lediglich Relationsschemata für die Untertypen gebildet.

Unterstüzt Verarbeitungen auf Ebene der Untertypen. Obertypen ergeben sich als Sicht auf die Untertypen.

Nur anwendbar, wenn jede Entität des Obertyps auch zu einem Untertyp gehört. Erzeugt Redundanz, wenn Entitäten zu mehreren Untertypen gehören.

Lake(name, waterqual., depth, area)
River(name, waterqual., length)

Lake			
name	*waterqual.*	*depth*	*area*
Aralsee	5	67	62155
Bodensee	1	252	538,5

River		
name	*waterqual.*	*length*
Nil	4	6700
Elbe	2	1144

Obertypen als universelle Relation:

Alle Relationsschemata werden zu einem vereinigt; im Falle von nicht anwendbaren Attributen werden Nullwerte verwendet.

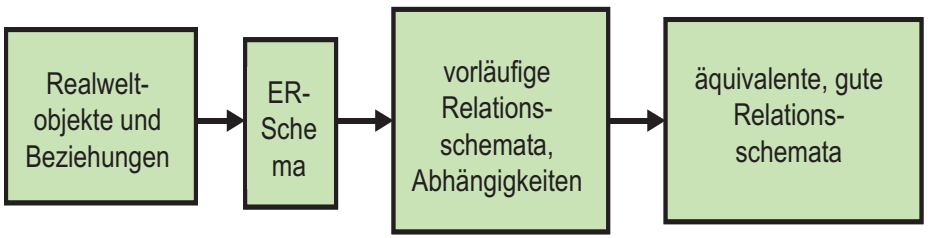
Water(name, waterqual., length, depth, area)

Water					
name	*type*	*waterqual.*	*length*	*depth*	*area*
Nil	River	4	6700	NULL	NULL
Aralsee	Lake	5	NULL	67	62155
Elbe	River	2	1144	NULL	NULL
Bodensee	Lake	1	NULL	252	538,5

Formaler Datenbankentwurf: Motivation

Ziel ist ein Datenbankschema, das die betreffenden Realweltzusammenhänge angemessen repräsentiert und bzgl. einer gegebenen Menge von **(Daten-) Abhängigkeiten** (spezielle Integritätsbedingungen), "gute" Eigenschaften besitzt.

Vorgehensweise:



Beispiel:

Es werden Lieferungen eines Lieferanten an Kunden betracht. Die Artikel werden allen Kunden zu den gleichen Preisen geliefert. Die Liefermenge und der Artikelpreis ist desweiteren von Interesse.

Lieferung	Name	Adr	Artikel	Menge	Preis
	Meier	MA	a	10	1
	Meier	MA	b	15	1
	Meier	MA	c	20	2
	Müller	KA	b	12	1
	Müller	KA	c	23	2

Die Tabelle enthält redundante Information.

durch Redundanz bewirkte Probleme:

(1) Anomalien beim Ändern (potentielle Inkonsistenz),
(2) Anomalien beim Einfügen, Löschen (Nullwerte).

verfeinertes Datenbankschema:

Kunde	Name	Adr
	Meier	MA
	Müller	KA

Lieferung	Name	Artikel	Menge
	Meier	a	10
	Meier	b	15
	Meier	c	20
	Müller	b	12
	Müller	c	23

Artikel	Artikel	Preis
	a	1
	b	1
	c	2

ist das verfeinerte Schema auch ein verbessertes Schema?

- Sind die Anomalien beseitigt? Sind die Schemaentwürfe äquivalent, d.h. können mittels der Relationenalgebra dieselben Antworten berechnet werden?

Benötigte Technologie:

- Analyse der relevanten Abhängigkeiten,
- Kriterium für Zerlegbarkeit von Relationsschemata,
- Kriterien über die Güte eines Schemaentwurfs.

Grundbegriffe

Sei V eine Menge von Attributen und $r \in \text{Rel}(V)$. Seien $X, Y \subseteq V$.
r erfüllt die **funktionale Abhängigkeit (FA)** $X \to Y$ gdw. für alle $t, s \in r$ gilt:

$$t[X] = s[X] \Rightarrow t[Y] = s[Y].$$

Gilt $Y \subseteq X$, so heißt $X \to Y$ **trivial**. Eine triviale Abhängigkeit wird durch jede Relation $r \in \text{Rel}(V)$ erfüllt.

Beispiel: Welche FA's erfüllt die folgende Relation?

A	B	C	D	E
a_2	b_1	c_1	d_1	e_1
a_1	b_2	c_1	d_1	e_1
a_1	b_1	c_2	d_1	e_1
a_1	b_1	c_1	d_2	e_1
a_1	b_1	c_1	d_1	e_2
a_1	b_1	c_1	d_1	e_1

□

Ein **(Relations)-Schema** zu einem **Relationsbezeichner** R hat die Form $R(X, \Sigma_X)$; X ist das **Format** des Schemas und Σ_X eine Menge von funktionalen Abhängigkeiten.

Eine Relation $r \in \text{Rel}(X)$ heißt **Instanz** zu R, wenn sie alle Abhängigkeiten in Σ_X erfüllt.
Die Menge aller Instanzen zu R ist $\text{Sat}(X, \Sigma_X)$, bzw. $\text{Sat}(R)$.

Beispiel:

$X = \{A, B, C\}$, $\Sigma_X = \{A \to B, B \to C, C \to A\}$.

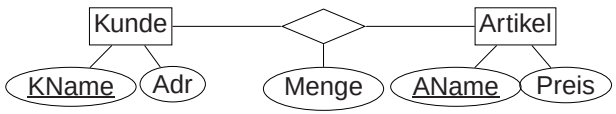
A	B	C
1	1	1
2	2	2
3	3	3
1	2	3
3	2	1

□

Das verfeinerte und verbesserte Datenbankschema zum "Lieferung"-Problem war aufgrund der Kenntnis der folgenden FAs möglich:

Name → Adr
Artikel → Preis
Name Artikel → Menge

direkter ER-Entwurf des "Lieferung"-Problems:



Kann nicht davon ausgegangen werden, daß ein "guter" ER-Entwurf bereits alle wünschenswerten Eigenschaften besitzt? Müssen wir tatsächlich den formalen Aufwand der Abhängigkeitsanalyse betreiben?

Analyse von Entitätsmengen

Es werden stundenweise Beschäftigte eines Unternehmens betrachtet. Relevante Eigenschaften seien Personal-Nummer, Name, Bewertung, Stundenlohn und Arbeitsstunden.

(A) Resultat eines ER-Entwurfs:

Entitätstyp *Teilzeit* wie folgt:

Teilzeit(Personal-Nummer, Name, Bewertung, Arbeitsstunden, Stundenlohn)

(B) Abhängigkeitsanalyse: *Bewertung* $\rightarrow$ *Stundenlohn*

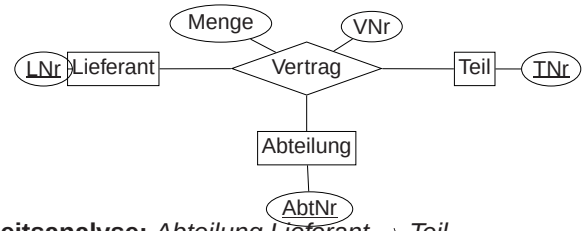
Teilzeit(Personal-Nummer, Name, Bewertung, Arbeitsstunden)
Lohn(Bewertung, Stundenlohn)

Analyse von Beziehungsmengen

Es werden Lieferanten, Teile und Abteilungen betrachtet. Es bestehen Verträge zwischen diesen dreien. Verträge sind nummeriert und legen fest, welche Menge an Teilen ein Lieferant einer Abteilung liefert.

(A) Resultat eines ER-Entwurfs:

Entitätstypen *Lieferant, Teil* und *Abteilung* und Beziehungstyp *Vertrag* wie folgt:



(B) Abhängigkeitsanalyse: *Abteilung Lieferant* $\rightarrow$ *Teil*

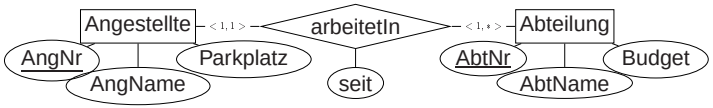
ergibt: *Vertrag(VNr, Menge, LNr, AbtNr), Lieferung(LNr, AbtNr, TNr)*

Identifizierung der Attribute eines Entitätstyps

Angestellte arbeiten in Abteilungen; jeder Angestellte in genau einer. Es sind die Nummer, der Name und der Parkplatz der Angestellten von Interesse, desweiteren die Nummer, der Name und das Budget der Abteilung und der Beginn der Tätigkeit eines Angestellten in einer Abteilung.

(A) Resultat eines ER-Entwurfs:

Entitätstypen *Angestellte* und *Abteilung*, sowie ein Beziehungstyp *arbeitetIn* wie folgt:



(B) Abhängigkeitsanalyse: *AbtNummer* $\rightarrow$ *Parkplatz*

ergibt: *Angestellte(AngNr, AngName, AbtNr, seit), Abteilung(AbtNr, AbtName, Budget, Parkplatz)*

Formaler Datenbankentwurf: funktionale Abhängigkeiten

Sei V eine Menge von Attributen, $X, Y \subseteq V$ und $\mathcal{F}$ eine Menge funktionaler Abhängigkeiten über V.

Sei $r \in Rel(V)$. r erfüllt die **funktionale Abhängigkeit (FA)** $X \rightarrow Y$ *gdw. für alle $t, s \in r$ gilt:*

$$t[X] = s[X] \Rightarrow t[Y] = s[Y].$$

$\mathcal{F}$ **impliziert** *die funktionale Abhängigkeit $X \rightarrow Y$,*

$$\mathcal{F} \models X \rightarrow Y,$$

gdw. jede Relation $r \in Sat(V, \mathcal{F})$ erfüllt auch $X \rightarrow Y$.

$\mathcal{F}^+ = \{X \rightarrow Y \mid \mathcal{F} \models X \rightarrow Y\}$ *ist die* **Hülle** *von $\mathcal{F}$.*

Sei $V = \{A_1 \ldots A_n\}$. X heißt **Schlüssel** *für V (bzgl. $\mathcal{F}$) gdw.*

(1) $X \to A_1 \ldots A_n \in \mathcal{F}^+$,
(2) $Y \subset X \Rightarrow Y \to A_1 \ldots A_n \notin \mathcal{F}^+$.

Sei X Schlüssel und $Y \supseteq X$, dann nennen wir Y **Superschlüssel**.

Gilt $A \in X$ für irgendeinen Schlüssel X, so heißt A **Schlüsselattribut (SA)**; *gilt $A \notin X$ für jeden Schlüssel X, so heißt A* **Nicht-Schlüsselattribut (NSA)**.

Problem: *Wie läßt sich entscheiden, ob $X \to Y \in \mathcal{F}^+$? (Membership Test)*
Basis für einen entsprechenden Test sind die sogenannten **Armstrong-Axiome**:

Sei $\mathcal{F}$ eine Menge **FAs** *über einer Menge V von Attributen und sei $r \in Sat(V, \mathcal{F})$.*

(A1) Reflexivität:
$Y \subseteq X \subseteq V \Rightarrow r$ *erfüllt auch* $X \to Y$.
(A2) Augmentation:
$X \to Y \in \mathcal{F}, Z \subseteq V \Rightarrow r$ *erfüllt auch* $XZ \to YZ$.
(A3) Transitivität:
$X \to Y, Y \to Z \in \mathcal{F} \Rightarrow r$ *erfüllt auch* $X \to Z$.

Beweis zu (A2):

- *Seien $t_1, t_2 \in r$. Es gilt $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$.*
- *Betrachte $XZ \to YZ$.*
- *Wegen $t_1[XZ] = t_2[XZ] \Rightarrow t_1[X] = t_2[X]$ und $t_1[Z] = t_2[Z]$, folgt $t_1[Y] = t_2[Y]$ und weiter $t_1[YZ] = t_2[YZ]$.* □

Armstrong-Axiome als Inferenzregeln für FAs:

Sei $X \subseteq V$. X^+ heißt die **(Attribut-)Hülle** *von X (bzgl. $\mathcal{F}$):*

$X^+ = \{A \mid A \in V$ *und* $X \to A$ *kann mittels (A1) - (A3) hergeleitet werden*$\}$.

Satz: *Die Armstrong-Regeln sind* **korrekt** *(alle hergeleiteten FAs sind in $\mathcal{F}^+$) und* **vollständig** *(alle FAs in $\mathcal{F}^+$ können mittels (A1)-(A3) hergeleitet werden).* □

Beweis:
Korrektheit ist klar. Zum Nachweis der Vollständigkeit zeigen wir:

> *Wenn $X \to Y$ nicht herleitbar mittels (A1)-(A3), dann $X \to Y \notin \mathcal{F}^+$, d.h. $\exists r, r$ erfüllt $\mathcal{F}$, jedoch nicht $X \to Y$.*

Betrachte zu X eine Relation r wie folgt:

$\underbrace{1\ 1\ \ldots\ 1}_{\text{Attribute in } X^+}$	$\underbrace{1\ 1\ \ldots\ 1}_{\text{alle übrigen Attribute}}$
$\underbrace{1\ 1\ \ldots\ 1}_{\text{Attribute in } X^+}$	$\underbrace{0\ 0\ \ldots\ 0}_{\text{alle übrigen Attribute}}$

(1) r erfüllt $\mathcal{F}$. Ang. $\exists Z \to W \in \mathcal{F}$, so daß r $Z \to W$ nicht erfüllt.
Damit, wegen Konstruktion von r, $Z \subseteq X^+$ und $W \not\subseteq X^+$.
Wegen $Z \subseteq X^+$ folgt $X \to Z$, $Z \to W$, und somit $W \subseteq X^+$, ein Widerspruch.
(2) r erfüllt nicht $X \to Y$. Ang. r erfüllt $X \to Y$. Damit $Y \subseteq X^+$.
D.h. jedoch gerade $X \to Y$ mit (A1)–(A3) herleitbar, ein Widerspruch.

□

Membership Problem: $X \to Y \in \mathcal{F}^+$ *?*

Membership-Test Variante 1 :
Berechne $\mathcal{F}^+$ aus $\mathcal{F}$ unter Anwendung der Regeln (A1)–(A3) solange bis entweder $X \to Y$ hergeleitet, oder $\mathcal{F}^+$ hergeleitet und $X \to Y \notin \mathcal{F}^+$.

Ein solcher Algorithmus ist i.a. nicht effizient; er hat mindestens die Zeitkomplexität $O(2^n)$, $||\mathcal{F}|| = n$.

Beispiel: $\mathcal{F} = \{A \to B_1, \ldots, A \to B_n\} \Rightarrow A \to Y \in \mathcal{F}^+$ *für $Y \subseteq \{B_1, \ldots, B_n\}$. Gilt wegen (A4) im folgenden Lemma.* □

Lemma: *Sei V eine Menge von Attributen, $\mathcal{F}$ eine Menge FAs über V und $r \in Sat(V, \mathcal{F})$. Seien weiter $X, Y, Z, W \subseteq V; A \in V$.*

(A4) Vereinigung:
$X \to Y, X \to Z \in \mathcal{F} \Rightarrow r$ *erfüllt auch* $X \to YZ$

(A5) Pseudotransitivität:
$X \to Y, WY \to Z \in \mathcal{F} \Rightarrow r$ *erfüllt auch* $XW \to Z$

(A6) Dekomposition:
$X \to Y \in \mathcal{F}, Z \subseteq Y \Rightarrow r$ *erfüllt auch* $X \to Z$

(A7) Reflexivität:
r *erfüllt* $X \to X$

(A8) Akkumulation:
$X \to YZ, Z \to AW \in \mathcal{F} \Rightarrow r$ *erfüllt auch* $X \to YZA$. □

Beweis zu (A8):

Wegen (A2) gilt auch $YZ \to YZAW$ und wegen (A3) $X \to YZAW$.

Anwenden von (A6) ergibt dann $X \to YZA$. □

Lemma: *Die Regelsysteme {(A1), (A2), (A3)} und {(A6), (A7), (A8)} sind äquivalent, d.h. für gegebenes $\mathcal{F}$ sind dieselben FAs herleitbar.* □

Membership-Test Variante 2 :
Berechne X^+ mittels (A6) - (A8) und teste $Y \subseteq X^+$.

X^+-Algorithmus:

```
result := X;
WHILE (changes to result) DO
  FOR each Y → Z ∈ F DO
     IF (Y ⊆ result ) THEN result := result ∪ Z ;
  end.
```

Satz: *Der X^+-Algorithmus berechnet in der Tat X^+ und terminiert; er benötigt Zeit $O(||\mathcal{F}||)$.* □

Beispiel:

$\mathcal{F} = \{AB \to E, BE \to I, E \to G, GI \to H\}$, $AB \to GH \in \mathcal{F}^+$ *?*

Herleitung mittels (A6)–(A8)		*Zwischenergebnis X^i des X^+-Algorithmus*
(A7)	$AB \to AB$	$X^0 = \{A, B\}$
(A8)	*[* $AB \to E$ *]*	
	$AB \to ABE$	$X^1 = \{A, B, E\}$
(A8)	*[* $BE \to I$ *]*	
	$AB \to ABEI$	$X^2 = \{A, B, E, I\}$
(A8)	*[* $E \to G$ *]*	
	$AB \to ABEIG$	$X^3 = \{A, B, E, I, G\}$
(A8)	*[* $GI \to H$ *]*	
	$AB \to ABEIGH$	$X^4 = \{A, B, E, I, G, H\}$
(A6)	$AB \to GH$	

Sei $R = (V, \mathcal{F})$ ein Relationsschema.

Bemerkung:

- *Mittels des X^+-Algorithmus kann zu R in Zeit $O(|V| \; ||\mathcal{F}||)$ ein Schlüssel bestimmt werden.*

  Man beginne mit Superschlüssel V und streiche sukzessive Attribute solange die resultierende Menge einen Superschlüssel ergibt. Kann kein Attribut mehr gestrichen werden, so hat man einen Schlüssel gefunden.
- *Alle Schlüssel eines Relationsschemas zu finden ist im allgemeinen in polynomieller Zeit nicht möglich. (Hinweis: Das Problem "Existiert ein Schlüssel mit maximal k Attributen?" ist NP-vollständig.)* □

Formaler Datenbankentwurf: abhängigkeitsbewahrende und verlustfreie Zerlegungen

verlustfreie Zerlegungen

Sei V eine Attributmenge und $\mathcal{F}$ eine Menge FAs.

Sei $\rho = \{X_1, \ldots, X_k\}$ eine **Zerlegung** *von V.*

ρ heißt **verlustfrei**, *gdw. für jede Relation $r \in Sat(V, \mathcal{F})$ gilt:*

$$r = \pi[X_1](r) \bowtie \ldots \bowtie \pi[X_k](r).$$

Beobachtung:

Sei $V = \{A, B, C\}$ und $\mathcal{F} = \{A \to B, A \to C\}$).

Sei $r \in Sat(V, \mathcal{F})$ wie folgt:

A	B	C
a_1	b_1	c_1
a_2	b_1	c_2

Dann

$$r \subset \pi[AB](r) \bowtie \pi[BC](r),$$
$$r = \pi[AB](r) \bowtie \pi[AC](r).$$

□

Lemma: *Sei $r \in Rel(V)$ und $\rho = \{X_1, \ldots, X_k\}$ eine Zerlegung von V.*

$$r \subseteq \pi[X_1](r) \bowtie \ldots \bowtie \pi[X_k](r)$$

. □

Satz: *Sei V eine Attributmenge mit einer Menge $\mathcal{F}$ funktionaler Abhängigkeiten. Sei weiter $\rho = (X_1, X_2)$ eine Zerlegung von V. ρ ist verlustfrei, gdw.*

$$(X_1 \cap X_2) \to (X_1 \setminus X_2) \in \mathcal{F}^+, oder \; (X_1 \cap X_2) \to (X_2 \setminus X_1) \in \mathcal{F}^+.$$

Beweis:
Angenommen ρ verlustfrei und $(X_1 \cap X_2) \to (X_1 \setminus X_2) \notin \mathcal{F}^+$ und $(X_1 \cap X_2) \to (X_2 \setminus X_1) \notin \mathcal{F}^+$. Sei $X = (X_1 \cap X_2)^+$.

Dann existiert $r \in Sat(V, \mathcal{F})$ der folgenden Form:

	X	$X_1 \setminus X$	$X_2 \setminus X$
X_1	$a \ldots a$	$a \ldots a$	$b \ldots b$
X_2	$a \ldots a$	$b \ldots b$	$a \ldots a$

Wegen $X_1 = (X_1 \cap X_2) \cup (X_1 \setminus X_2)$ und $X_2 = (X_1 \cap X_2) \cup (X_2 \setminus X_1)$ ergibt dies einen Widerspruch zur Verlustfreiheit von ρ.

Die andere Richtung ist klar. □

Sei ein Schema $R = (V, \mathcal{F})$ und eine Zerlegung $\rho = \{X_1, \ldots, X_k\}$ von V gegeben.

Wünschenswerte Eigenschaften von ρ:

(1) **verlustfrei**, *d.h. die Zusammengehörigkeit der Komponenten der einzelnen Tupel bleibt mittels $\bowtie$ rekonstruierbar, ohne daß zusätzliche Tupel (quasi als Nebeneffekt) gebildet werden.*

(2) **abhängigkeitsbewahrend**, *d.h. die Abhängigkeiten des Relationsschemas bleiben auch bzgl. der Zerlegung (ohne den Verbund zu ρ zu berechnen) überprüfbar. Dies bedeutet insbesondere, daß es bei Einfügungen und Änderungen von Tupeln genügt, die Zerlegung ρ zu betrachten.*

abhängigkeitsbewahrende Zerlegungen

Seien $\mathcal{F}, \mathcal{G}$ Mengen von funktionalen Abhängigkeiten.

- *$\mathcal{F}, \mathcal{G}$ heißen* **äquivalent**, *$\mathcal{F} \equiv \mathcal{G}$, gdw. $\mathcal{F}^+ = \mathcal{G}^+$.*
- *$\mathcal{F}$ heißt* **minimal**, *gdw.*
  - *(1) $X \to Y \in \mathcal{F} \Rightarrow Y$ besteht aus genau einem Attribut.*
  - *(2) $X \to A \in \mathcal{F} \Rightarrow \mathcal{F} \setminus \{X \to A\}$ nicht äquivalent $\mathcal{F}$.*
  - *(3) $X \to A \in \mathcal{F}, Z \subset X \Rightarrow \mathcal{F} \setminus \{X \to A\} \cup \{Z \to A\}$ nicht äquivalent $\mathcal{F}$.*

Satz: *Zu jeder Menge $\mathcal{F}$ von funktionalen Abhängigkeiten existiert eine äquivalente minimale Menge von funktionalen Abhängigkeiten $\mathcal{F}^{min}$.* □

Beispiel:

1. *Betrachte*
$$A \to B, B \to A, B \to C, A \to C, C \to A$$
Es können entweder $B \to A$, $A \to C$, oder $B \to C$ eliminiert werden.
2. *Betrachte*
$$AB \to C, A \to B, B \to A$$
$AB \to C$ kann entweder zu $A \to C$, oder zu $B \to C$ reduziert werden.

Sei $\mathcal{F}$ gegeben und sei $\mathcal{F}'$ die erzeugte Menge von FA's. Gilt $\mathcal{F}^+ = \mathcal{F}'^+$?

zu 1. (Rechtsreduktion): *Sei $X \to Y \in \mathcal{F}$ und $X \to Y \notin \mathcal{F}'$.*
Wegen $\mathcal{F}' \subseteq \mathcal{F}$ gilt $\mathcal{F}'^+ \subseteq \mathcal{F}^+$.
$\mathcal{F}^+ \subseteq \mathcal{F}'^+$ gilt, sofern $Y \subseteq X^+_{\mathcal{F}'}$.

zu 2. (Linksreduktion): *Sei $X \to Y \in \mathcal{F}$ und $X' \to Y \in \mathcal{F}'$, $X' \subset X$.*
Wegen $X' \subseteq X$ gilt $\mathcal{F}^+ \subseteq \mathcal{F}'^+$.
$\mathcal{F}'^+ \subseteq \mathcal{F}^+$ gilt, sofern $Y \subseteq X'^+_{\mathcal{F}}$. □

Bemerkung:

- *$\mathcal{F}^{min}$ kann mittels des X^+-Algorithmus (ohne Berechnung von $\mathcal{F}^+$) in polynomieller Zeit erzeugt werden.*
- *$\mathcal{F}^{min}$ ist nicht notwendigerweise eindeutig.*

Sei $R = (V, \mathcal{F})$ gegeben. Sei weiter $Z \subseteq V$.

$\pi[Z](\mathcal{F}) = \{X \to Y \in \mathcal{F}^+ \mid XY \subseteq Z\}$ ist die Projektion von $\mathcal{F}$ auf Z.

Eine Zerlegung $\rho = \{X_1, \ldots, X_k\}$ von V ist **abhängigkeitsbewahrend** *bzgl. $\mathcal{F}$ gdw.*

$$\bigcup_{i=1}^{k} \pi[X_i](\mathcal{F}) \equiv \mathcal{F}.$$

Beispiel :

Sei $V = \{A, B, C, D\}$, $\mathcal{F} = \{A \to B, B \to C, C \to A\}$, und $\rho = \{AB, BC\}$.

Ist ρ abhängigkeitsbewahrend ? (d.h. bleibt $C \to A$ erhalten ?)

Ja, wegen : $\pi[AB](\mathcal{F}) \supseteq \{A \to B, B \to A\}$, $\pi[BC](\mathcal{F}) \supseteq \{B \to C, C \to B\}$ gilt auch $C \to A \in (\pi[AB](\mathcal{F}) \cup \pi[BC](\mathcal{F}))^+$. □

Bemerkung :

(A) Es gibt Mengen $V, \mathcal{F}$, $X \subseteq V$, wobei $\mathcal{F}$ $2n+1$ FAs, so dass $\pi[X](\mathcal{F})$ mindestens 2^n Elemente hat.
Betrachte

$$V = \{A_1, \ldots, A_n, B_1, \ldots B_n, C_1, \ldots, C_n, D\},$$
$$\mathcal{F} = \{A_i \to C_i, B_i \to C_i \mid 1 \leq i \leq n\} \cup \{C_1 \ldots C_n \to D\},$$
$$X = A_1 \ldots A_n B_1 \ldots B_n D.$$

(B) Es gibt verlustfreie Zerlegungen, die nicht abhängigkeitsbewahrend sind !
Betrachte $R = (V, \mathcal{F})$, wobei

- *V = { Stadt, Adresse, PLZ }, und*
- *$\mathcal{F}$ =\{ Stadt Adresse $\to$ PLZ, PLZ $\to$ Stadt }.*

R_1(Adresse, PLZ), R_2(Stadt,PLZ) ist verlustfrei wg. $(R_1 \cap R_2) \to (R_2 \setminus R_1) \in \mathcal{F}$, jedoch offensichtlich nicht abhängigkeitsbewahrend.
Beachte R hat die Schlüssel Adresse PLZ und Stadt Adresse.

(B):

$\mathcal{F}$ = { Stadt Adresse $\to$ PLZ, PLZ $\to$ Stadt, PLZ $\to$ Einwohner }

R	Stadt	Adresse	PLZ	Briefträger	Einwohner
	FR	Wiehre	79104	Hurtig	100Tsd
	FR	Wiehre	79104	Sorgsam	100Tsd
	FR	Wiehre	79104	Luftig	100Tsd
	FR	Herdern	79106	Eilig	80Tsd
	FR	Flughafen	79110	Luftig	20Tsd

Es soll zusätzlich (FR, Herdern, 79100, Eilig, 80Tsd) eingefügt werden.
Die Verletzung der FA Stadt Adresse $\to$ PLZ kann direkt erkannt werden.

R_1	Adresse	PLZ	Briefträger
	Wiehre	79104	Hurtig
	Wiehre	79104	Sorgsam
	Wiehre	79104	Luftig
	Herdern	79106	Eilig
	Flughafen	79110	Luftig
	Herdern	79100	Eilig

R_2	Stadt	PLZ	Einwohner
	FR	79104	100Tsd
	FR	79106	80Tsd
	FR	79110	20Tsd
	FR	79100	80Tsd

Die durch die neuen Tupel hervorgerufene Verletzung der FA Stadt Adresse $\to$ PLZ kann nur erkannt werden, wenn der Verbund der Relationen berechnet wird. □

(A):

Betrachte

$$V = \{A_1, \ldots, A_n, B_1, \ldots B_n, C_1, \ldots, C_n, D\},$$
$$\mathcal{F} = \{A_i \to C_i, B_i \to C_i \mid 1 \leq i \leq n\} \cup \{C_1 \ldots C_n \to D\},$$
$$X = A_1 \ldots A_n B_1 \ldots B_n D.$$

- *$\mathcal{F}$ ist offensichtlich minimal.*
- *Es gilt weiter*
  - *$A_1 A_2 ... A_k \to D \in \mathcal{F}^+$,*
  - *$B_1 B_2 ... B_k \to D \in \mathcal{F}^+$,*
  - *und allgemein für $w \in \Pi_{i=1}^n \{A_i, B_i\}$ auch $w \to D \in \mathcal{F}^+$.*
- *$\mid \{w \mid w \in \Pi_{i=1}^n \{A_i, B_i\}\} \mid = 2^n$.*
- *Sei $\mathcal{F}' = \{w \to D \mid w \in \Pi_{i=1}^n \{A_i, B_i\}\}$. $\mathcal{F}' \subseteq \pi[X]\mathcal{F}$.*
- *$\mid \pi[X]\mathcal{F} \mid = O(2^n)$.*

Formaler Datenbankentwurf: Normalformen

Normalisierungsproblem:

Sei $R = (V, \mathcal{F})$ ein Schema.

Finde eine Zerlegung $\rho = (X_1, \ldots, X_n)$ von R so, dass:

(1) jedes $R_i = (X_i, \pi[X_i](\mathcal{F}))$, $1 \leq i \leq n$ ist in der gewünschten Normalform,
(2) ρ ist verlustfrei und (möglichst) auch abhängigkeitsbewahrend,
(3) n minimal.

Beispiel:

Angenommen, die Systemanalyse eines Unternehmens hat als relevante Menge von Daten ergeben:

Lieferantennummer	*LNr*
Lieferantenregion	*R*
Lieferantenstadt	*S*
Artikelnummer	*ANr*
Artikelmenge	*M*
Projektnummer	*PNr*
Sachbearbeiter	*B*

Die Daten werden in folgender Weise in einem Formular (Relation mit Wiederholungsgruppen) verwendet:

			ArtProj			
			MeProj			
LNr	*R*	*S*	*ANr*	*M*	*PNr*	*B*
L1	*BW*	*MA*	*A1*	*20*	*P5*	*aa*
				30	*P6*	*bb*
			A2	*10*	*P7*	
			A3	*20*	*P6*	
L2	*H*	*DA*	*A2*	*20*	*P7*	*bb*
				30	*P5*	*cc*
				20	*P6*	
L3	*BW*	*MA*	*A1*	*10*	*P5*	*cc*

Genestete Relationen in dieser Form bezeichnet man als **unnormalisiert**.

die 1. Normalform (1NF):

Definition: *Ein Relationsschema ist in 1NF gdw. die Wertebereiche der Attribute atomar sind.* □

Unnormalisierte Relationen werden in 1NF gebracht durch 'Ausmultiplizieren' von Wiederholungsgruppen auf gleicher Stufe.

Beispiel:

r_1 :

LNr	R	S	ANr	M	PNr	B
L1	BW	MA	A1	20	P5	aa
L1	BW	MA	A1	20	P5	bb
L1	BW	MA	A1	30	P6	aa
L1	BW	MA	A1	30	P6	bb
L1	BW	MA	A2	10	P7	aa
L1	BW	MA	A2	10	P7	bb
L1	BW	MA	A3	20	P6	aa
L1	BW	MA	A3	20	P6	bb
L2	H	DA	A2	20	P7	bb
L2	H	DA	A2	20	P7	cc
L2	H	DA	A2	30	P5	bb
L2	H	DA	A2	30	P5	cc
L2	H	DA	A2	20	P6	bb
L2	H	DA	A2	20	P6	cc
L3	BW	MA	A1	10	P5	cc

R_1 = LNr R S ANr M PNr B

$\mathcal{F}_1$ = { LNr → R S, LNr ANr PNr → M, S → R}

Viele Redundanzen!

die 3. Normalform (3NF):

Definition: *Ein Relationsschema $R = (V, \mathcal{F})$ ist in 3NF gdw. für jedes NSA A gilt: $X \to A \in \mathcal{F}$, $A \notin X \Rightarrow X$ enthält einen Schlüssel.* □

3NF verbietet nichttriviale FAs $X \to A$, in denen ein NSA A in der Weise von einem Schlüssel K funktional abhängt, daß $K \to X$, $K \not\subseteq X$ und $X \to A$. Es können dann mehrere Tupel mit demselben X-Wert existieren. Für alle diese Tupel muß A wiederholt werden.

Beispiel 3NF-Zerlegung: verlustfrei und abhängigkeitsbewahrend

R_{11} = LNr S, $\mathcal{F}_{11} = \pi[LNr, S]\mathcal{F}$ = { LNr → S };
R_{12} = S R, $\mathcal{F}_{12} = \pi[S, R]\mathcal{F}$ = { S → R };
R_{13} = LNr ANr PNr M, $\mathcal{F}_{13} = \pi[LNr, ANr, PNr, M]\mathcal{F}$ = { LNr ANr PNr → M };
R_{14} = LNr ANr PNr B , $\mathcal{F}_{14} = \pi]LNr, ANr, PNr, B]\mathcal{F}$ = ∅.

□

die Boyce-Codd Normalform (BCNF):

Definition .1 *Ein Relationsschema $R = (V, \mathcal{F})$ ist in BCNF gdw. $X \to A \in \mathcal{F}$, $A \notin X \Rightarrow X$ enthält einen Schlüssel.* □

BCNF erweitert 3NF auf SA's.

Beispiel BCNF-Zerlegung: verlustfrei und nicht abhängigkeitsbewahrend

Sei $R = (V, \mathcal{F})$, wobei V = { Stadt, Adresse, PLZ }, und $\mathcal{F}$ ={ Stadt Adresse → PLZ, PLZ → Stadt }. R ist in 3NF, aber nicht in BCNF.

R_1(Adresse, PLZ), R_2(Stadt,PLZ) ist eine verlustfreie, nicht abhängigkeitsbewahrend BCNF-Zerlegung. □

Beispiel BCNF-Zerlegung: verlustfrei und abhängigkeitsbewahrend

$(R_{11}, \mathcal{F}_{11})$, $(R_{12}, \mathcal{F}_{12})$, $(R_{13}, \mathcal{F}_{13})$, $(R_{14}, \mathcal{F}_{14})$.

□

einige interessante Zusammenhänge

Satz: *Ein Relationsschema R habe genau einen Schlüssel. R ist in BCNF genau dann, wenn R in 3NF.*

□

Lemma: *Ein Relationsschema $R = (V, \mathcal{F})$ ist in BCNF genau dann, wenn für jede nichttriviale FA $X \to A \in \mathcal{F}^+$ X ein Superschlüssel ist.* □

Korollar *Ein Relationsschema $R = (V, \mathcal{F})$ ist in BCNF genau dann, wenn $R' = (V, \mathcal{F}^+)$ in BCNF.* □

Bemerkung:

- *BCNF kann in polynomieller Zeit getestet werden.*
  Idee: Teste mittels X^+-Algorithmus für jede gegebene FA $X \to Y$ ob X ein Superschlüssel.
- *Der Test auf 3NF ist NP-vollständig.*
- *Sei $R = (V, \mathcal{F})$ ein Relationsschema mit FA $X \to Y$, wobei $X \cap Y = \emptyset$. Dann ist die Zerlegung $\rho = (R \setminus Y, XY)$ verlustfrei.*
- *Sei $\mathcal{F}$ eine Menge FA's über V und sei $X \subseteq V$.*
  Zur Berechnung von $\pi[X]\mathcal{F}$ sind lediglich Algorithmen bekannt, die im schlechtesten Fall exponentiell in der Mächtigkeit von X sind.

Weitere Datenabhängigkeiten

- *Sei V eine Menge von Attributen und $r \in$ Rel(V). Seien $X_1, \ldots, X_n$ eine Zerlegung von V.*
  r erfüllt die **Verbundabhängigkeit (VA)** *$\bowtie [X_1, \ldots, X_n]$ gdw. gilt:*

$$r = \bowtie_{i=1}^{n} \pi[X_i](r).$$

  Falls $n = 2$ redet man von einer **mehrwertigen Abhängigkeit (MVD)** *und schreibt*

$$X_1 \cap X_2 \twoheadrightarrow X_1 \setminus X_2, \text{ bzw., } X_1 \cap X_2 \twoheadrightarrow X_2 \setminus X_1.$$

- *Seien X_1, X_2 Mengen von Attributen und $r_1 \in$ Rel$(X_1), r_2 \in$ Rel(X_2) Relationen. Sei $Y \subseteq X_1 \cap X_2$.*
  r_1, r_2 erfüllen die **Inklusionsabhängigkeit (IA)** *$R_1[Y] \subseteq R_2[Y]$ gdw. gilt:*

$$\pi[Y](r_1) \subseteq \pi[Y](r_2).$$

Inklusionsabhängigkeiten verallgemeinern referentielle Integritätsbedingungen.

die 4. Normalform (4NF):

Ziel: *Von einander unabhängige Zusammenhänge sollen nicht in derselben Relation repräsentiert werden!*

Sei $R = (V, \mathcal{D})$ ein Relationsschema, $\mathcal{D}$ eine Menge von MVDs und FAs.

Definition: *Ein Relationsschema $R = (V, \mathcal{D})$ ist in 4NF gdw. für $Y \not\subseteq X$, $XY \neq V$ gilt: $X \twoheadrightarrow Y \in \mathcal{D} \Rightarrow X$ enthält einen Schlüssel.* □

Bemerkung:

- *Jede FA ist auch eine MVD, jedoch nicht umgekehrt.*
- *Die* **trivialen** *MVDs haben die Form $X \twoheadrightarrow Y$, $Y \subseteq X$, bzw. $X \twoheadrightarrow V \setminus X$.*
- *Für MVD's gilt die Komplementeigenschaft:*
  Wenn $X \twoheadrightarrow Y \in \mathcal{D}^+$, dann auch $X \twoheadrightarrow (V \setminus (X \cup Y)) \in \mathcal{D}^+$.

Beispiel:

R_{14} = LNr ANr PNr B;
$\mathcal{D}_{14}$ = { LNr $\twoheadrightarrow$ ANr PNr}

R_{141} = LNr ANr PNr;
R_{142} = LNr B

r_{14} :

LNr	ANr	PNr	B
L1	A1	P5	aa
L1	A1	P5	bb
L1	A1	P6	aa
L1	A1	P6	bb
L1	A2	P7	aa
L1	A2	P7	bb
L1	A3	P6	aa
L1	A3	P6	bb
L2	A2	P7	bb
L2	A2	P7	cc
L2	A2	P5	bb
L2	A2	P5	cc
L2	A2	P6	bb
L2	A2	P6	cc
L3	A1	P5	cc

$\rho = (R_{11}, R_{12}, R_{13}, R_{142})$ ist eine verlustfreie und abhängigkeitsbewahrende Zerlegung von R_1 in 4NF.

Bemerkungen:

- *1NF $\Leftarrow$ (2NF) $\Leftarrow$ 3NF $\Leftarrow$ BCNF $\Leftarrow$ 4NF - die umgekehrten Richtungen gelten nicht.*
  2NF ist lediglich von historischem Interesse.
- *Es existiert im allgemeinen keine abhängigkeitsbewahrende und verlustfreie Zerlegung über 3NF hinaus.*
- *Es existiert immer eine verlustfreie Zerlegung in 4NF.* □

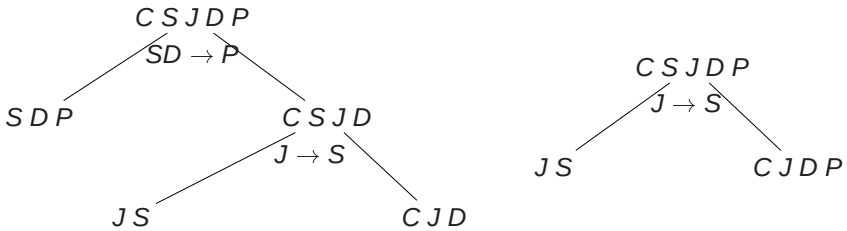
Formaler Datenbankentwurf: Algorithmen

BCNF-Analyse: verlustfrei und nicht abhängigkeitsbewahrend

Sei $R = (V, \mathcal{F})$ ein Relationsschema gegeben. R sei nicht in BCNF.

1. *Sei $X \subset V$, $A \in V$ und $X \to A \in \mathcal{F}$ eine FA, die die BCNF-Bedingung verletzt.*
   Sei $\rho = (V - A, XA)$ eine Zerlegung von V und entsprechend
   $$R_1 = (V - A, \pi[V - A]\mathcal{F}), R_2 = (XA, \pi[XA]\mathcal{F})$$
   die entsprechenden Schemata.
2. *Teste die BCNF-Bedingung bzgl. R_1 und R_2 und wende den Algorithmus gegebenenfalls rekursiv an.*

Beispiel: V = {C,S,J,D,P}, $\mathcal{F}$ = {SD $\to$ P, J $\to$ S}



3NF-Analyse: verlustfrei und abhängigkeitsbewahrend

Sei $R = (V, \mathcal{F})$ ein Relationsschema gegeben und sei $\rho = (X_1, \ldots, X_k)$ eine Zerlegung von V, so daß die entsprechenden Schemata $R_1 = (X_1, \pi[X_1]\mathcal{F})$, ..., $R_k = (X_k, \pi[X_k]\mathcal{F})$ in BCNF. Sei $\mathcal{F}$ minimal.

1. *Identifiziere die Menge N derjenigen FA's, für die die Abhängigkeitsbewahrung verletzt ist.*
2. *Für jede solche FA der Form $X \to A$ erweitere ρ um XA; die entsprechenden Schemata haben die Form $(XA, \pi[XA]\mathcal{F})$.*

Die resultierende Zerlegung ist offensichtlich weiterhin verlustfrei und zusätzlich abhängigkeitsbewahrend. Jedes neu eingeführte Schema der Zerlegung der Form $(XA, \pi[XA]\mathcal{F})$ ist zusätzlich in 3NF.

Beweisidee: *Wegen $X \to A \in \mathcal{F}$ und $\mathcal{F}$ minimal gilt nicht $Y \to A$ für $Y \subset X$. Damit ist X Schlüssel für XA. Weitere FAs über XA können somit nur über X definiert sein. Wegen X Schlüssel können diese jedoch nicht die 3NF-Bedingung verletzten (wohl aber die BCNF-Bedingung).* □

Beispiel: V = {C,S,J,D,P}, $\mathcal{F}$ = {SD $\rightarrow$ P, J $\rightarrow$ S}

3NF-Synthese: verlustfrei und abhängigkeitsbewahrend

3NF-Synthese-Algorithmus
Eingabe: *Relationsschema $R = (V, \mathcal{F})$ und dazugehöriges $\mathcal{F}^{min}$.*
Ausgabe: *Verlustfreie und abhängigkeitsbewahrende Zerlegung $\rho = (X_1, \ldots, X_k)$, wobei alle $R_i = (X_i, \pi[X_i](\mathcal{F}))$ in 3NF.*

(1) Betrachte jeweils maximale Klassen von FA's aus F^{min} mit derselben linken Seite. Sei $\{X \rightarrow A_1, X \rightarrow A_2, \ldots\}$ eine solche Klasse. Bilde zu jeder solchen Klasse ein Schema mit Format $XA_1A_2\ldots$.

(2) Sofern unter den in (1) gebildeten Schemata kein Format einen Schlüssel für R enthält, berechne einen Schlüssel für R; sei K ein solcher Schlüssel. Dann bilde zu K ein Schema mit Format K.

Die gewünschte Zerlegung ergibt sich aus den Schemata mit Formaten gebildet in (1) und (2). □

Beispiel: *Sei R gegeben durch V = {C,S,J,D,P} und $\mathcal{F}$ = {SD $\rightarrow$ P, J $\rightarrow$ S}*

(1) $\mathcal{F}$ ist eine minimale Überdeckung.

(2) Es ergeben sich SDP und JS. Beide enthalten keinen Schlüssel zu R.

(3) Schlüssel zu R ist CJD. Damit

$$\rho = \{SDP, JS, CJD\}$$

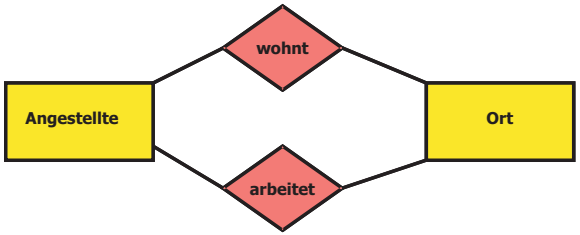
Bemerkung :

- *Der 3NF-Synthese-Algorithmus ist korrekt (s. Literatur).*
- *Der 3NF-Synthese-Algorithmus benötigt polynomielle Zeit.*
- *Es ist jedoch ein NP-vollständiges Problem zu testen, ob ein gegebenes Relationsschema in 3NF ist.*
- *Durch Algorithmus erzeugtes ρ ist nicht notwendig minimal.*
  Sei $V = \{AB\}, \mathcal{F}^{min} = \{A \rightarrow B, B \rightarrow A\}$; dann ergibt sich $\rho = (\underline{A}B, \underline{B}A)$. □

Eindeutigkeitsannahme

Die formale Entwurfstheorie beruht auf der Annahme, dass mit gleichen Namen ausgedrückte Zusammenhänge auch identisch sind.

Betrachte:



Es sei hier intendiert, dass Angestellte in unterschiedlichen Orten wohnen und arbeiten können.
Die folgende relationale Darstellung berücksichtigt dies nicht explizit:
Angest(ANr, . . .), Ort(PLZ, . . .), wohnt(ANr, PLZ, . . .), arbeitet(ANr, PLZ, . . .)

Korrekt wäre: wohnt(ANr, W-PLZ, . . .) und arbeitet(ANr, A-PLZ, . . .)
Insbesondere ist folgendes Relationenschema inkorrekt:
AngestOrt(ANr, . . . , PLZ, . . .)

- Eine **Datei** ist eine Menge von Sätzen des gleichen Typs; sie repräsentiert einen Teil des konzeptuellen Schemas, beispielsweise eine Relation, einen Teil einer Relation, den Verbund mehrerer Relationen, etc., oder auch eine zur Effizienzsteigerung erforderliche Hilfsstruktur (**Index**).
- Die Zugriffseinheiten einer Datei sind **Seiten**, oder auch **Blöcke**. Typischerweise hat eine Seite eine Größe von 4KB bis 8KB, bzw. besteht ein Block aus aufeinanderfolgenden Seiten mit einer Größe bis zu 32KB.
- Jeder **Satz** besteht aus einem oder mehreren **Feldern**; eine Seite enthält in der Regel mehrere Sätze.

Physischer Datenbankentwurf: Datenspeicherung

Beim **Entwurf des konzeptuellen Schemas** wird definiert, welche Daten benötigt werden und wie sie zusammenhängen (**logische Datenbank**).

Beim **Entwurf des physischen Schemas** muß vor dem Hintergrund des konzeptuellen Schemas und den zu erwartenden Anwendungen eine Abbildung auf eine **physische Datenbank** definiert werden, die eine möglichst hohe Effizienz bei der Ausführung von Anfragen und Anwendungsprogrammen garantiert.

Eine **physische Datenbank** ist eine Menge von Dateien.

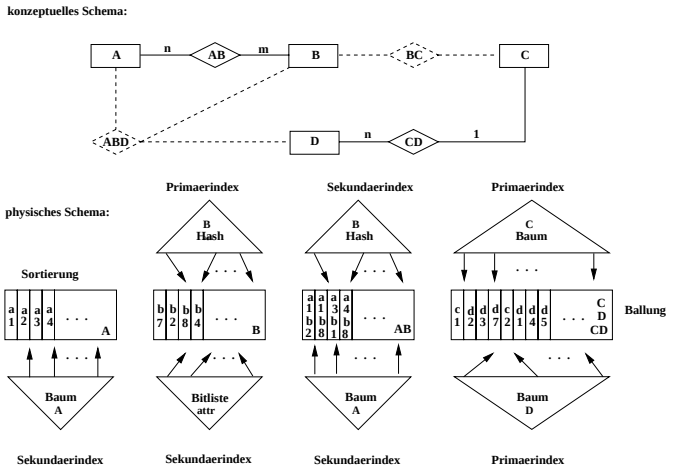


Abbildung konzeptuelles Schema – physisches Schema

Speicherhierarchie

Typ	Einsatzbereich	Zugriffsart	Zugriffszeit
Primärspeicher (Cache, Hauptspeicher)	Verarbeitung der Daten	direkt	10 - 100ns
Sekundärspeicher (Platten)	permanente Speicherung	direkt	10 msec
Tertiärspeicher (Bänder, CD)	Archivierung	sequentiell	n.a.

Die Zugriffszeit bei Platten besteht aus (Zeitangaben bzgl. eines Beispiels):

- *seek time*: Zeit um den Lesekopf zu positionieren (2.2msec bis 15.5msec),
- *rotational delay*: Zeit bis die gewünschte Seite unter dem Lesekopf auftaucht (im Durchschnitt 4.17msec),
- *transfer time*: Zeit um die Seite zu lesen (abhängig von der der Rotationszeit).

$\Rightarrow$ Zusammengehörige Seiten sollten auf demselben Zylinder abgelegt werden.

$\Rightarrow$ Nacheinander benötigte Seiten sollten hintereinander stehen und möglichst mit einem Zugriff gelesen werden (**blocked access**).

Lesen und Schreiben von Sätzen

Zu verarbeitende Sätze müssen vom Plattenspeicher in den Hauptspeicher übertragen werden:

- Der *Datei-Manager* lokalisiert die Seite des gesuchten Satzes.
- Der *Datei-Manager* beauftragt den *Puffer-Manager* die Seite in den Hauptspeicher zu übertragen.
- Der *Puffer-Manager* führt den Auftrag aus und meldet dies dem *Datei-Manager*.

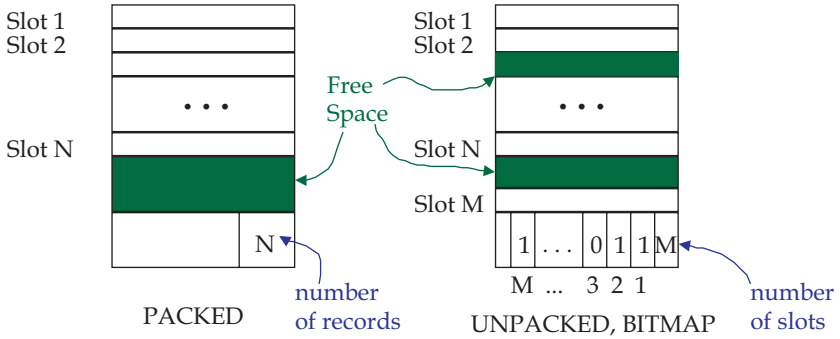
Pufferverwaltung

- Die zur Verfügung stehenden Seiten des Hauptspeichers heißen *Rahmen*. Für jeden Rahmen werden die Variablen *pin_count* und *dirty* verwaltet.
- *pin_count* zählt die Anzahl der Transaktionen, die die betreffende Seite angefordert haben und noch nicht freigegeben haben; *dirty* zeigt an, ob die Seite verändert wurde.
- Wenn eine angeforderte Seite nicht im Pool ist und kein Rahmen frei ist, dann wird eine Seite mit *pin_count = 0* für die Ersetzung bestimmt. Wenn *dirty* gesetzt ist, dann muß die Seite vor der Ersetzung in der Datenbank materialisiert werden. Existiert keine Seite mit *pin_count = 0*, muß der Puffer-Manager auf eine solche warten, bzw. die veranlassende Transaktion wird abgebrochen.
- Sind *Zugriffsmuster* der Transaktionen erkennbar, kann ein *prefetching* von Seiten praktiziert werden.

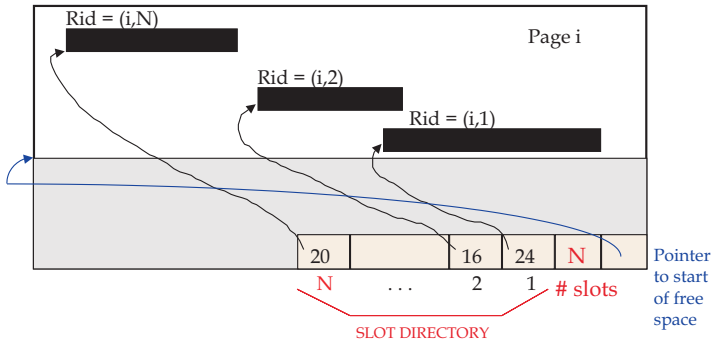
Adressierung von Sätzen

- Eine **Adresse**, bzw. Tupel-Identifikator (tid) oder Satz-Identifikator hat die Form (b, k), wobei b Seitennummer und k laufende Nummer innerhalb der Seite.
- Mittels eines **Adreßbuchs** (*directory*) der betreffenden Seite wird k eine physikalische Adresse innerhalb der Seite zugeordnet.
- Adressbuch und Sätze sollten gegeneinander wachsend in der Seite organisiert werden.
- Ein Satz ist in seiner Seite frei verschiebbar; es muß lediglich das Adressbuch aktualisiert werden.
- Sätze können gelöscht werden (mit Speicherfreigabe) durch entsprechende Kennzeichnung im Adreßbuch.

Seiten-Organisation für Sätze fester Länge:



Seiten-Organisation für Sätze fester und variabler Länge:



Bemerkungen:

- Sätze können durch **Verweisketten** über Seitengrenzen verschiebbar gemacht werden: anstatt des Satzinhaltes wird seine neue Adresse gespeichert.
- **Variabel lange Sätze** beginnen mit einem Längenzähler. Jedes Feld variabler Länge hat am Anfang einen Längenzähler. Alternativ hat jeder Satz am Anfang eine Zeigerliste, in der die Zeiger auf den Beginn des Feldes zeigen.

Datei-Organisation und Indexstrukturen

Die Effizienz der Verarbeitung der Daten hängt von der gewählten Datei-Organisation und den vorliegenden Indexstrukturen ab. Relevante Operationen:

- **Durchsuchen** (*Scan*): Alle Seiten, die Sätze enthalten, werden gelesen.
- **Suchen in Abhängigkeit von einem Gleicheitstest** (*Equality Search*): Alle Seiten, die die Sätze enthalten, die die Gleichheitsbedingung erfüllen, werden gelesen.
- **Suchen in Abhängigkeit von einem Bereichstest** (*Range Search*): analog.
- **Einfügungen** (*Insert*): Lesen und Schreiben einer oder mehrerer Seiten erforderlich.
- **Löschungen** (*Delete*): analog.

Datei-Organisation und Indexstrukturen

Die Effizienz der Verarbeitung der Daten hängt von der gewählten Datei-Organisation und den vorliegenden Indexstrukturen ab. Relevante Operationen:

- **Durchsuchen** (*Scan*): Alle Seiten, die Sätze enthalten, werden gelesen.
- **Suchen in Abhängigkeit von einem Gleicheitstest** (*Equality Search*): Alle Seiten, die die Sätze enthalten, die die Gleichheitsbedingung erfüllen, werden gelesen.
- **Suchen in Abhängigkeit von einem Bereichstest** (*Range Search*): analog.
- **Einfügungen** (*Insert*): Lesen und Schreiben einer oder mehrerer Seiten erforderlich.
- **Löschungen** (*Delete*): analog.

Organisationsformen für Dateien

Haufenorganisation: Die Datei wird ohne spezielle Organisation in einer Menge von Blöcken abgespeichert. Es wird angenommen, daß die Nummern dieser Blöcke bekannt sind (intern gehaltenes Blockverzeichnis).

Sortierung: Die Sätze in den Seiten sind sortiert. Die Anordnung der Seiten und entsprechend der Blöcke berücksichtigt die Sortierung.

Hash-Organisation: Die Datei wird in Abhängigkeit einer Hash-Funktion in Kacheln (Blöcke) aufgeteilt.

Sei n die Anzahl Seiten einer Datei.

Typ	Scan	Equality Search	Range Search	Insert	Delete
Haufen	n	$0.5n$	n	2	$0.5n+1$
Sortierung	n	$\log n$	$\log n + \#matches$	$\log n + 1$	$\log n + 1$
Hash	$1.25n$	1	$1.25n$	2	2

Indexstrukturen

- Ein **Index** zu einer Datei ist eine Hilfsstruktur, um Operationen zu beschleunigen, die aufgrund der Organisationsform der Datei nicht effizient ausgeführt werden können.
- Ein **Suchschlüssel** ist eine Feldkombination einer Datei, bzgl. der der Zugriff unterstützt werden soll.
- Ein Index ist eine Menge von **Dateneinträgen** zusammen mit einer effizienten Zugriffstechnik, um alle Dateneinträge zu einem gegebenen Suchschlüsselwert zu lokalisieren.
- Sei k ein Suchschlüsselwert, dann ist $k*$ der zugehörige Dateneintrag. $k*$ enthält genug Information, um alle Sätze mit Suchschlüsselwert k zu lokalisieren.

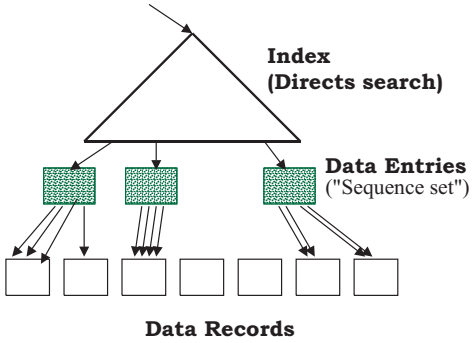
Alternativen:

$k*$ ist der eigentliche Satz (mit Suchschlüsselwert k),
$k* = [k, adr]$, wobei adr die Adresse des Satzes mit Suchschlüsselwert k ist,
$k* = [k, adr\text{-}liste]$, wobei $adr\text{-}liste$ die Liste der Adressen der Sätze mit Suchschlüsselwert k ist.

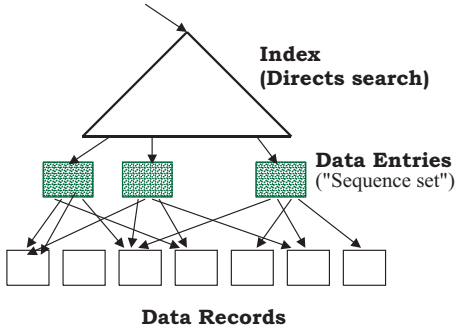
Eigenschaften von Indexstrukturen

- **geballte** oder **ungeballte** Indexstrukturen.
  Ballung bedeutet, daß logisch zusammenhängende Sätze/Seiten auch physisch zusammenhängend gespeichert werden.
- **dichte** oder **spärliche** Indexstrukturen.
  Dichte Indexstrukturen enthalten pro Satz der betreffenden Datei einen Eintrag; **spärliche** Strukturen pro Seite der Datei einen Eintrag.
  Eine Datei heißt **invertiert** bzgl. einem Feld, wenn ein dichter Sekundär-Index zu diesem Feld existiert; sie heißt **voll invertiert**, wenn zu jedem Feld, das nicht Teil des Primärschlüssels ist, ein dichter Sekundär-Index existiert.
- **Primär-** oder **Sekundär-**Indexstrukturen.
  Bei einem **Primär-Index** enthält der Suchschlüssel den Primärschlüssel; anderenfalls redet man von einem **Sekundär-Index**.
- **zusammengesetzte Suchschlüssel**.
  Ein zusammengesetzter Suchschlüssel besteht aus mehreren Feldern. Sind nicht alle Felder durch Konstante gebunden, so liegt eine **Bereichs-**Anfrage (*range query*) vor.

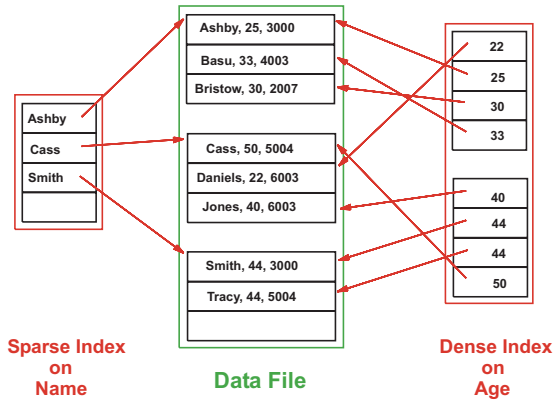
hierarchischer Index mit Ballung:



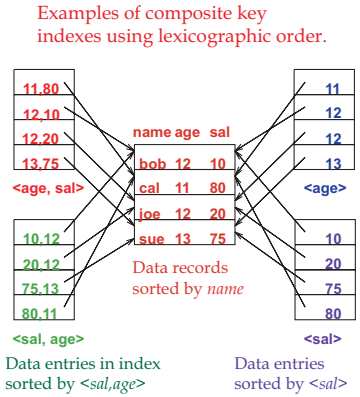
hierarchischer Index ohne Ballung:



dichter vs. spärlicher Index:



Index mit zusammengesetztem Schlüssel:



baum-basierter Index

Ein **B-Baum** der Ordnung (m, l), $m \geq 3$, $l \geq 1$, ist ein Vielweg-Suchbaum mit den folgenden Eigenschaften:

(1) Die Wurzel ist entweder ein Blatt oder hat mindestens zwei Söhne.

(2) Jeder innere Knoten, außer der Wurzel, hat mindestens $\lceil m/2 \rceil$ und höchstens m Söhne. Die garantierte Mindestspeicherplatzauslastung beträgt somit 50%.

(3) Alle Blätter befinden sich auf dem gleichen Level. Sei N die Anzahl Blätter. Für die Höhe h des Baumes gilt dann höchstens $h = 1 + \lfloor log_{m/2}(N/2) \rfloor$ und mindestens $h = \lceil log_m N \rceil$.

(4) Einfügen und Löschen kann in Zeit proportional zur Höhe des Baumes durchgeführt werden.

(5) Die inneren Knoten sind Tupel der Form $(p_0, k_1, p_1, k_2, p_2, \ldots, k_n, p_n)$, wobei $\lceil m/2 \rceil - 1 \leq n \leq m - 1$ und:

(a) p_i ist ein Zeiger auf den $i+1$-ten Sohn und jedes k_i ist ein Suchschlüsselwert, wobei $0 \leq i \leq n$.

(b) Die Suchschlüsselwerte sind geordnet, d.h. $k_i < k_j$, für $1 \leq i < j \leq n$.

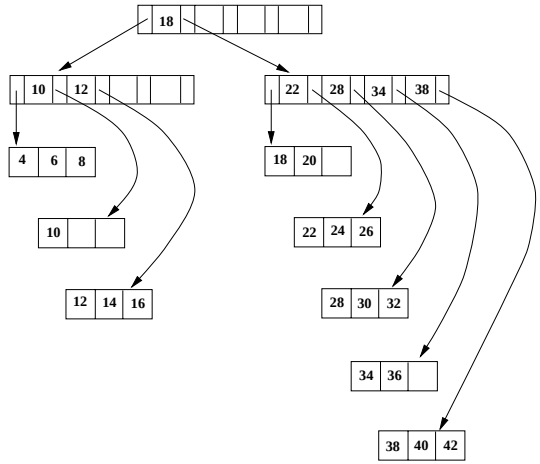
(c) Alle Suchschlüsselwerte im linken (rechten) Teilbaum von k_i sind kleiner (größer-gleich) als der Wert von k_i, für $1 \leq i \leq n$.

(6) Die Blätter sind Tupel der Form $([k_1, s_1], [k_2, s_2], \ldots, [k_g, s_g])$, wobei $g \leq l$ und k_i Suchschlüsselwert und s_i der betreffende Dateneintrag ist.

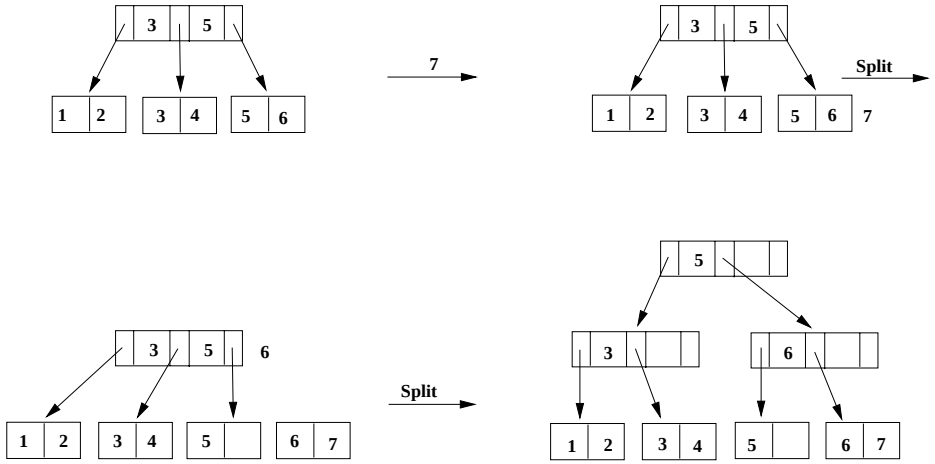
(7) Die Blätter des Baumes können verkettet sein, wenn Zugriff gemäß der Sortierfolge unterstützt werden soll.

B-Bäume eignen sich für geballte und ungeballte Indexstrukturen, für Primär- und Sekundär-Indexstrukturen, und unterstützen sowohl dichte, als auch spärliche Formen.

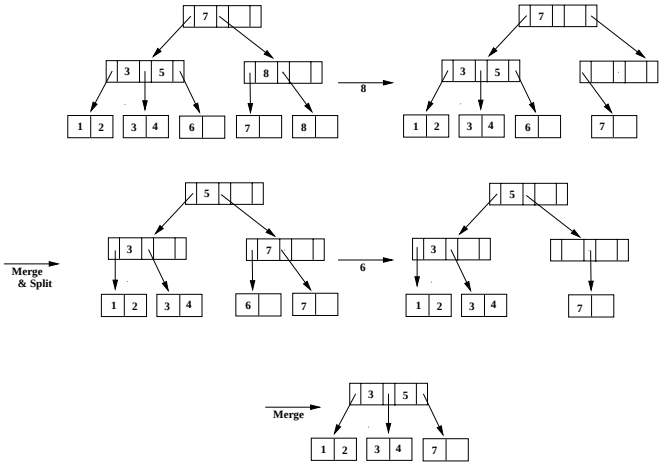
Beispiel: B-Baum der Ordnung (5,3)



Einfügen gefährdet die Ausgeglichenheit:



Löschen gefährdet die Ausgeglichenheit:



hash-basierter Index

- Eine gegebene Datei wird mittels einer Streuungsfunktion (*Hash-Funktion* h in **Kacheln** aufgeteilt. Die Kacheln seien numeriert von 0, 1, . . . , $K-1$. Eine Kachel besteht aus einer oder mehreren Seiten, die als **Buckets** bezeichnet werden. Auf der Menge der Suchschlüsselwerte v sollte $h(v)$ gleichverteilt über 0, 1, . . . , $K-1$ sein.
- **Standardtechnik:** Konvertiere den Suchschlüsselwert in eine ganze Zahl und berechne den Rest dieser Zahl modulo K, d.h. $h(v) = v \ mod \ K$.
- Die Buckets einer Kachel sind verkettet, sofern für Überlaufsätze zusätzlicher Speicherbedarf besteht.
- Hash-Verfahren zerstören i.a. Sortierungen unter den Sätzen.
- Hash-Verfahren ermöglichen einen direkten Zugriff zu den Sätzen mit 1 Externzugriff, sofern keine Überlaufsätze existieren.
- Es wurden eine Reihe von (dynamischen) Hash-Verfahren entwickelt, die die Vermeidung von Überläufern ermöglichen.

Physischer Datenbankentwurf: mehrdimensionaler Zugriff

Es soll der Zugriff zu Daten in Abhängigkeit von mehreren Attributen unterstützt werden. Eine wichtige Anwendung sind Anfragen über **räumlichen** Daten:

Punkte: Die Datenbank speichert eine Menge von Punkten in einem mehrdimensionalen Raum. Die Punkte resultieren beispielsweise aus Messungen, oder aus *bit/pixel maps* von Bilddaten (Rasterdaten), oder aus *feature vectors*, die die eigentlichen Objekte identifizieren.

Regionen: Eine Region hat eine räumliche Ausdehnung mit Ort und Begrenzung; häufig wird das eigentliche Objekt approximiert (Vektordaten) mittels Punkten, Linien, Polygonen, Kuben, etc. Typische Anwendungen sind geographische Zusammenhänge, Bildverarbeitung, Multimedia, oder auch CAD.

Bereichsanfragen: Es kann nach allen Punkten gefragt werden, die in dem angegebenen Bereich liegen, bzw. nach allen Regionen, die in ihm enthalten sind, oder auch überlappen.

partial-match-**Anfragen:** Der Suchschlüssel ist nur teilweise gegeben.

nearest neighbor-**Anfragen:** Was ist der nächste Punkt zu dem gegebenen Punkt?

Verbundanfragen: Bestimme alle Paare von Objekten, die ein gegebenes raumbezogenes Prädikat erfüllen. Beispiele für solche Prädikate sind *intersects, contains, is_enclosed_by, distance(...), adjacent, northwest, ...*

Beispiel: Rectangle(X1, Y1, X2, Y2)

(a) SELECT X1, Y1, X2, Y2 *partial-match-Anfrage*
FROM Rectangle
WHERE X1=5 AND Y1 = 6

(b) SELECT X1, Y1, X2, Y2 *Bereichsanfrage*
FROM Retangle
WHERE X1 ≤ 0 AND X2 ≥ 10 AND Y1 ≤ 0 AND Y2 ≥ 10

unbefriedigende Unterstützung bei Einsatz eindimensionaler Indexstrukturen:

- Existiere ein B-Baum für jede der Koordinaten.
  Problematisch ist der Aufwand vier Indexstrukturen zu unterhalten; es können pro Index lediglich Obermengen der Antwort bestimmt werden; es können keine Anfragen des Typs X1 $\geq$ Y2 beantwortet werden.
- Existiere ein B-Baum für eine Konkatenation der Koordinaten.
  Angenommen die Konkatenation ist X1 Y1 X2 Y2. Anfragen des Typs Y1 $\leq$ a werden nicht unterstützt; im allgemeinen muß für jede Permutation der Koordinaten ein Index aufgebaut werden.

mehrdimensionale Indexstrukturen:

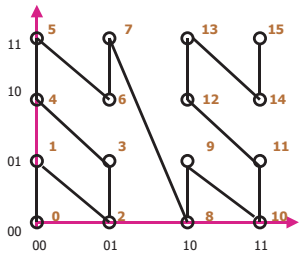
- **baumbasiert:** Quadbaum, R-Baum
- hashbasiert: s. Literatur.

räumlicher Zugriff mittels space-filling curves

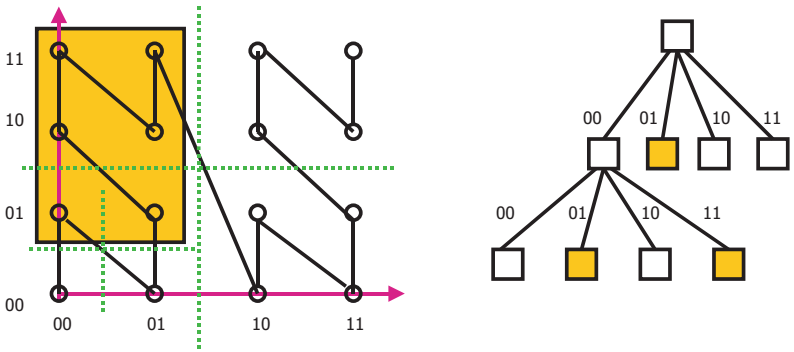
Jeder Attributwert wird durch k Bits repräsentiert. Mittels einer *space-filling* Kurve wird eine lineare Ordnung auf allen Punkten im Raum erzeugt, die die räumliche Nähe berücksichtigt. Gemäß dieser Ordnung wird ein B-Baum Index aufgebaut.

Der Z-Wert eines Punktes ergibt sich alternierend aus den Bits seines X- und Y-Wertes.

Beispiel, Z-ordering von Punkten:



Beispiel, Z-ordering von Regionen und Quad-Baum:

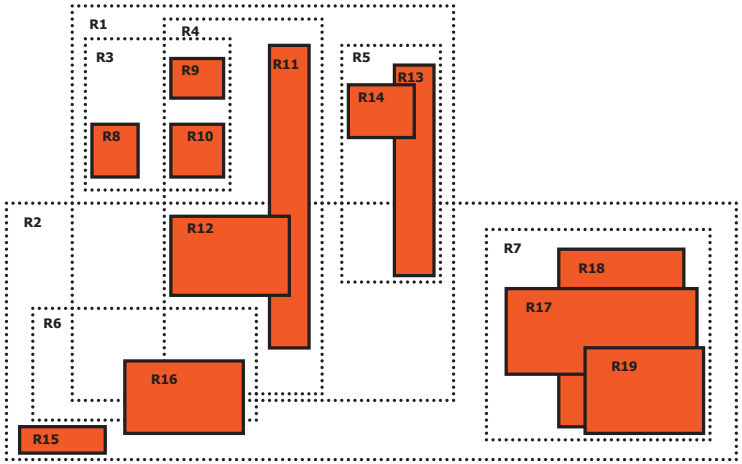


Jeder Teilbaum enthält alle Punkte seines Quadranten.

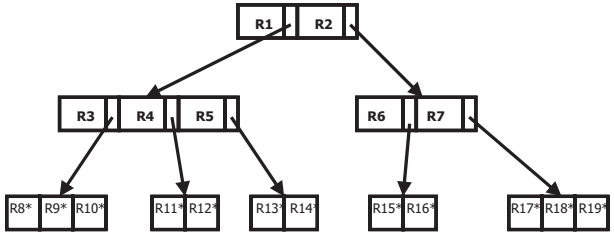
räumlicher Zugriff mittels R-Bäumen

- R-Bäume adaptieren die Ideen von B-Bäumen, verwenden jedoch als Suchschlüssel Intervalle: jeder Suchschlüsselwert entspricht einem durch die Intervalle definierten Körper, dessen Begrenzungslinien parallel zu den Koordinatenachsen laufen (*bounding box*).
- Ein Dateneintrag besteht aus einem Paar *(n-dim box, rid)*, wobei *rid* ein Objekt identifiziert und *n-dim box* die kleinste bounding box, die das Objekt enthält. Ein Punkt ist ein Spezialfall eines Körpers.
- Jedes Objekt befindet sich in genau einem Blatt des Baumes - die bounding boxes der inneren Knoten können überlappen.
- Beim Aufbau des Baumes müssen die bounding boxes möglichst überlappungsfrei berechnet werden. Beim Einführen eines Vorgängerknotens ergibt sich dessen bounding box als kleinste bounding box, die alle boxes seiner Nachfolger enthält.
- Im Unterschied zum Quad-Baum orientiert sich ein R-Baum an den existierenden Daten.

Beispiel, einige Objekte mit bounding boxes:



Beispiel, zugehöriger R-Baum:



Physischer Datenbankentwurf: dynamische Hash-Verfahren

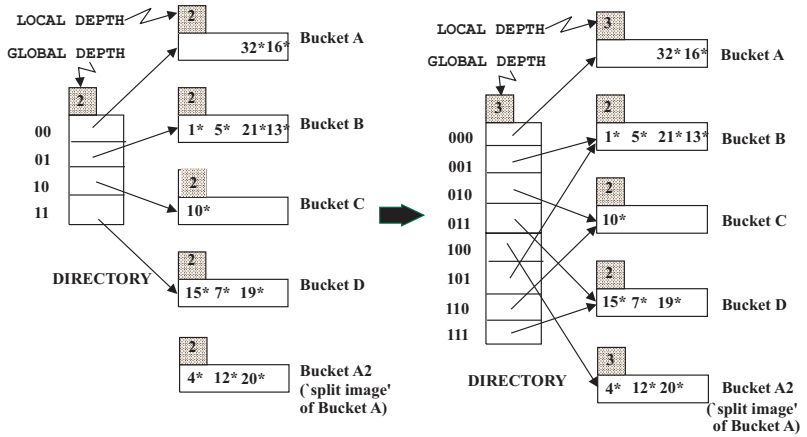
erweiterbares Hashing

- Die Hashfunktion bildet Suchschlüssel auf ein Kachelverzeichnis ab, das auf die eigentlichen Buckets verweist.
- Tritt eine Überlaufsituation auf, so wird der überlaufende Bucket gesplittet indem ein zusätzlicher Bucket hinzugenommen wird, auf den die bisherigen Sätze des überlaufenden Buckets und der neue Satz verteilt werden. Die beiden Buckets bezeichnen wir auch als **Split-Partner**.

  Das Kachelverzeichnis wird verdoppelt:

  Sei K das Kachelverzeichnis vor und K' das Kachelverzeichnis nach der Verdopplung. Sei k die Anzahl Einträge in K und sei i der übergelaufene Bucket.
  - $K'[i] = K[i]$ und $K'[i+k]$ verweist auf den neu hinzugenommenen Bucket.
  - $K'[j] = K[j+k]$ für $1 \leq j \leq 2k, j \neq i$.

Einfügen von Satz 20:



- Die Resultate der Hash-Funktion $h(v)$ werden als Binärzahl interpretiert und die letzten d Bits werden als Offset für das Verzeichnis verwendet. Die aktuelle Größe k des Verzeichnises muß die Bedingung $k = 2^d$ erfüllen.
- d ist die **globale Tiefe** der gestreuten Datei; sie wird im Header der Datei gespeichert.
- Im allgemeinen werden zu einem Zeitpunkt Buckets existieren, die aus unterschiedlich vielen Splitvorgängen hervorgegangen sind. Zur Kennzeichnung erhält jeder Bucket eine **lokale Tiefe**; die lokale Tiefe wird bei jedem Splitvorgang eines Buckets um 1 erhöht; sie ist höchstens so hoch wie die globale Tiefe der Datei.
- Enthält ein Bucket verursacht durch Löschungen keinen Satz mehr, so kann er freigegeben werden. Der bisherige Zeiger auf ihn aus dem Kachelverzeichnis wird auf seinen Split-Partner umgelegt, sofern ein solcher noch existiert.
- Gilt für ein Kachelverzeichnis $K[i] = K[2^{d-1} + i]$, $0 \leq i \leq d$, dann kann das Kachelverzeichnis auf die Hälfte reduziert werden und die globale Tiefe um 1 verringert.

- Erweiterbares Hashing erlaubt einen direkten Zugriff zu den Sätzen mit 1 Externzugriff, sofern das Kachelverzeichnis im Hauptspeicher gehalten werden kann und nicht zu viele Kollisionen auftreten.
- Kollisionen können auftreten, wenn mehr Sätze mit **denselben** Hashwerten auftreten, als in den betreffenden Bucket hineinpassen. Es werden dann Überlaufseiten eingeführt.
- Im Falle von nicht gleichverteilten Hashwerten kann das Kachelverzeichnis unverhältnismäßig groß werden.

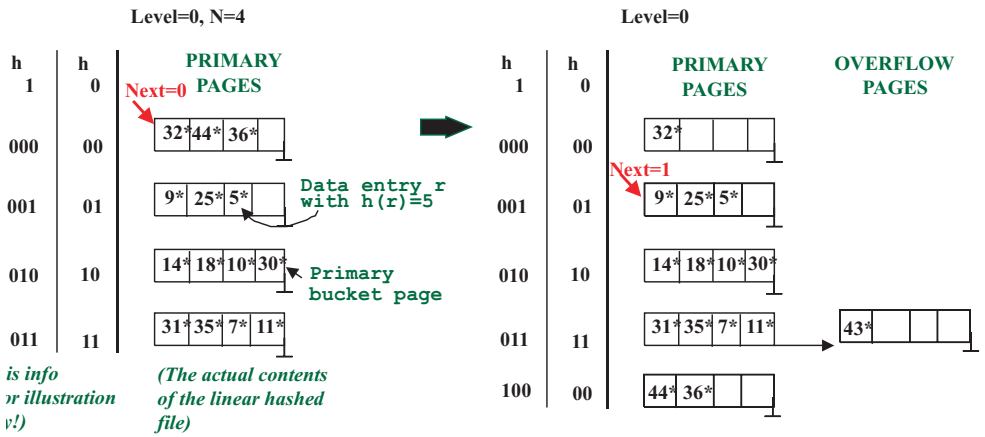
lineares Hashing

- Lineares Hashing benötigt kein Kachelverzeichnis.
- Es wird eine Familie von Hash-Funktionen $h_0, h_1, h_2, \ldots$ verwendet, wobei der Wertebereich von h_i gerade doppelt so groß ist, wie der von h_{i-1}, $i \geq 1$.
- Typischerweise sind die einzelnen Hash-Funktionen wie folgt definiert:

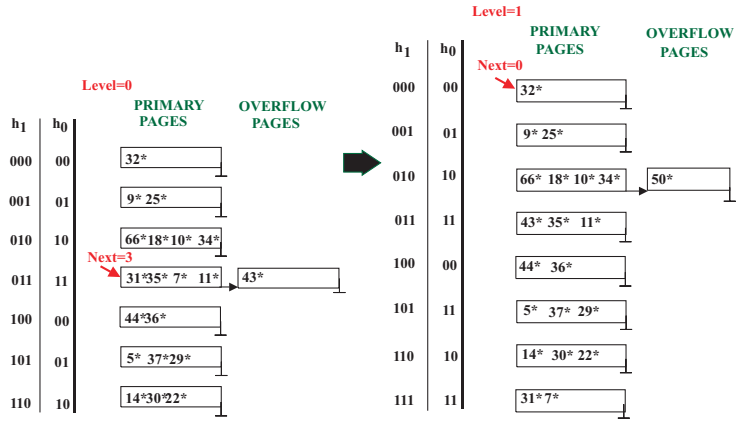
  $$h_i(v) = h(v) mod (2^i N),$$

  wobei h eine Hash-Funktion, die Suchschlüsselwerte auf ganze Zahlen abbildet und N die initiale Anzahl Buckets. N ist eine Potenz von 2.
- Sei $h(v)$ in Binärdarstellung gegeben. $h_i(v)$ ergibt sich dann aus den letzten d_i Bits von $h(v)$, wobei $d_i = d_0 + i$ und d_0 die für N benötigte Anzahl Bits.
- Buckets werden gemäß **Runden** gesplittet. Innerhalb einer Runde werden die Buckets beginnend mit dem ersten der Reihe nach gesplittet. Nach Ende einer Runde hat sich die Anzahl Buckets verdoppelt. Der als nächstes zu splittende Bucket $next$ wird gesplittet, wenn in irgendeinem Bucket ein Überlauf auftritt, oder wenn eine spezielle Bedingung erfüllt ist.
- Bezeichne $level$ die aktuelle Runde. Es werden gerade die Hash-Funktionen $h_{level}, h_{level+1}$ benutzt.

Einfügen von Satz 43:



Einfügen von Satz 50:



- Zu gegebenem v kann der zugehörige Bucket durch Anwendung von h_{level} und $h_{level+1}$ lokalisiert werden. Gilt für $b = h_{level}(v)$ gerade $next \leq b \leq 2^{level}N$, dann ist Bucket b der zugehörige Bucket. Anderenfalls wende $h_{level+1}$ an.
- Wird der letzte Bucket aufgrund von Löschungen leer, so kann er freigegeben werden; $next$ wird um 1 verringert sofern $next > 0$, bzw., sofern $next = 0$, wird $level$ um 1 verringert und $next$ auf $2^{level}N - 1$ gesetzt.
- Lineares Hashing erlaubt einen direkten Zugriff zu den Sätzen mit 1 Externzugriff, sofern keine Überläufer existieren.
- Im Falle von nicht gleichverteilten Hashwerten können im schlechtesten Fall linear in der Anzahl Sätze viele Zugriffe erforderlich werden.

Physischer Datenbankentwurf: Sortieren

Annahme: die zu sortierende Datei paßt nicht komplett in den Hauptspeicher.

Es wird eine Variante des Mergesort verwendet. Es werden zunächst Teilfolgen sortiert und dann mittels Mischen zusammengeführt.

Seien B Pufferseiten verfügbar und sei N die Anzahl Seiten der Datei.

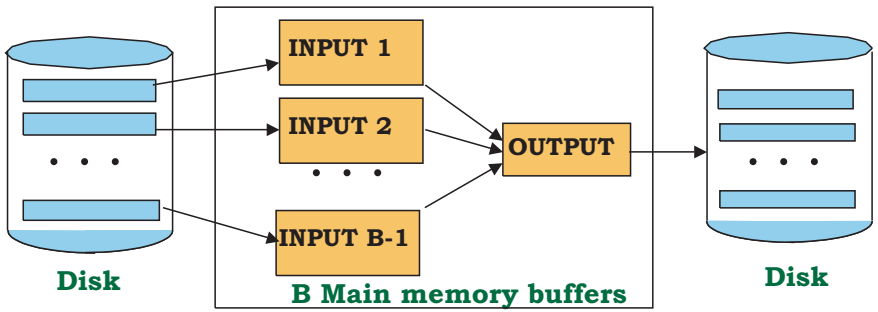
Durchlauf 0: Es werden jeweils B Seiten gelesen, sortiert und wieder ausgegeben. Es werden $\lceil N/B \rceil$ sortierte Teilfolgen (**Runs**) ausgegeben.

Durchlauf $i = 1, 2, \ldots$**:** Verwendet $B - 1$ Puffer-Seiten für die Eingabe und die verbleibende für die Ausgabe. Mische $B - 1$ Runs zu einem neuen Run.

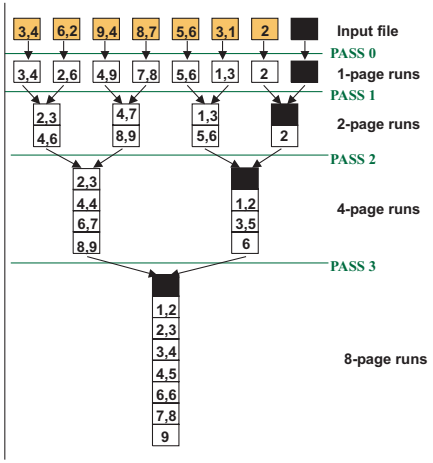
Es werden $\lceil \log_{B-1} \lceil \frac{N}{B} \rceil \rceil + 1$ Durchläufe benötigt; in jedem Durchlauf werden alle N Seiten gelesen und geschrieben. Anzahl Seitenzugriffe:

$$(\lceil \log_{B-1} \lceil \frac{N}{B} \rceil \rceil + 1) 2N$$

externer Merge-Sort mit B Puffer-Seiten:



Beispiel:



Im 0-ten Durchlauf werden in diesem Beispiel N Runs erzeugt.

Beobachtung:

Die Seiten eines Runs werden immer nacheinander benötigt. Kann hier ein blockweiser Zugriff effizienzsteigernd wirken?

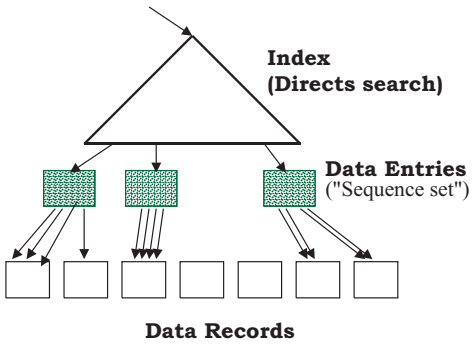
Sei b die Anzahl Seiten pro Block. Anzahl Block-Zugriffe in Abhängigkeit von b:

$$(\lceil \log_{\frac{B}{b}-1} \lceil \frac{Nb}{B} \rceil \rceil + 1) 2\frac{N}{b}$$

Eine Vergrößerung des Blockungsfaktors bringt also eine Verringerung des Reduktionsfaktors mit sich.

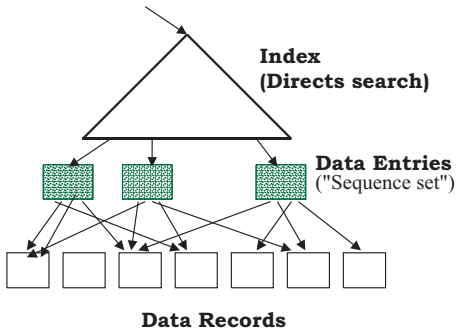
In der Praxis genügen typischerweise 2 Durchläufe. Angenommen Seitengröße 4KB, Puffergröße 50.000 Seiten (< 256 MB) und Blockungsfaktor $32 \Longrightarrow 1561$ Runs werden pro Durchlauf gemischt!

Zugriff in Sortierfolge eines geballten B-Baums:



In der Regel erheblich effizienter als separates Sortieren der Datei, da sich in den Seiten in der Sortierfolge aufeinanderfolgende Sätze befinden.

Zugriff in Sortierfolge eines nicht geballten B-Baums:



Separates Sortieren ist in der Regel effizienter, da sich in der Sortierfolge aufeinanderfolgende Sätze möglicherweise in unterschiedlichen Seiten befinden.

Transaktionsverwaltung

Transaktionen erfüllen die **ACID**-Eigenschaften:

Atomicity: Eine Transaktion ist (logisch) eine nicht weiter zerlegbare Einheit; die durch sie bewirkten Änderungen eines DB-Zustandes sind atomar und werden entweder vollständig oder gar nicht vorgenommen.

Concistency: Eine Transaktion bewirkt einen korrekten Zustandsübergang in der DB; insbesondere ist sie ein Prozeß eines korrekt programmierten Programms.

Isolation: Im allgemeinen wird eine Menge von Transaktionen gleichzeitig in ihrer Ausführungsphase sein (Mehrbenutzerbetrieb). Obwohl somit die DB-Zugriffe der einzelnen Transaktionen potentiell verzahnt ablaufen, bleibt dies für den Benutzer verborgen (*simulierter* Einbenutzerbetrieb).

Durability: Wenn eine Transaktion erfolgreich ihr Ende erreicht hat, dann sind alle von ihr verursachten Änderungen dauerhaft (persistent), d.h. sie überdauern alle möglichen Fehlersituationen der DB.

Die elementaren **Aktionen** einer Transaktion sind

- Lesezugriffe: READ
  Mittels READ A (RA) wird der Wert eines DB-Objektes A aus der DB in den lokalen Arbeitsbereich der Transaktion übertragen.
- Schreibzugriffe: WRITE
  Mittels WRITE A (WA) wird der Wert eines DB-Objektes A aus dem lokalem Arbeitsbereich der Transaktion in die DB übertragen.
- BEGIN WORK und COMMIT WORK zur Kennzeichnung ihres Beginns (BOT) und ihres erfolgreichen Abschlusses (EOT).
- ROLLBACK WORK zum Selbstabbruch mit Rückgängigmachung aller bewirkten Zustandsänderungen (ABORT).

Die Elementaroperationen werden als (physisch) atomar betrachtet. D.h., zu einem Zeitpunkt wird höchstens eine Aktion ausgeführt (keine Verzahnung der Teilschritte von Aktionen).

typische Situation:

> **Im allgemeinen befinden sich zu einem Zeitpunkt mehrere Transaktionen in der Ausführungsphase.**

Im Unterschied zu **seriellen** Transaktionen redet man von **nebenläufigen**, oder auch **parallelen** Transaktionen und meint damit **verzahnt** ablaufende Transaktionen.

Ziel einer Mehrbenutzerkontrolle: Es soll möglichst viel Parallelität (also viele Verzahnungen der Transaktionen) möglich sein, ohne daß durch die Parallelität fehlerhafte Ergebnisse entstehen können.

Problem: Im allgemeinen laufen die Transaktionen auf gemeinsamen Daten ab; dies gefährdet die Korrektheit der Ergebnisse.

Typische Fehlersituationen

Verlorengegangene Änderung (lost update)

Gehaltsänderung	**Gehaltsänderung**
SELECT gehalt INTO :gehalt FROM Pers WHERE pnr=2345	
	SELECT gehalt INTO :gehalt FROM Pers WHERE pnr=2345
gehalt := gehalt + 2000;	
	gehalt := gehalt + 1000;
UPDATE Pers SET gehalt= :gehalt WHERE pnr = 2345	
	UPDATE Pers SET gehalt= :gehalt WHERE pnr = 2345

Schmutziges Lesen (dirty read)

Gehaltsänderung	**Gehaltsänderung**
UPDATE Pers SET gehalt = gehalt + 1000 WHERE pnr = 2345	
	SELECT gehalt INTO :gehalt FROM Pers WHERE pnr=2345
	gehalt := gehalt * 1.05;
	UPDATE Pers SET gehalt= :gehalt WHERE pnr = 2345
ABORT	

Inkonsistente Analyse (non-repeatable read)

Gehaltssumme berechnen	**Gehaltsänderung**
SELECT gehalt INTO :gehalt FROM Pers WHERE pnr=2345 summe := summe + gehalt	
	UPDATE Pers SET gehalt= gehalt +1000 WHERE pnr = 2345
	UPDATE Pers SET gehalt= gehalt +2000 WHERE pnr = 3456
SELECT gehalt INTO :gehalt FROM Pers WHERE pnr=3456 summe := summe + gehalt	

Phantom-Problem (phantom)

Gehaltssumme überprüfen	**Einfügen eines neuen Angestellten**
SELECT SUM(gehalt) INTO :summe FROM Pers WHERE abtnr = 17	
	INSERT INTO Pers (pnr, abtnr, gehalt) VALUES (4567, 17, 55.000)
	UPDATE Pers SET gehaltsumme = gehaltssumme + 55.0 WHERE abtnr = 17
SELECT gehaltssumme INTO :gsumme FROM Abt WHERE abtnr = 17	
IF gsumme <> summe THEN <Fehlerbehandlung>	

Serialisierbarkeit

- Ein **Schedule** bzgl. einer Menge von Transaktionen ist eine Verzahnung ihrer Aktionen, die die Reihenfolge der Aktionen einer jeden Transaktion erhält.
- Ein Schedule heißt **seriell**, wenn die Aktionen der einzelnen Transaktionen jeweils direkt aufeinander folgen.

Transaktionen T und Schedule S werden durch die Folge ihrer READ- und WRITE-Aktionen dargestellt; Zuordnung Aktionen zu Transaktionen innerhalb eines Schedules durch geeignete Indizierung.

Beispiel:

$T_1 = RA\ WA\ RB\ WB,$
$T_2 = RA\ WA\ RB\ WB.$

$$S_1 = R_1A\ W_1A\ R_1B\ W_1B\ R_2A\ W_2A\ R_2B\ W_2B$$
$$S_2 = R_1A\ R_2A\ W_1A\ W_2A\ R_1B\ R_2B\ W_1B\ W_2B$$
$$S_3 = R_1A\ R_2A\ W_1A\ R_1B\ W_2A\ R_2B\ W_1B\ W_2B$$

Serialisierbarkeit

- Ein **Schedule** bzgl. einer Menge von Transaktionen ist eine Verzahnung ihrer Aktionen, die die Reihenfolge der Aktionen einer jeden Transaktion erhält.
- Ein Schedule heißt **seriell**, wenn die Aktionen der einzelnen Transaktionen jeweils direkt aufeinander folgen.

Transaktionen T und Schedule S werden durch die Folge ihrer READ- und WRITE-Aktionen dargestellt; Zuordnung Aktionen zu Transaktionen innerhalb eines Schedules durch geeignete Indizierung.

Beispiel:

$T_1 = RA\ WA\ RB\ WB,$
$T_2 = RA\ WA\ RB\ WB.$

$$S_1 = R_1A\ W_1A\ R_1B\ W_1B\ R_2A\ W_2A\ R_2B\ W_2B$$
$$S_2 = R_1A\ R_2A\ W_1A\ W_2A\ R_1B\ R_2B\ W_1B\ W_2B$$
$$S_3 = R_1A\ R_2A\ W_1A\ R_1B\ W_2A\ R_2B\ W_1B\ W_2B$$

Korrektheitskriterium für parallele Transaktionen

Ein Schedule heißt serialisierbar genau dann, wenn zu ihm ein äquivalenter serieller Schedule derselben Transaktionen existiert. □

Problem: Was ist ein geeigneter Äquivalenzbegriff?

Zwei Schedule S, S' (derselben Transaktionsmenge) heißen genau dann **äquivalent**, wenn für jeden Startzustand und jede Interpretation der Schreibaktionen

(1) die Transaktionen bzgl. S und S' jeweils dieselben Werte lesen,
(2) S und S' dieselben Endzustände erzeugen.

□

© Professor Dr. Georg Lausen, 2002: Vorlesung Datenbanken Serialisierbarkeit.2

Korrektheitskriterium für parallele Transaktionen

Ein Schedule heißt serialisierbar genau dann, wenn zu ihm ein äquivalenter serieller Schedule derselben Transaktionen existiert. □

Problem: Was ist ein geeigneter Äquivalenzbegriff?

Zwei Schedule S, S' (derselben Transaktionsmenge) heißen genau dann **äquivalent**, wenn für jeden Startzustand und jede Interpretation der Schreibaktionen

(1) die Transaktionen bzgl. S und S' jeweils dieselben Werte lesen,
(2) S und S' dieselben Endzustände erzeugen.

□

Beispiel:

T_1: RA; A:=A-10; WA; RB; B:=B+10; WB T_2: RA; A:=A-20; WA; RB; B:=B+20; WB A=B=10;

T_1	T_2
RA	
A:=A-10	
WA	
RB	
B:=B+10	
WB	
	RA
	A:=A-20
	WA
	RB
	B:=B+20
	WB

seriell
A=-20, B=40
A+B=20

T_1	T_2
RA	
A:=A-10	
WA	
	RA
	A:=A-20
	WA
RB	
B:=B+10	
WB	
	RB
	B:=B+20
	WB

nicht serialisierbar
A=-10, B=30
A+B=20

T_1	T_2
RA	
A:=A-10	
WA	
	RA
	A:=A-20
	WA
	RB
	B:=B+20
	WB
RB	
B:=B+10	
WB	

serialisierbar?
A=-20,B=40
A+B=20

T_1	T_2
RA	
A:=A-10	
WA	
	RA
	A:=A-20
	WA
RB	
B:=B+20	
WB	
	RB
	B:=B+10
	WB

serialisierbar
A=-20,B=40
A+B=20

Die Serialisierbarkeit eines Schedules soll allein aufgrund der Folge der Aktionen der Transaktionen entschieden werden.

Formalisierung der Semantik einer Transaktion:

Sei D der Wertebereich der betrachteten DB-Objekte.

Sei WA eine Schreibaktion einer Transaktion T und seien $RB_1, \ldots, RB_k$ $k \geq 0$ die in T vor WA ausgeführten Leseaktionen.

Der mittels WA erzeugte Wert von A ergibt sich dann zu

$$f_{T,A}(B_1, \ldots, B_k),$$

wobei

$$f_{T,A}: \ D^k \longrightarrow D$$

eine WA in T zugeordnete Funktion.

$f_{T,A}$ bezeichnen wir als die **Interpretation** der Schreibaktion WA von T.

Beispiel:

Sei der Startzustand der DB gegeben durch A_0, B_0. Seien T_1, T_2 Transaktionen wie folgt:

$$T_1 = RA\ WA\ RB\ WB;$$
$$T_2 = RA\ WA\ RB\ WB$$

Seien S_1, S_2, S_3, S_4 Schedule wie folgt:

$$S_1 = R_1A\ W_1A\ R_1B\ W_1B\ R_2A\ W_2A\ R_2B\ W_2B$$
$$S_2 = R_1A\ R_2A\ W_1A\ W_2A\ R_1B\ R_2B\ W_1B\ W_2B$$
$$S_3 = R_1A\ W_1A\ R_2A\ W_2A\ R_2B\ W_2B\ R_1B\ W_1B$$
$$S_4 = R_1A\ W_1A\ R_2A\ W_2A\ R_1B\ W_1B\ R_2B\ W_2B$$

Schedule S_2, Schedule S_3 sind nicht serialisierbar; Schedule S_4 ist serialisierbar.

Schedule T_1T_2		Schedule T_2T_1	
$T_1:$		$T_2:$	
RA	A_0	RA	A_0
WA	$f_{T_1,A}(A_0)$	WA	$f_{T_2,A}(A_0)$
RB	B_0	RB	B_0
WB	$f_{T_1,B}(A_0, B_0)$	WB	$f_{T_2,B}(A_0, B_0)$
$T_2:$		$T_1:$	
RA	$f_{T_1,A}(A_0)$	RA	$f_{T_2,A}(A_0)$
WA	$f_{T_2,A}(f_{T_1,A}(A_0))$	WA	$f_{T_1,A}(f_{T_2,A}(A_0))$
RB	$f_{T_1,B}(A_0, B_0)$	RB	$f_{T_2,B}(A_0, B_0)$
WB	$f_{T_2,B}(f_{T_1,A}(A_0), f_{T_1,B}(A_0, B_0))$	WB	$f_{T_1,B}(f_{T_2,A}(A_0), f_{T_2,B}(A_0, B_0))$

Sei S ein Schedule. Der Schedule $\hat{S} = T_0\ S\ T_\infty$ ist der **augmentierte Schedule** zu S. T_0, T_∞ sind zwei Transaktionen, die nicht in S enthalten sind. T_0 erzeugt den Startzustand, T_∞ liest den Endzustand des Schedules S.

- T_0 ist eine Transaktion, die gerade zu jedem Objekt eine Schreibaktion ausführt, zu dem eine Transaktion in S eine Lese- oder Schreibaktion ausführt.
- T_∞ ist eine Transaktion, die gerade zu jedem Objekt eine Leseaktion ausführt, zu dem eine Transaktion in S eine Lese- oder Schreibaktion ausführt.

Sei S ein Schedule. Der **D-Graph** (engl. *dependency graph*) von S ist ein gerichteter Graph $DG(S) = (V, E)$, wobei V die Menge aller Aktionen in $\hat{S}$ und E die Menge der Kanten gemäß den folgenden Bedingungen ($i \neq j$):

- $\hat{S} = \ldots R_iB \ldots W_iA \ldots \Rightarrow R_iB \rightarrow W_iA \in E$
- $\hat{S} = \ldots W_iA \ldots R_jA \ldots \Rightarrow W_iA \rightarrow R_jA \in E$, sofern zwischen W_iA und R_jA in $\hat{S}$ keine weitere Schreibaktion zu A existiert.

Satz: Zwei Schedule S, S' sind genau dann äquivalent, wenn sie dieselben Transaktionen enthalten und $DG(S) = DG(S')$ gilt. □

Sei S ein Schedule. Der **C-Graph** (engl. *conflict graph*) von S ist ein gerichteter Graph $CG(S) = (V, E)$, wobei V die Menge aller Transaktionen in $\hat{S}$ und E die Menge der Kanten gemäß den folgenden Bedingungen (jeweils $i \neq j$):

- $\hat{S} = \ldots W_iA \ldots R_jA \ldots \Rightarrow T_i \rightarrow T_j \in E$, sofern zwischen W_iA und R_jA in $\hat{S}$ keine weitere Schreibaktion zu A existiert. **(WR-Konflikt)**
- $\hat{S} = \ldots W_iA \ldots W_jA \ldots \Rightarrow T_i \rightarrow T_j \in E$, sofern zwischen W_iA und W_jA in $\hat{S}$ keine weitere Schreibaktion zu A existiert. **(WW-Konflikt)**
- $\hat{S} = \ldots R_iA \ldots W_jA \ldots \Rightarrow T_i \rightarrow T_j \in E$, sofern zwischen R_iA und W_jA in $\hat{S}$ keine weitere Schreibaktion zu A existiert. **(RW-Konflikt)**

Satz: Ein Schedule S ist serialisierbar, wenn $CG(S)$ zyklenfrei ist.

Beweisidee: Da $CG(S)$ zyklenfrei ist, existiert eine topologische Sortierung der Knoten, d.h. der Transaktionen. Sei S' der serielle Schedule gemäß einer solchen Sortierung. Zeige dann $DG(S) = DG(S')$. □

Es gibt jedoch serialisierbare Schedule, deren C-Graph nicht zyklenfrei ist:

Beispiel:

$$S: \begin{array}{ll} T_1: & \quad\quad\quad\quad R(X) \quad\quad W(Y) \\ T_2: & \quad\quad\quad\quad\quad\quad R(Y) \quad\quad\quad W(Y) \\ T_3: & R(Z)\ \ W(Y) \end{array}$$

Sei $S = R_3(Z)\ W_3(Y)\ S'$.

Dann gilt:

- S ist serialisierbar. Der zu S äquivalente serielle Schedule ist :

$$S_0 = T_1T_3T_2$$

- S' ist nicht serialisierbar.

Ein Schedule heißt **C-serialisierbar** (konflikt-serialisierbar) genau dann, wenn sein C-Graph zyklenfrei ist.

Transaktionsverwaltung: Scheduler

Der **Scheduler** eines Datenbanksystems ist ein Modul, das sicherstellt, daß nur serialisierbare Schedule zur Ausführung gelangen.

Ein Scheduler ist formal eine Abbildung Φ, die eine (Eingabe-) Folge der Aktionen einer Menge von Transaktionen S_I in eine serialisierbare auszuführende (Ausgabe-) Folge von Aktionen der Transaktionen S_O transformiert:

$$\Phi(S_I) = S_O,$$

wobei S_I, S_O Schedule. S_O muß nicht notwendigerweise alle Transaktionen in S_I enthalten.

Sei S ein Schedule. Eine Transaktion T' heißt **abhängig** von einer Transaktion T, wenn in S entweder T' einen von T erzeugten Wert, oder einen Wert, der von einer von T abhängigen Transaktion erzeugt wurde, gelesen hat.

(1) **2-Phasen Sperren** (s. später). Eine Aktion wird nur dann ausgeführt, wenn ihrer Transaktion ein entsprechendes Privileg zugewiesen wurde.

(2) **Überwachen des Konflikt Graphen:**
Sei S der aktuelle (teilweise) Schedule und sei $action$ die nächste Aktion einer Transaktion T.
Falls $CG(S\ action)$ zyklenfrei, dann führe $action$ aus. Anderenfalls breche T und alle von T abhängigen Transaktionen ab und streiche die entsprechenden Aktionen dieser Transaktionen in S.

(3) **Zeitmarken:**
Jeder Transaktion T wird bei ihrem Beginn eine eindeutige Zeitmarke $Z(T)$ zugewiesen.
Sei S der aktuelle (teilweise) Schedule und sei $action$ die nächste Aktion einer Transaktion T.
Falls für alle Transaktionen T', die eine zu $action$ in Konflikt stehende Aktion in S ausgeführt haben, gerade $Z(T') \leq Z(T)$, dann führe $action$ aus. Anderenfalls breche T und alle von T abhängigen Transaktionen ab und streiche die entsprechenden Aktionen dieser Transaktionen in S.

(4) **Optimistische Verfahren:**
Sei S der aktuelle (teilweise) Schedule. Eine Transaktion T heißt **aktiv** in S, wenn eine Aktion von T in S enthalten ist und T noch nicht ihr Ende erreicht hat.
Sei desweiteren $readset(T), writeset(T)$ die Menge der von einer Transaktion T gelesenen, bzw. geschriebenen Objekte.
Sei schließlich $action$ die nächste zu berücksichtigende Aktion und sei T die betreffende Transaktion.
Führe $action$ aus. Falls $action$ die letzte Aktion von T ist, dann breche T und alle von T abhängigen Transaktionen ab und streiche die entsprechenden Aktionen in S, sofern bzgl. einer anderen aktiven Transaktion T' gilt:

- $readset(T) \cap writeset(T') \neq \emptyset$,
- $writeset(T) \cap writeset(T') \neq \emptyset$,
- $writeset(T) \cap readset(T') \neq \emptyset$.

Beispiel: $T_1 = RA\ WB$, $T_2 = RA\ WA$, $T_3 = RB\ WB$.

$$S_I = R_1A\ R_2A\ W_2A\ R_3B\ W_3B\ W_1B$$

Konfliktgraph-Überwachen (CG):

$$\Phi_{CG}(S_I) = S_I.$$

Zeitmarkenordnung (TO):

$$\Phi_{TO}(S_I) = R_2A\ W_2A\ R_3B\ W_3B.$$

optimistisches Verfahren (Opt):

$$\Phi_{Opt}(S_I) = R_1A\ R_3B\ W_3B\ W_1B.$$

striktes 2-Phasen-Sperren (strict2PL):

$$\Phi_{strict2PL}(S_I) = R_1A\ R_3B\ W_3B\ W_1B\ R_2A\ W_2A.$$

Transaktionsverwaltung: Sperrverfahren

- Eine erworbene Sperre zu einem Objekt bedeutet Zugriffsprivilegien zu diesem Objekt.
- Bevor eine Transaktion eine Aktion ausführen darf, muß sie ein entsprechendes Privileg für das betreffende Objekt erhalten haben.

LOCK X (LX): Bewerbung um Privileg für X.
UNLOCK X (UX): Freigabe des Privilegs für X.

- Privilegien können lesenden, bzw. zusätzlich schreibenden Zugriff zu einem Objekt erlauben. Es werden entsprechend unterschiedliche Sperren verwendet:
  - Lese- und Schreibprivileg: WLOCK ($L_W X$)
  - Leseprivileg: RLOCK ($L_R X$)
- Lock- und Unlock Operationen werden bei Bedarf wie Aktionen behandelt und in die Folge der Aktionen einer Transaktion, bzw. eines Schedules mit aufgenommen.
- Jede Aktion einer Transaktion ist durch ein entsprechendes Paar (Bewerbung/LOCK, Freigabe/UNLOCK) umgeben. Pro Transaktion und Objekt genau ein Paar.
- Falls eine Transaktion bzgl. desselben Objektes eine Lese- und Schreibaktion ausführt, müssen beide Aktionen von demselben Paar WLOCK, UNLOCK umgeben sein.

- Privilegien werden zugeteilt gemäß einer sogenannten **Kompatibilitätsmatrix**:

  J: angefordertes Privileg kann erteilt werden
  N: angefordertes Privileg kann nicht erteilt werden

gehaltenes Privileg:

angefordertes Privileg:	LOCK
LOCK	N

Die Privilegien beziehen sich jeweils auf dasselbe Objekt.

- Bei Unterscheidung zwischen Lese- und Schreibprivilegien:

	RLOCK	WLOCK
RLOCK	J	N
WLOCK	N	N

- Es ist möglich, daß eine Transaktion sich um ein Privileg bewirbt, es aber nie bekommt, weil andere Transaktion vorgezogen werden (**Livelock**). Abhilfe: first-come-first-served Strategie
- Während des Ablaufs von Transaktionen können bei Verwendung von Sperren **Verklemmungen (Deadlocks)** auftreten.

T_1: LOCK A; LOCK B; UNLOCK A,B; T_2: LOCK B; LOCK A; UNLOCK A,B;

T_1: LOCK A;
T_2: LOCK B;
↑
Deadlock

Vermeiden/Auflösen von Deadlocks:

- Jede Transaktion bewirbt sich zu Beginn um alle benötigten Privilegien auf einmal (in einer atomaren Aktion).
- Auf den Objekten wird eine lineare Ordnung definiert; die jeweiligen Privilegien einer Transaktion werden gemäß dieser Ordnung angefordert.
- Überwachung eines **Wartegraphen** auf den Transaktionen. Der Wartegraph hat eine Kante $T_i \rightarrow T_j$ gdw. T_i sich um ein Privileg bewirbt, das T_j besitzt.
  - Ein Deadlock liegt genau dann vor, wenn der Wartegraph einen Zyklus hat.
  - Ein Deadlock kann aufgelöst werden, indem zu jedem Zyklus eine beteiligte Transaktion abgebrochen wird.

Sperren alleine garantieren keine Serialisierbarkeit.

gesucht: ein Protokoll, dessen Einhaltung durch die Transaktionen Serialisierbarkeit garantiert.

2-Phasen Sperrprotokoll (2PL):

„Nach dem ersten UNLOCK einer Transaktion darf sie keinen LOCK mehr ausführen."

d.h. jede Transaktion gemäß 2PL hat eine Sperrphase und eine Freigabephase. Ein 2PL-Protokoll heißt **strikt**, wenn alle UNLOCK-Operationen zum Transaktionsende durchgeführt werden.

Beispiel:

Eine Transaktion T verändert DB-Objekte A, B, C in dieser Reihenfolge.

Zwei mögliche Varianten von T, die 2PL erfüllen, sind: (RWA steht für $RA\,WA$.)

- LA RWA LB RWB LC RWC UA UB UC
- LA RWA LB LC UA RWB UB RWC UC

Die letzte LOCK-Operation einer Transaktion definiert ihren sogenannten **Sperrpunkt**.

Satz .1 *Das 2-Phasen Protokoll garantiert Serialisierbarkeit.*

Beweis: Sei S ein Schedule bzgl. einer Menge $\{T_1, T_2, \ldots\}$ 2-phasiger Transaktionen.

Angenommen S nicht serialisierbar, d.h. $CG(S)$ enthält einen Zyklus, o.B.d.A. $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k \rightarrow T_1$. Dann existieren Objekte $A_1, \ldots, A_k$, so daß bzgl. S gilt:

$$
\begin{aligned}
S &= \ldots U_1 A_1 \ldots L_2 A_1 \ldots; \\
S &= \ldots U_2 A_2 \ldots L_3 A_2 \ldots; \\
&\vdots \\
S &= \ldots U_{k-1} A_{k-1} \ldots L_k A_{k-1}; \\
S &= \ldots U_k A_k \ldots L_1 A_k \ldots;
\end{aligned}
$$

Sei l_i der Sperrpunkt von T_i. Dann impliziert S, daß l_1 vor l_2 vor l_3 vor ... vor l_{k-1} vor l_k vor l_1. Dies ist ein Widerspruch zu der Struktur von S.

□

2-Phasen Sperren ist optimal in dem Sinn, daß zu jeder nicht 2-phasigen Transaktion T_1 eine 2-phasige Transaktion T_2 existiert, so daß zu T_1, T_2 ein nicht serialisierbarer Schedule existiert.

z.B.: $T_1 = L_1A\ R_1W_1A\ U_1A\ L_1B\ R_1W_1B\ U_1B$
$T_2 = L_2A\ L_2B\ R_2W_2A\ R_2W_2B\ U_2A\ U_2B$

Der folgende Schedule S (nur LOCK und UNLOCK) ist nicht serialisierbar:

$$S = L_1A\ U_1A\ L_2A\ L_2B\ U_2A\ U_2B\ L_1B\ U_1B$$

aber: optimal bedeutet nicht, daß jeder serialisierbare Schedule auch bei Einhaltung des 2-Phasen Protokolls entstehen kann!

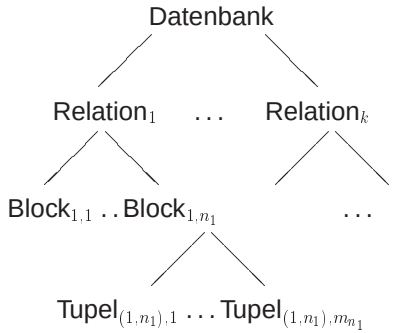
Der folgende Schedule S

$$S = R_1A\ R_2W_2A\ R_3W_3B\ W_1B$$

ist serialisierbar, jedoch gibt es keine Möglichkeit in die entsprechenden Transaktionen T_1, T_2 gemäß 2PL LOCK- und UNLOCK-Operationen einzufügen, sodaß der Schedule S bei Ablauf der beiden Transaktionen auch entstehen kann.

Transaktionsverwaltung: spezielle Verfahren

(A) Sperren bei strukturierten Daten



Teilmengenbeziehung auf den einzelnen Objekten.

gewünschtes Sperrprotokoll:

1. Sei T_1 eine Transaktion, die alle Tupel einer Relation benötigt: Sperre die Relation!
2. Sei T_2 eine Transaktion, die wenige Tupel einer Relation benötigt: Sperre die Tupel!

Ziel: Kompromiß zwischen der Anzahl Sperren und hoher Parallelität.

Problem: Wie kann garantiert werden, daß nicht ein Knoten gesperrt wird, zu dem bereits Nachfolger gesperrt sind?

$\Longrightarrow$ **Warnsperren (intention locks, IL)**

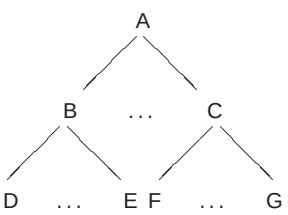
Warnprotokoll für hierarchische Granularitäten:

1. Zuerst muß die Wurzel mittels WL oder IL gesperrt werden.
2. Ein Objekt darf nur über WL oder IL gesperrt werden, wenn der direkte Vorgänger mit IL gesperrt ist.
3. Ein Objekt darf nur dann freigegeben werden, wenn kein Nachfolger mehr gesperrt ist.
4. Das 2-Phasen-Sperrverfahren wird eingehalten.

Kompatibilitätsmatrix: (nur Write-Locks)

	IL	WL
IL	ja	nein
WL	nein	nein

Beispiel:



Datenbank

Relationen

Tupel

Zeit ↓

T_1	T_2	T_3
IL A		
	IL A	
		IL A
IL B		
	WL C	
WL D		
	U C	
U D		
	U A	
U B		
		WL B
		IL C
		WL F
U A		
		U B
		U F
		U C
		U A

(B) Phantom-Problem

Bei Tupel-Sperren kann es trotz Anwendung von 2PL zu nicht serialisierbaren Schedulen kommen.

Beispiel:

Konto

KtoNR	Stadt	Saldo
10	MA	60
20	FR	70
30	FR	80

Bilanz

Stadt	Summe
MA	60
FR	150

Sei T_1 eine Transaktion, die die Bilanzsumme überprüft und demzufolge **alle** Konten liest; sei T_2 eine Transaktion, die ein neues Konto aufmacht und die Bilanz entsprechend anpaßt.

Der folgende nicht serialisierbare Schedule ist möglich. Konto[40] ist für T_1 ein **Phantom**; T_1 wollte alle Konten sperren und lesen, Konto[40] wurde jedoch erst nach Abschluß der Lesephase der Konten in die Datenbank eingefügt.

T_1	T_2
read(Konto[10])	
read(Konto[20])	
read(Konto[30])	
	insert(Konto[40,MA,10])
	read(Bilanz[MA60])
	write(Bilanz[MA,70])
read(Bilanz[MA])	

Lösung: Sperren ganzer Tabellen, Sperren von Schlüsselbereichen, Prädikat-Sperren, Indexsperren.

Fazit: Wenn die Menge der Datenbank-Objekte dynamisch verändert werden kann, dann garantiert ein auf Konflikten basierter Serialisierbarkeitstest im allgemeinen nicht mehr Serialisierbarkeit.

Der **SQL2-Standard** verlangt zur Gewährleistung der Serialisierbarkeit für jede Transaktion T die folgende Eigenschaft:

- T sieht nur Änderungen von abgeschlossenen Transaktion.
- Kein von T gelesener oder geschriebener Wert wird vor Abschluß von T durch eine andere Transaktion geändert.
- Wenn T eine Menge von Werten gelesen hat, die durch ein Suchkriterium definiert ist, dann wird diese Menge solange nicht verändert, bis T abgeschlossen ist.

Offensichtlich wird so auch das Phantom-Problem ausgeschlossen.

(C) Hot Spots

Im vorangehenden Beispiel ist das Bilanz-Tupel ein **hot spot**, da jede Buchung eine Änderung der Bilanz impliziert.

- Auf Bilanz-Tupeln werden ausschließlich Additionen und Subtraktionen durchgeführt.
- Additionen und Subtraktionen sind kommutativ.

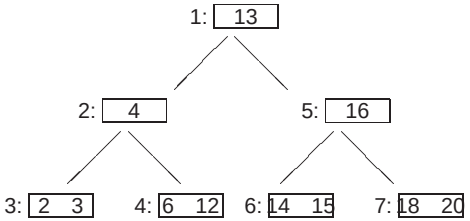
Um lang gehaltene Sperren bzgl. eines hot spot zu vermeiden, können spezielle kommutative atomare Operationen (hier INCREMENT, DECREMENT) mit gesonderten Lock-Modi betrachtet werden.

Kompatibilitätsmatrix:

	RLOCK	WLOCK	INCRLOCK	DECRLOCK
RLOCK	J	N	N	N
WLOCK	N	N	N	N
INCRLOCK	N	N	J	J
DECRLOCK	N	N	J	J

(D) B-Bäume

Beispiel:



T_1 = insert(10)	T_2 = search(12)
R 1	
R 2	
R 4	
	R 1
	R 2
	{nächster Knoten 4
{insert 10 into 4, split 4, neuer Knoten 9}	
W 4 := 6	
W 9 := 10 12	
	R 4
	{12 nicht gefunden!
W 2 := 4 10	

Problem: Paralleles Ändern.

Ziel: Ein Protokoll mit mehr potentieller Parallelität als 2PL, da Semantik der Operationen search, insert, delete bekannt.

Nachteil von 2-Phasen-Sperren:

Die Wurzel ist sehr lange gesperrt (hot spot): Transaktionen ohne Konflikte, die Änderungen verursachen, werden faktisch serialisiert!

Ein Knoten heißt **sicher** bzgl. insert (delete) wenn er nicht ganz voll (bzw. mehr als halbvoll) ist. Sichere Knoten schränken den Restrukturierungsbereich ein.

Ein einfaches Sperrprotokoll für B-Bäume:

	insert/delete	**read**
(1)	Sperre die Wurzel	Sperre die Wurzel
(2)	Lese die Wurzel und mache sie aktuell	Lese die Wurzel und mache sie aktuell
(3)	Solange der aktuelle Knoten kein Blatt ist,	Solange der aktuelle Knoten kein Blatt ist,
(3.1)	Sperre betreff. Nachfolger	Sperre betreff. Nachfolger
(3.2)	Lese ihn und mache ihn aktuell	Gib Sperre des Vorgängers frei
(3.3)	Falls er sicher ist, gib alle Sperren der Vorgänger frei	Lese den betreff. Nachfolger und mache ihn aktuell
(4)	Führe insert/delete aus	Führe read aus
(5)	Gib alle Sperren frei	Gib Sperre frei

Transaktionsverwaltung: Fehlerbehandlung

Grundlagen

Fehlerbehandlung (**Recovery**) garantiert

- Atomizität,
- Dauerhaftigkeit

der Transaktionen.

(1) **Transaktionsfehler**
Lokaler Fehler, z.B. Fehler im Anwendungsprogramm, Abbruch durch Benutzer oder DBS, Verletzung von Systemrestriktionen (Sicherheitsbestimmungen), Auflösen eines Deadlocks.

(2) **Systemfehler**
Fehler mit nicht eingrenzbarem Primärspeicherverlust, Hardwarefehler, falsche Werte in Systemtabellen.

(3) **Mediafehler**
Fehler auf dem Speichermedium, in erster Linie Plattenfehler (Sekundärspeicherverlust).

Annahme: Transaktionen werden gemäß striktem 2PL ausgeführt.

Das Datenbanksystem führt ein **Log** (Journal), in dem die Veränderungen der Datenbank und die Statuswechsel der Transaktionen protokolliert werden.

Das Log(-file), in der Regel ein Plattenfile, wird wie folgt sequentiell beschrieben

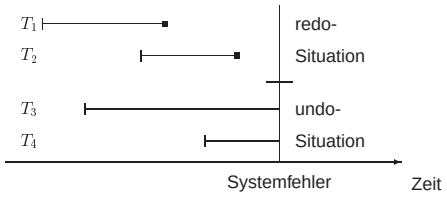
(1) Bei Transaktionsbeginn: $(T, begin)$

(2) Wenn eine Transaktion T eine Aktion WX ausführt:
(T, X, X_{old}, X_{new}) von T erzeugter Wert von X **(after image)**
Wert von X vor WX **(before image)**

(3) Bei Commit: $(T, commit)$

(4) Bei Abbruch: $(T, abort)$

redo- und undo-Situation



- **undo-Situation:** Eine Transaktion schreibt vor Abschluß ihres Commit in die Datenbank. Bevor das Commit ausgeführt werden konnte, tritt ein Fehler auf.
- **redo-Situation:** Eine Transaktion hat ihr Commit ausgeführt. Es tritt ein Fehler auf, bevor alle Schreibaktionen in der Datenbank materialisiert sind.

Stealing Frames and Forcing Pages

- Dürfen geänderte Seiten vor dem Commit in die Datenbank geschrieben werden? Dies kann bei Seitenfehlern notwendig werden.
  steal vs. no-steal
- Wenn das Commit ausgeführt wurde, müssen dann alle Änderungen in der Datenbank eingebracht sein? Aus Effizienzgründen ist man gerne flexibel.
  force vs. no-force

Log-Granulat:
Das Log-Granulat muß in der Regel kleiner oder gleich dem Sperrgranulat sein; werden beispielsweise Sperren auf Satzebene und Logs auf Seitenebene praktiziert, können Änderungen anderer Transaktionen auf derselben Seite beim redo oder undo überschrieben werden.

WAL-Regel (write-ahead-log):
Bevor eine Schreibaktion in der DB materialisiert ist, muß sie im Log-File materialisiert sein. **Materialisiert** heißt, daß der geschriebene Wert tatsächlich in der DB, bzw. im Log-File steht und nicht nur im Puffer gehalten wird.

Commit-Phase:
Jede Transaktion führt als letzte Aktion Commit aus.

- Eine Transaktion heißt **aktiv**, wenn sie ihr Commit noch nicht ausgeführt hat; nach dem Commit heißt sie **abgeschlossen**.
- Mit dem Abschluß des Commit sind alle Schreibaktionen auf dem Log-File materialisiert.
- Erst nach Abschluß des Commit dürfen die von einer Transaktion gehaltenen Sperren freigegeben werden (**striktes 2-Phasen Sperrprotokoll**).

Transaktionsfehler

Sei T eine Transaktion, die vor Abschluß ihrer Commit-Phase abgebrochen wird.

Lese Log-File rückwärts bis zum $(T, begin)$ und materialisiere für jeden gefundenen Eintrag (T, X, X_{old}, X_{new}) den Wert X_{old} für X in der Datenbank.

Systemfehler

Restart-Algorithmus (ohne Sicherungspunkt)

(1) $redone := \emptyset;\ undone := \emptyset$.

(2) Verarbeite das Log-File rückwärts solange, bis Ende erreicht oder $redone \cup undone$ alle Objekte der Datenbank enthält. Sei (T, X, X_{old}, X_{new}) der betrachtete Log-Eintrag. Falls $X \notin redone \cup undone$:

redo: Falls $(T, commit)$ im Log-File enthalten, dann schreibe X_{new} in die Datenbank und setze $redone := redone \cup \{X\}$.

undo: Anderenfalls schreibe X_{old} in die Datenbank und setze $undone := undone \cup \{X\}$.

Erstellen eines Sicherungspunktes:

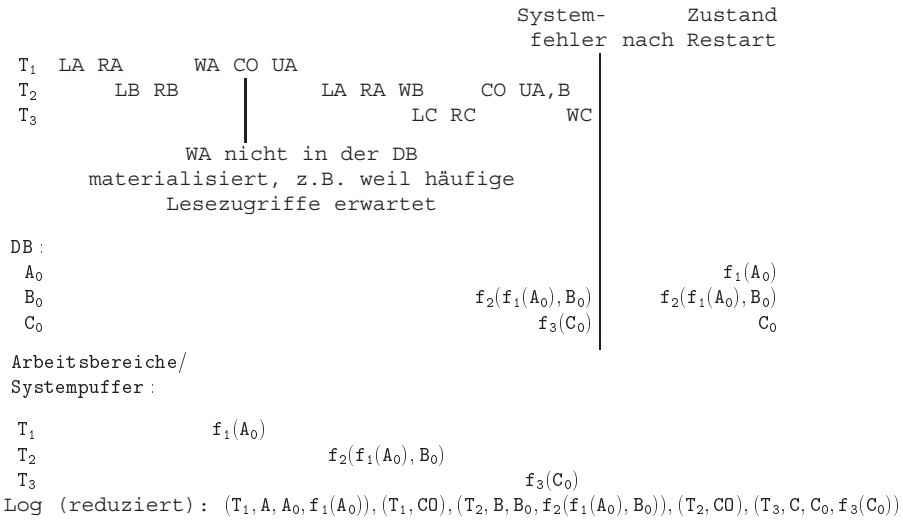
(1) Verbiete den Start neuer Transaktionen und warte bis alle Transaktionen abgeschlossen sind.

(2) Erzwinge das Schreiben aller Änderungen in die Datenbank.

(3) Schreibe zur Kenntlichmachung des Sicherungspunktes *(checkpoint)* an das Ende des Log-Files.

Der Restart-Algorithmus muß jetzt das Log-File nur bis zum letzten Sicherungspunkt verarbeiten.

Beispiel:



Mediafehler

Strategie 1: Halten von mindestens einer Kopie der Datenbank (einschl. Log, ...)

Die Warscheinlichkeit, daß ein Fehler alle Kopien (und das Original) gleichzeitig zerstört, soll praktisch gleich 0 sein.

Dies kann erreicht werden, indem pro Kopie eine eigene Platte, mit eigenem Kontroller, in eigenem Raum, ... verwendet wird.

Bei Verwendung von Kopien bedeutet Schreiben eines Satzes jetzt Schreiben in jeder Kopie.

Schreiben in die Kopie darf erst dann vorgenommen werden, wenn Schreiben im Original als korrekt bestätigt. Sonst besteht die Gefahr, daß Kopien und Original gleichzeitig zerstört werden (z.B: Stronausfall).

Strategie 2: Periodisches Erstellen einer Archiv-Datenbank (dump).

Nach dem Erstellen eines Dumps wird an das Ende des Log-Files ($archive$) geschrieben. Im Falle eines Mediafehlers kann Restart wie folgt durchgeführt werden:

(1) Laden des Dump.

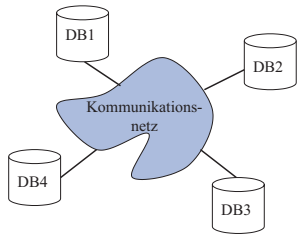
(2) Wende den Restart-Algorithmus nur bzgl. Redo von abgeschlossenen Transaktionen bis zum (archive) Eintrag an.

Transaktionsverwaltung: verteilte Transaktionen

Viele Organisationen sind geographisch verteilt.

- Die einzelnen lokalen (Unter-)Organisationen wollen ihre Daten autonom verwalten können.
- Darüberhinaus gibt es jedoch Bedarf an globalen Verarbeitungen.

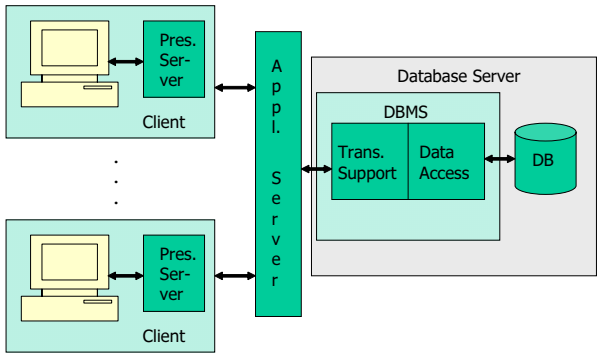
$\Longrightarrow$ Verteiltes Datenbanksystem:



Eine verteilte Datenbank (vDB)

- ist eine Sammlung von Daten, die auf mehrere durch ein Kommunikationsnetz (LAN, WAN, Telefon) verbundene Server verteilt ist.
- Sie ermöglicht eine Kooperation autonom arbeitender sogenannter Sites zur Durchführung einer globalen Aufgabe.
- Sie ist im allgemeinen verschieden von einem Client-Server-System; es werden *homogene* und *heterogene* sog. *Föderationen* unterschieden.
  Eine homogene Föderation ermöglicht *distribution transparency*, d.h. die Verteilung ist nach außen nicht sichtbar.
- Wir beschränken uns auf den Fall, dass die Daten auf den einzelnen Sites nicht repliziert sind; d.h. jedes Datenobjekt wird an genau einer Site gespeichert.

Architektur einer Site:



3-tiered Systemarchitektur

Transaktionen können sich in vDBS über mehrere Rechnerknoten erstrecken.

Transaktionen werden in Form einer Menge von Subtransaktionen ausgeführt, die unter der Kontrolle des DBS einer betreffenden Site stehen.

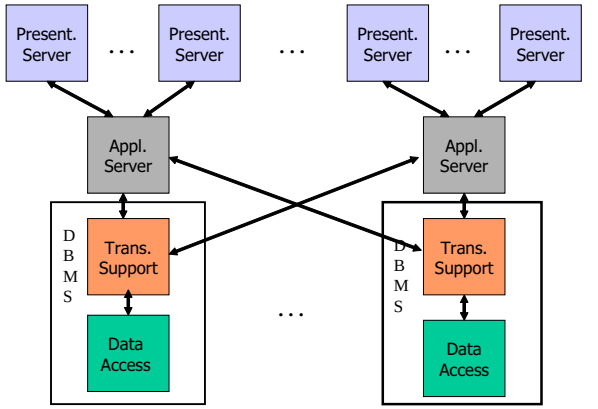
- Serialisierbarkeit muß jetzt global definiert werden.
- Recovery muß global durchgeführt werden; zusätzlich müssen Fehler des Kommunikationssystems berücksichtigt werden.

Hierzu muß eine verteilte Transaktionsverwaltung

- Abhängigkeiten zwischen lokalen Transaktionen und globalen Subtransaktionen an den einzelnen Sites, und
- Abhängigkeiten globaler Transaktionen über mehrere Sites hinweg

behandeln.

Architektur eines verteilten Systems:



3-tiered verteilte Systemarchitektur

(A) Serialisierbarkeit verteilter Transaktionen

Kritisch für die Gewährleistung der globalen Serialisierbarkeit ist die Unterscheidung in *lokale* und *globale* Transaktionen.

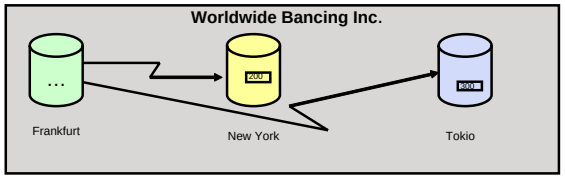
homogene Föderation: Die für den lokalen (nicht verteilten) Fall entwickelten Verfahren können adaptiert werden, da auf allen Sites dieselbe Technik zur Mehrbenutzerkontrolle angenommen werden kann.

Alle Transaktionen können damit als global betrachtet werden.

heterogene Föderation: Die Techniken zur Mehrbenutzerkontrolle auf den einzelnen Sites sind gegenseitig unbekannt. Die globale Serialisierbarkeit sollte sich aus der lokalen Serialisierbarkeit an den einzelnen Sites ergeben.

Es müssen lokale Transaktionen, die nur auf genau einer Site ablaufen, von globalen Transaktionen, die auf mehreren Sites ablaufen, unterschieden werden.

Was ist das Problem der globalen Serialisierbarkeit?



Transaktion 1:
Umbuchung 50€ von NY- nachTokio-Konto.

Transaktion 2:
Hebe von den Konten NY, Tokio alles bis auf einen Restbetrag von 100€ ab.
Auftrag jeweils in Frankfurt gestellt:

Zeit ↓ Transaktion 1, Transaktion 2

New York	Zeit	Tokio
Abheben	↓	Umbuchung
Umbuchung		Abheben

lokal jeweils serialisierbar;

global nicht serialisierbar!

Es werden 50€ zuviel abgehoben.

Globale und lokale Serialisierbarkeit

(A.1) Homogene Föderation

Die Menge der an einer Site i gespeicherten Datenobjekte sei D_i.

Seien n Sites und eine Menge (globaler) Transaktionen $\mathcal{T} = \{T_1, \ldots, T_m\}$ gegeben. Seien weiter $S_1, \ldots, S_n$ lokale Schedule.

- Ein **globaler Schedule** zu $\mathcal{T}$ und $S_1, \ldots, S_n$ ist ein Schedule S zu $\mathcal{T}$ mit der folgenden Eigenschaft.

  Für jede Site i ergibt die Projektion von S auf die an Site i ausgeführten Aktionen gerade den lokalen Schedule S_i.

Beispiel:

Seien $D_1 = \{A\}, D_2 = \{B\}$ und $S_1 = R_1A\ W_2A$, bzw. $S_2 = W_1B\ R_2B$ lokale Schedule.

Die folgenden Schedule S, S' sind zu S_1, S_2 global:

$$S = \begin{array}{ll} Site1: & R_1A \quad\quad W_2A \\ Site2: & \quad W_1B \quad\quad R_2B \end{array}$$

$$S' = \begin{array}{ll} Site1: & R_1A \quad\quad\quad\quad W_2A \\ Site2: & \quad W_1B\ \ R_2B \end{array}$$

S'' ist kein globaler Schedule zu S_1, S_2.

$$S'' = \begin{array}{ll} Site1: & R_1A \quad\quad\quad\quad W_2A \\ Site2: & \quad R_2B\ \ W_1B \end{array}$$

S, S' sind serialisierbar, S'' ist nicht serialisierbar. □

Beispiel:

Seien $D_1 = \{A\}, D_2 = \{B, C\}$.

Der folgende globale Schedule ist nicht serialisierbar:

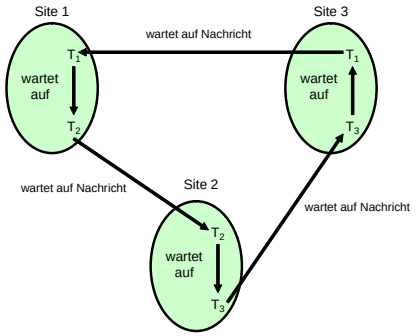
$$S = \begin{array}{ll} Site1: & R_1A \qquad\qquad\quad W_2A \\ Site2: & \quad R_3B\ W_1B \qquad\quad R_2C\ W_3C \end{array}$$

obwohl die beiden zugehörigen lokalen Schedule serialisierbar sind und auf Site 2 die Transaktionen T_1 und T_2 keinen direkten Konflikt haben. □

(i) verteiltes 2-Phasen Sperren

- An jeder Site wird 2PL angewendet.
- Das Problem ist die Synchronisation der Freigabephase der einzelnen Subtransaktionen einer globalen Transaktion.
  - Die Sperrverwaltung wird einer ausgewählten Site (primary site) übertragen, die damit globale Kontrolle hat. Die primary site ist ein potentieller Flaschenhals.
  - Vor Beginn der Freigabephase synchronisieren sich die einzelnen Subtransaktionen einer globalen Transaktion. Hierdurch wird ein hoher Nachrichtenverkehr impliziert.
  - Die Subtransaktionen halten Sperren bis zur Einleitung eines globalen Commit (s. später).
- Deadlocks ergeben sich potentiell über mehrere Sites; ihre Erkennung ist aufwendig.

verteilter Deadlock:



Deadlock über 3 Sites

Erkennen verteilter Deadlocks

zentraler Monitor: Die lokalen Wartegraphen werden durch einen zentralen Monitor integriert und überwacht. Im Falle eines Zyklus ist die Auswahl eines Opfers zum Abbruch problematisch. Desweiteren können durch die Übertragungsverzögerungen nicht mehr existierende Deadlocks behandelt werden.

Time-out: Deadlocks werden in Abhängigkeit von Time-outs angenommen und aufgelöst. Abbruch von Transaktionen auch ohne Vorliegen eines Deadlocks wahrscheinlich.

Edge-chasing: Beim Eintritt in einen Wartezustand sendet die wartende Transaktion ihre ID in einer speziellen Nachricht (*probe*) an die sie blockierende Transaktion.

Erhält eine Transaktionen eine probe-Nachricht, so sendet sie diese an alle Transaktionen weiter, auf die sie wartet. Ist sie nicht blockiert, so wird die probe-Nachricht ignoriert.

Erhält eine Transaktion ihre eigene probe-Nachricht, so erkennt sie einen Deadlock und bricht sich ab. Willkürlicher Abbruch einer Transaktion.

Path pushing: Existiert an einer Site S_u ein Pfad im Wartegraphen von T_i nach T_j, wobei T_i Senke einer 'wartet-auf'-Kante von einer anderen SiteS_v und T_j Quelle einer 'wartet-auf'-Kante zu einer anderen Site S_w, dann sendet die Site S_u den Pfad $T_i \rightarrow \ldots \rightarrow T_j$ zu S_w, sofern der ID von T_i kleiner als der ID von T_j.

Empfängt eine Site einen Pfad, so vereinigt sie diesen mit ihrem Pfad und schickt den resultierenden Pfad wiederum gemäß der ausgehenden 'wartet-auf'-Kante weiter.

Existiert ein Zyklus im globalen Wartegraphen der Länge n, so wird dieser in maximal n Pfadaustauschrunden entdeckt.

Erkennt eine Site einen Zyklus, so kennt sie auch alle beteiligten Transaktionen. Damit ist ein gezielter Abbruch möglich.

(ii) verteiltes Zeitmarken-Verfahren

- Zeitmarken werden global eindeutig vergeben, z.B. $(lokaleZeitmarke, SiteID)$.
- Problematisch, wenn die Uhren an den Sites unterschiedlich schnell hochgesetzt werden.
- Unter einer 'Uhr' soll ein Zähler verstanden werden.

  Die *Lamport*-Uhr macht sich zu Nutzen, dass Sites über Nachrichten miteinander kommunizieren und sich so die Werte ihrer Uhren bekanntgeben können. Jeder Nachricht enthält den Wert der Uhr des Senders. Ist dieser Wert größer als der Wert der Uhr des Empfängers, so kann dieser seine Uhr hochsetzen.

(A.2) Heterogene Föderation

- Globale Transaktionen senden ihre Aktionen einem globalen Transaktionsverwalter (*global transaction manager GTM*). Ein GTM kontrolliert die Ausführung und die Verteilung an die einzelnen lokalen Transaktionsverwalter (*local transaction manager LTM*).
- Der GTM läuft auf einer ausgewählten Site.
- Lokale Transaktionen sind nur ihrem LTM bekannt
- Ziel ist die Gewährleistung globaler Serialisierbarkeit aufgrund gewährleisteter lokaler Serialisierbarkeit. Wir nehmen an, dass der GTM die Aktionen der globalen Transaktionen konsistent zu einer Serialisierung der globalen Transaktionen untereinander an die LTM's weiterreicht.

Beispiel:

Seien $D_1 = \{A, B\}, D_2 = \{C, D\}$. Seien weiter $T_1 = RA\ RD$, $T_2 = RB\ RC$ globale Transaktionen und seien $T_3 = RA\ RB\ WA\ WB$, $T_4 = RD\ WD\ RC\ WC$ lokale Transaktionen.

Der folgende globale Schedule ist nicht serialisierbar:

$$S = \begin{array}{l} Site1: \ R_1A \ \ R_3A \ \ \ R_3B \ \ W_3A \ \ W_3B \ \ R_2B \\ Site2: \ R_4D \ \ W_4D \ \ R_1D \ \ R_2C \ \ \ R_4C \ \ \ W_4C \end{array}$$

obwohl die beiden globalen Transaktionen ausschließlich lesen, ihre Subtransaktion auf beiden Sites in derselben seriellen Folge ausgeführt werden und die zugehörigen lokalen Schedule serialisierbar sind. □

Die Probleme entstehen dadurch, dass lokale Transaktionen *indirekte* Konflikte zwischen globalen Transaktionen erzeugen können. Da keine Kommunikation zwischen einem GTM und einer LTM möglich ist, kann ein GTM nicht entsprechend reagieren.

explizite Tickets

- Jede Site unterhält einen speziellen Zähler, genannt *Ticket*, als reguläres Datenobjekt, zu dem jede globale Transaktion zugreift.
- Jede Subtransaktion einer globalen Transaktion liest das entsprechende Ticket, erhöht den Wert um 1 und schreibt den neuen Wert zurück (*take-a-ticket*-Operation).
- An jeder Site ist die durch die Tickets vorgegebene Ordnung der globalen Transaktionen konsistent zu der Ordnung der lokalen Serialisierung.
- Der GTM muß garantieren, das die Ticketordnung der globalen Subtransaktionen eine lineare Ordnung auf den globalen Transaktionen impliziert.

Beispiel:

Seien $D_1 = \{A, B\}, D_2 = \{C, D\}$. Seien weiter $T_1 = RA\ RD$, $T_2 = RB\ RC$ globale Transaktionen und seien $T_3 = RA\ RB\ WA\ WB$, $T_4 = RD\ WD\ RC\ WC$ lokale Transaktionen.

Die betreffenden Tickets seien I_1, I_2.:

$$S = \begin{array}{l} Site1: \ R_1(I_1)\ W_1(I_1+1)\ R_1A\ R_3A\ R_3B\ W_3A\ W_3B\ R_2(I_1)\ W_2(I_1+1)\ R_2B \\ Site2: \ R_4D\ W_4D\ R_1(I_2)\ W_1(I_2+1)\ R_1D\ R_2(I_2)\ W_2(I_2+1)\ R_2C\ R_4C\ W_4C \end{array}$$

Der LTM von Site 2 erkennt einen nicht serialisierbaren lokalen Schedule. □

implizite Tickets

Ein Schedule S erfüllt die *ACA*-Eigenschaft (*avoids cascading abort*), wenn gilt: T_i liest einen von T_j geschriebenen Wert $\Longrightarrow$ T_j hat bereits *commit* ausgeführt.

- Jeder LTM gewährleistet Serialisierbarkeit für die globalen Subtransaktionen und die lokalen Transaktionen.
- Jede globale Subtransaktion enthält eine *take-a-ticket*-Operation.
- Jeder lokale Schedule erfüllt die *ACA*-Eigenschaft.
- Unter diesen Voraussetzungen ist globale Serialisierbarkeit ohne zusätzliche Überprüfungen des GTM garantiert.

(B) Fehlerbehandlung verteilter Transaktionen

Annahme: keine Replikation der Daten.

Wesentliche Unterschiede zum zentralen Fall?

- Das commit/abort der Transaktion ist verteilt.
- Es sind Fehler möglich, die nur eine Teilmenge der Sites betreffen.

Fehler in einem verteilten System

- **Site Failures:** verursacht durch Systemfehler mit Verlust des Hauptspeichers.
  Annahme: eine Site arbeitet entweder korrekt (operational) oder arbeitet gar nicht (down). Bei einem partiellen Fehler sind nicht alle Sites down; bei einem totalen Fehler sind alle Sites down.
- **Communication Failures:** keine Verbindung zwischen Sites, bzw. verlorene Nachrichten; Partitionierung des Netzwerkes möglich.
- **Undeliverable Messages:** Empfänger ist down, bzw. befindet sich in einer anderen Partition.
  Annahme: Netzwerk versucht nicht die Nachricht später zuzustellen, sondern vernichtet sie.
- **Timeout:** erhält der Sender nach einem gewissen Zeitintervall keine Empfangsbestätigung, so nimmt er einen Site-, bzw. Kommunikationsfehler an; die beiden Typen können nicht unterschieden werden.

2Phase-Commit Protocol (2PC)

Annahme:

- zu jeder Transaktion T existiert an jeder Site, an der eine Operation von T ausgeführt wurde, ein Prozeß zur Durchführung des commit von T. Die Home-Site von T ist der **Koordinator,** die übrigen die **Teilnehmer**. Der Koordinator kennt die Teilnehmer und umgekehrt; die Teilnehmer untereinander kennen sich nicht notwendigerweise.
- an jeder Site wird ein verteilter Transaktions-Log (DT log) geführt, in dem der Koordinator und die Teilnehmer Informationen über den verteilten Ablauf der Transaktion speichern, so daß ein konsistentes Recovery im Fehlerfall möglich wird.

2PC ist ein spezielles Atomic Commit Protocol. In einer ersten Phase **votieren** die beteiligten lokal abgelaufenen Prozesse entweder mit Yes oder No für Commit; in einer zweiten Phase **entscheidet** der Koordinator und **informiert** alle Teilnehmer entsprechend.

Ein Prozess votiert mit Yes, wenn sein lokaler Transaction Manager (im zentralen Fall) ein commit einleiten würde.

Votierungs-Phase:

(1) Koordinator sendet VOTE-REQ zu allen Teilnehmern.

(2) Nach Erhalt VOTE-REQ antwortet jeder Teilnehmer Yes/No. Im Falle von No bricht er sich selbständig ab.

Entscheidungs-Phase:

(3) Koordinator sammelt die Yes/No-Antworten aller Teilnehmer ein und sendet an alle Teilnehmer Commit, sofern alle Yes sendeten, und ansonsten Abort.

(4) Die Teilnehmer, die Yes votierten, warten auf die Entscheidung des Koordinators und führen diese aus.

Two Phase Commit Protocol: Coordinator`s algorithm

```
send VOTE-REQ to all participants;
write start-2PC record in DT log;
wait for vote messages (YES or NO) from all participants
    on timeout begin
        let P_Y be the processes from which YES was received;
        write abort record in DT log;
        send ABORT to all processes in P_Y;
        return
    end;
if all votes were YES and coordinator votes Yes then begin
    write commit record in DT log;
    send COMMIT to all participants
end
else begin
    let P_Y be the processes from which YES was received;
    write abort record in DT log;
    send ABORT to all processes in P_Y;
end;
return
```

Two Phase Commit Protocol: Participant's algorithm

```
wait for VOTE-REQ from coordinator
    on timeout begin
        write abort record in DT log;
        return
    end;
if paticipant votes Yes then begin
    write a yes record in DT log;
    send YES to coordinator;
    wait for decision message (COMMIT or ABORT) from coordinator
        on timeout initiate termination protocol;
    if decision message is COMMIT then write commit record in DT log
    else write abort record in DT log
end
else /* participant's vote is No */ begin
    write abort record in DT log;
    send NO to coordinator
end;
return
```

Cooperative Termination Protocol: Initiator's algorithm

```
start: send DECISION-REQ to all processes;
wait for decision message from any process;
    on timeout goto start;
if decision message is COMMIT then
    write commit record in DT log
    else write abort record in DT log;
return
```

Cooperative Termination Protocol: Responder's algorithm

```
wait for DECISION-REQ from any processes p;
if responder has not voted Yes or has decided to Abort
    then send ABORT to p
else if responder has decided to Commit
    then send COMMIT to p;
return
```

unsicher: Zustand eines Prozesses nach Senden von Yes und vor Erhalt der Entscheidung des Koordinators.

mögliche Wartesituationen:

Schritt (2): Ein Teilnehmer wartet auf VOTE-REQ. Da noch kein Votum des Teilnehmers erfolgte, kann er selbständig auf Abort entscheiden und abbrechen.

Schritt (3): Der Koordinator wartet auf Voten. Da noch keine Entscheidung erfogte, kann er auf Abort entscheiden und alle Teilnehmer, die Yes votierten, informieren.

Schritt (4): Ein Teilnehmer hat Yes votiert und wartet auf die Entscheidung des Koordinators. Der Teilnehmer ist unsicher und kann nicht selbständig entscheiden. Er muß auf Informationen des Koordinators warten, oder andere Teilnehmer nach deren Wissenstand fragen (cooperative termination protocol). (Durch Anhängen an VOTE-REQ kann der Koordinator diese bekannt gemacht haben.) Ist letzteres nicht möglich, so ist er **blockiert.**

Recovery während der Commit-Phase:

- Im Log wird der Zustand der einzelnen Prozesse so protokolliert, daß mittels Analyse des Log konsistentes Recovery möglich wird.
- Kritisch ist Recovery eines unsicheren Prozesses; er muß versuchen durch Ausführung des cooperative termination protocols eine Entscheidung über commit oder abort zu treffen.
- Die Informationen in den Log-Files muß hinreichend lange verfügbar sein: an einer Site müssen mindestens die Informationen so lange verfügbar sein, bis alle beteiligten Teilnehmer erfolgreich ihr commit oder abort durchgeführt haben.

Transaktionsverwaltung: Transaktionsmodelle

Flache Transaktionen

Transaktionen der bisher betrachteten Form haben keine innere Struktur; sie werden als **flach** bezeichnet. Typische Beispiele sind Debit/Credit, Flugbuchung, etc; d.h. **kurze** Transaktionen (1000 Trans. pro Sek.)

- Eine flache Transaktion kann eine beliebige Anzahl von DB-Zugriffen und Nachrichten an die Umwelt enthalten, die durch BEGIN WORK und COMMIT WORK geklammert zu einer atomaren Anwendungs-Aktion werden.
- Mit ROLLBACK WORK können alle Änderungen des DB-Zustandes wieder ruckgängig gemacht werden.
- Bei Auftreten eines die Transaktion betreffenden Systemfehlers wird vom System ROLLBACK WORK ausgelöst.

Beispiel:

Skizze einer Transaktion zum Verbuchen einer Geldüberweisung von Konto A nach Konto B:

1. BEGIN WORK
2. Empfange Benutzerauftrag.
3. Führe Abbuchung (READ und WRITE) Konto A durch.
4. Führe Zubuchung (READ und WRITE) Konto B durch.
5. Wenn Konteninhaber verschieden und Konto A negativ,
   dann ROLLBACK, sonst COMMIT WORK

Bemerkung: Durch die Möglichkeit der Verwendung von COMMIT und ROLLBACK wird die Programmierung prinzipiell vereinfacht.

Flache Transaktionen mit Sicherungspunkten

Grenzen flacher Transaktionen: Reiseplanungen etc, Bulk Updates; **lange** Transaktionen.

Skizze einer Transaktion zur Reiseplanung:

1. BEGIN WORK
2. Empfange Benutzerauftrag Reise von A nach B.
3. Solange Reiseziel B noch nicht erreicht und Reservierung eines Teilstücks möglich
   Führe Reservierung (READ und WRITE) durch.
4. Wenn Reiseziel B erreicht
   dann COMMIT WORK, sonst ROLLBACK WORK

Bemerkung: Es fehlt die Möglichkeit eines selektiven Zurücksetzens der Transaktion, um alternative Reservierungen probieren zu können.

- Mittels SAVE WORK wird erreicht, daß das DBS Sicherungspunkte (Zwischenzustände) einer Transaktion protokolliert; die Abschnitte einer Transaktion zwischen jeweils zwei SAVE WORK Anweisungen sind atomar.
- Mit ROLLBACK WORKi wird zu dem i-ten Sicherungspunkt zurückgegangen.
- COMMIT WORK bezieht sich auf die ganze Transaktion. Bei Auftreten eines die Transaktion betreffenden Systemfehlers wird ROLLBACK WORK bzgl. der gesamten Transaktion ausgelöst.

Skizze einer Transaktion zur Reiseplanung mit Sicherungspunkten:

1. BEGIN WORK
2. Empfange Benutzerauftrag Reise von A nach B; $i := 0$.
3. Solange Reiseziel B noch nicht erreicht und Reservierung eines Teilstücks möglich
   Führe Reservierung (READ und WRITE) durch und SAVE WORKi.
   $i := i + 1$.
4. Wenn Reiseziel B erreicht
   dann COMMIT WORK
   sonst ROLLBACK WORK zu einem ausgewählten Zwischenzustand j,
   $i := j + 1$ und weiter mit 3.

Verteilte Transaktionen

Beispiel:

Ein eCommerce-Unternehmen unterhält Kataloge an unterschiedlichen Sites. Die Preise einer Reihe von Produkten sollen geändert werden. Es muß garantiert sein, daß die Preise in allen Katalogen konsistent geändert werden.

Skizze einer verteilten Transaktion zur Preisänderung:

1. BEGIN WORK
2. Starte an jeder Katalog-Site eine Transaktion zur Preisänderung.
3. COMMIT WORK(2PC)

Geschachtelte Transaktionen

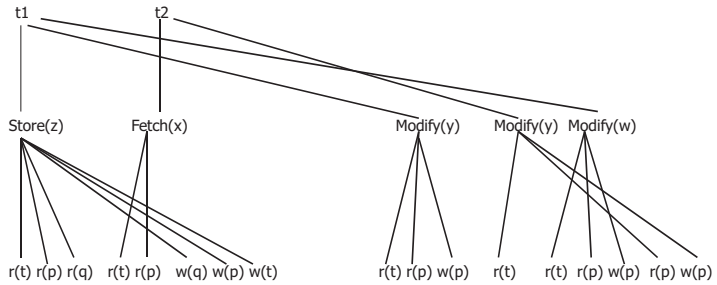
Geschachtelte (genestete) Transaktionen erlauben die interne Zerlegung einer Transaktion in eine Hierarchie von Sub-Transaktionen. Die Zerlegung kann abhängen

- von der Modularisierung von Anwendungsfunktionen: bei Aufruf einer Funktion wird eine neue Sub-Transaktion gestartet,
- von der Abbildungshierarchie der Schichtenarchitektur eines Datenbanksystems (**Mehrebenen-Transaktion**): eine Operation der Ebene i (beispielsweise eine mengenorientierte DB-Operation) wird als Sub-Transaktion gebildet aus der Folge der resultierenden Operationen der Ebene $i - 1$ (den Satzoperationen) realisiert.

Bemerkung:

Die Sub-Transaktionen einer (Sub-) Transaktion können synchron seriell, synchron parallel, und auch asynchron parallel ausgeführt werden. Die Transaktionen als Ganzes erfüllen die ACID-Eigenschaft; die Sub-Transaktionen unter Umständen nicht.

Beispiel:



- Transaktionen - Operationen auf Tupeln - Seitenzugriffe
- Kommutativität auf der Seitenebene ergibt:

  Fetch(t2)(x) vor Store(t1)(z) vor Modify(t1)(y) vor Modify(t1)(w) vor Modify(t2)(y)

- Kommutativität auf der Tupelebene ergibt: t1 vor t2.
- Der Schedule der geschachtelten Transaktionen ist somit serialisierbar!

- **Eigenschaften von Sub-Transaktionen:**

  Atomizität: erforderlich, um isoliertes Rücksetzen zu ermöglichen,

  Konsistenz: unnötig strikt, da Vater-Transaktion (spätestens Wurzel-Transaktion) Konsistenz wiederherstellen kann,

  Isolation: erforderlich, um isoliertes Rücksetzen zu ermöglichen,

  Dauerhaftigkeit: nicht möglich, da Rücksetzen einer übergeordneten Transaktion Rücksetzen aller nachgeordneten Sub-Transaktionen impliziert.

- **Commit-Regel:** Das (lokale) Commit einer Sub-Transaktion macht ihre Ergebnisse nur der Vater-Transaktion zugänglich. Das endgültige Commit der Sub-Transaktion erfolgt genau dann, wenn für alle Sub-Transaktionen bis zur Wurzel das (lokale) Commit erfolgreich verlaufen ist.
- **Rücksetzregel:** Wenn eine Sub-Transaktion auf irgendeiner Schachtelungsebene zurückgesetzt wird, werden rekursiv alle ihre Sub-Transaktionen, unabhängig von deren lokalen Commit-Status, zurückgesetzt.
- **Sichtbarkeitsregel:** Alle Änderungen einer Sub-Transaktion werden bei ihrem Commit für die Vater-Transaktion sichtbar. Alle Objekte, die die Vater-Transaktion hält, können ihren Sub-Transaktionen zugänglich gemacht werden. Änderungen einer Sub-Transaktion sind für Geschwister-Transaktionen nicht sichtbar.

Änderungen von Sub-Transaktionen bleiben somit bis zum endgültigen Commit der geschachtelten Transaktion anderen Transaktionen unsichtbar (Isolation); man redet von **geschlossen geschachtelten Transaktionen**. Wird dies aufgegeben, beispielsweise aus Effizienzgründen, redet man von **offen geschachtelten Transaktionen**.

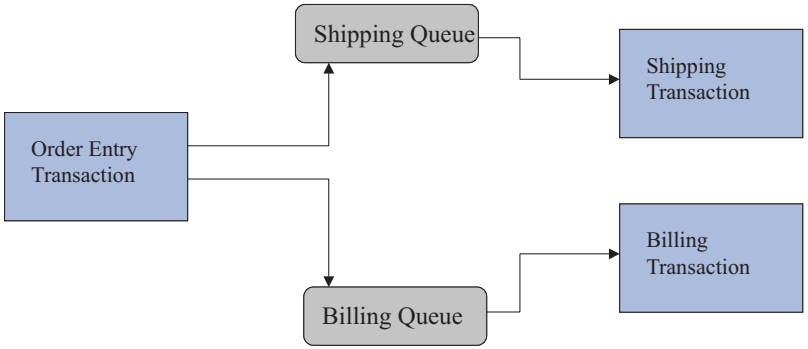
Langlaufende (chained) Transaktionen

... am Beispiel sogenannter **Sagas**. Eine Saga ist eine spezielle Art einer zweistufigen offen geschachtelten Transaktion T bestehend aus Sub-Transaktionen $T_1, \ldots, T_n$. Jede Sub-Transaktion T_i ist eine konventionelle ACID-Transaktion.

- Gemäß dem Prinzip der offenen Schachtelung werden am Ende einer Sub-Transaktion alle Sperren freigegeben, um ein Minimum an Sperrkonflikten zu erreichen.
- Auf eine weitergehende Synchronisation wird verzichtet, sodaß die Gesamttransaktion im allgemeinen nicht serialisierbar abläuft.
- Es besteht die implizite Annahme, daß nur solche Sub-Transaktionen im Rahmen von Sagas verwendet werden, bei denen das Sichtbarwerden der Änderungen vor Ende der Gesamttransaktion tolerierbar ist.
- Wenn eine laufende Saga abgebrochen werden muß, ist das Zurücksetzen bereits beendeter Sub-Transaktionen nur durch **Kompensation** möglich. Aus diesem Grund muß von Seiten des Anwenders für jede Sub-Transaktion T_i eine Kompensations-Transaktion C_i bereitgestellt werden.

Recoverable Queues

- In langlaufenden Transaktionen werden die einzelnen Sub-Transaktionen hintereinander ausgeführt in der Weise, daß bei Ende der i-ten Sub-Transaktion direkt die $i+1$-te gestartet wird.
- In manchen Anwendungen ist dies nicht erforderlich, bzw. nicht erwünscht. Es werden die noch zu erledigenden Teile der Transaktion in einer Schlange gespeichert.
- Eine solche Schlange muß **recoverable** sein und eine Reihe von Bedingungen erfüllen:
  - Führt eine Transaktion **enqueue** bzgl. einem Objekt O aus und danach **abort**, so muß O aus der Schlange wieder entfernt werden.
  - Führt eine Transaktion **dequeue** bzgl. einem Objekt O aus und danach **abort**, so muß O wieder in die Schlange eingebracht werden.
  - Ein mit **enqueue** von einer Transaktion T der Schlange hinzugefügtes Objekt darf erst nach dem **commit** von T von einer anderen Transaktion aus der Schlange entfernt werden.



Sicherheit: Zugriffskontrolle

Geheimhaltung: Informationen dürfen nicht unautorisierten Benutzern offengelegt werden.

Integrität: Daten dürfen nur durch autorisierte Benutzer geändert werden.

Verfügbarkeit: Berechtigter Zugriff darf nicht verweigert werden.

Sicherheitspolitik: Vorgehensweise um das Ziel der Sicherheit zu ereichen.

Sicherheitsmechanismen des DBMS, Betriebssystems, externen Umgebung (z.B. Zugang zu Gebäuden) müssen benutzt werden.

- Denken in mehreren Ebenen erforderlich: Sicherheitslöcher im Betriebssystem oder Netzwerk können die Sicherheitsmechanismen des DBMS umgehen helfen. Beispiel: Log-in als DBA wird ermöglicht.
- Bedenke den menschlichen Faktor, z.B. leicht zu erratende Passworte.

Sicherheitsmechanismen eines DBMS: Sichten (*views*) und Mechanismen für eine Zugriffskontrolle.

Zugriffskontrolle mit Zugriffsrechten: Discretionary access control

SQL2 bietet für die Zugriffskontrolle `GRANT` und `REVOKE` Kommandos an: Zugriffsrechte zu Basisrelationen und Sichten können zugewiesen oder wieder entzogen werden.

```
GRANT privileges ON object TO users [ WITH GRANT OPTION ]

REVOKE [ GRANT OPTION FOR ] privileges ON object FROM users {
                                RESTRICT | CASCADE }
```

Objekte sind u.a. Basisrelationen und Sichten.

Rechte (Privilegien) sind u.a.

- `SELECT`: Recht alle Spalten einer Relation zu lesen.
- `INSERT`: Recht Zeilen in eine Relation einzufügen. Beschränkung auf gewisse Spalten ist möglich.
- `UPDATE`: analog.
- `DELETE`: Recht Zeilen zu löschen.
- `REFERENCES`: Recht Fremdschlüssel zu definieren, die auf die betreffende Relation verweisen. Beschränkung auf gewisse Spalten ist möglich.

Bemerkung:

- Der Erzeuger einer Basistabelle (`CREATE`) hat alle Rechte zu dieser Tabelle.
- Rechte mit `GRANT OPTION` dürfen mittels `GRANT` an andere Benutzer, bzw. Benutzergruppen weitergegeben werden.
- Der Erzeuger einer Sicht hat für diese Sicht gerade diejenigen Rechte, die er bzgl. aller zugrundeliegender Relationen hat. Ändern sich diese Rechte, so ändern sich auch die Rechte für die Sicht; insbesondere, wenn er das `SELECT`-Recht verliert, wird die Sicht entfernt.
- Um Integritätsbedingungen (`CONSTRAINT`) zu definieren, ist ein `SELECT`-Recht erforderlich.

Diskussion am Beispiel:

```
Segler( SId, SName, Bewertung, Alter )
Boote( BId, BName, Farbe )
Reserviert( SName, BId, Tag )
```

Joe erzeugt die Relationen `Segler, Boote, Reserviert`.

Joe weist folgende Zugriffsrechte zu:

```
GRANT INSERT, DELETE ON Reserviert TO Yuppy WITH GRANT OPTION
GRANT SELECT ON Segler TO Michael
GRANT SELECT ON Reserviert TO Michael
GRANT SELECT ON Segler TO Michael WITH GRANT OPTION
GRANT UPDATE( Bewertung ) ON Segler TO Leah
GRANT REFERENCES( BId ) ON Boote TO Bill
```


Michael kann eine Sicht `AktiveSegler` auf `Segler` und `Reserviert` und eine Sicht `JungeSegler` auf `Segler` erzeugen, jedoch keine dieser Sichten verändern. Allerdings kann er das Leserecht zu `JungeSegler` weiterreichen:

```
GRANT SELECT ON JungeSegler TO Eric, Guppy WITH GRANT OPTION
```

Er kann außerdem Constraints definieren, die auf Informationen beruhen, zu denen er ein `SELECT`-Recht besitzt:

```
CREATE TABLE Heimlich ( MaxBewertung INTEGER,
    CHECK ( MaxBewertung >= ( SELECT MAX (S.Bewertung) FROM Segler S )))
```

Beachte, daß durch Einfügen von Tupeln in `Heimlich` die maximale Bewertung in `Segler` bestimmt werden kann.

Leah darf `UPDATE Segler S SET S.Bewertung = 8` ausführen, jedoch wegen eines fehlenden `SELECT`-Rechts nicht:

```
UPDATE Segler S SET S.Bewertung = S.Bewertung - 1
```

Bill darf aufgrund seines `REFERENCES`-Rechts definieren:

```
CREATE TABLE Reserviert2 (
       SName CHAR (10) NOT NULL,
       BId INTEGER, Tag DATE,
       PRIMARY KEY ( BId, TAG ), UNIQUE ( SNAME),
       FOREIGN KEY ( BID ) REFERENCES Boote )
```


Beachte, daß das `REFERENCES`-Recht die Definition von referentiellen Aktionen erlaubt, die das Arbeiten mit der Parent-Relation einschränken.

Die folgenden Kommandos haben im allgemeinen unterschiedliche Effekte:

```
GRANT INSERT ON Segler TO Michael
GRANT INSERT ON Segler ( SId ), Segler ( SName ), ... TO Michael
```

Wird zu `Segler` mittels `ALTER TABLE` eine zusätzliche neue Spalte definiert, so erhält nur im ersteren Fall `Michael` ein `INSERT`-Recht zu dieser Spalte.

`REVOKE` ist als inverse Operation zu `GRANT` gedacht - dies wird daurch kompliziert, daß ein Benutzer dasselbe Recht über unterschiedliche `GRANT`-Anweisungen erhalten haben kann.

Ein Recht heißt **verlassen** (*abandoned*), wenn das Recht, daß für seine Vergabe erforderlich war, zurückgezogen wurde und keine weiteren Vergaben existieren, deren erforderliche Rechte noch existieren.

Die Option `CASCADE` veranlaßt das `REVOKE` der bewirkten verlassenen Rechte; `RESTRICT` führt zum Abbruch der `REVOKE`-Anweisung, wenn diese verlassene Rechte bewirken würde.

1.
```
GRANT SELECT ON Segler TO Art WITH GRANT OPTION    (ausgeführt von Joe)
GRANT SELECT ON Segler TO Bob WITH GRANT OPTION    (ausgeführt von Art)
REVOKE SELECT ON Segler FROM Art CASCADE           (ausgeführt von Joe)
```

   `Art` und `Bob` verlieren ihr `SELECT`-Recht.

2.
```
GRANT SELECT ON Segler TO Art WITH GRANT OPTION    (ausgeführt von Joe)
GRANT SELECT ON Segler TO Bob WITH GRANT OPTION    (ausgeführt von Joe)
GRANT SELECT ON Segler TO Bob WITH GRANT OPTION    (ausgeführt von Art)
REVOKE SELECT ON Segler FROM Art CASCADE           (ausgeführt von Joe)
```

   `Art` verliert sein `SELECT`-Recht; `Bob` behält jedoch sein `SELECT`-Recht.

3.
```
GRANT SELECT ON Segler TO Art WITH GRANT OPTION   (ausgeführt von Joe)
GRANT SELECT ON Segler TO Art WITH GRANT OPTION   (ausgeführt von Joe)
REVOKE SELECT ON Segler FROM Art CASCADE          (ausgeführt von Joe)
```

`Art` verliert sein `SELECT`-Recht.

4.
```
GRANT SELECT ON Segler TO Art WITH GRANT OPTION   (ausgeführt von Joe)
GRANT SELECT ON Segler TO Bob WITH GRANT OPTION   (ausgeführt von Art)
GRANT SELECT ON Segler TO Art WITH GRANT OPTION   (ausgeführt von Bob)
GRANT SELECT ON Segler TO Cal WITH GRANT OPTION   (ausgeführt von Joe)
GRANT SELECT ON Segler TO Bob WITH GRANT OPTION   (ausgeführt von Cal)
REVOKE SELECT ON Segler FROM Art CASCADE          (ausgeführt von Joe)
```

`Art` behält sein Recht.

5.
```
REVOKE SELECT ON Segler FROM Cal CASCADE          (ausgeführt von Joe)
```

Jetzt verliert `Art` sein Recht.

Formalisierung mit **Autorisierungsgraph**:

- Knotenmenge ist die Menge der Benutzer zuzüglich einem Benutzer `System`. Der Erzeuger einer Relation bekommt seine Rechte von `System` zugewiesen.
- Die Kanten zwischen den Knoten sind gerichtet und gewichtet.
  `GRANT` privilege `P ON` object `O TO` user `V [ WITH GRANT OPTION ]` (ausgeführt von U)
  impliziert eine Kante von `U` nach `V` gewichtet mit [`P, O, yes`], falls `WITH GRANT OPTION` gewählt, sonst [`P, O, no`].

Ein Autorisierungsgraph muß die folgende Eigenschaft haben:

Wenn ein Knoten N Quelle einer mit einem Recht gewichteten Kante ist, dann existiert ein Pfad vom Knoten `System` zu N, in dem jede Kante mit demselben Recht und der `GRANT`-Option gewichtet ist.

Zugriffskontrolle mit Zugriffsklassen: Mandatory access control

Zugriffskontrolle mit Zugriffsrechten hat eine Reihe von Schwachstellen, insbesondere sind *Trojanische Pferde* möglich:

Student `Tricky Dick` möchte die Bewertungstabelle des Instruktors `Trustin Justin` lesen. Er macht es so:

- Er kreiert eine neue Tabelle `MineAllMine` und gibt `Trustin` ein `INSERT`-Recht zu dieser Tabelle; `Trustin` merkt nichts davon.
- Er ändert den Code irgendeiner häufig benutzten DB-Anwendung von `Trustin` so ab, daß sie zwei Dinge zusätzlich tut: Lesen der Bewertungstabelle und Schreiben in `MineAllMine`. Er kann dies, da die Anwendung nicht unter der Kontrolle der Schutzmechanismen des DBMS ist.
- Er lehnt sich zurück und wartet ab.

Zugriffsklassen helfen, solche Sicherheitslöcher zu stopfen. Sie können mit einer Zugriffskontrolle mit Zugriffsrechten kombiniert werden.

Bell-LaPadula Modell:

Betrachte **Objekte** (Tabellen, Sichten, Zeilen, Spalten, ...), **Subjekte** (Benutzer, Programme, ...), **Sicherheitsklassen** und **Berechtigungen** (*clearances*). Auf den Sicherheitsklassen ist eine partielle Ordnung definiert, z.B. $TS > S > C > U$ (*top secret, secret, confidential, unclassified*).

Jedem Objekt wird eine Sicherheitsklasse und jedem Subjekt eine Berechtigung für eine Sicherheitsklasse zugewiesen.

Lesen und Schreiben zu Objekten muß den folgenden Bedingungen genügen:

- Subjekt U darf Objekt O lesen, wenn $class(U) \geq class(O)$.
- Subjekt U darf zu Objekt O schreiben, wenn $class(U) \leq class(O)$.

Zurück zum Pferd:

Angenommen die Bewertungstabelle wird der Klasse S zugeordnet und `Justin` bekommt Berechtigung für S, während `Tricky` Berechtigung für C erhält. `Tricky` kann `MineAllMine` höchstens in Klasse C erzeugen. Damit kann `Justin` nicht in `MineAllMine` schreiben.

Die Verwendung einer klassenbasierten Zugriffskontrolle kann **Mehr-Ebenen-Relationen** bewirken mit der Konsequenz, daß Benutzer mit unterschiedlichen Berechtigungen unterschiedliche Tupel der Relation sehen (**Polyinstantiierung**).

Betrachte:

BId	BName	Farbe	Sicherheitsklasse
101	Salsa	Red	S
102	Pinto	Brown	C

Ein Benutzer mit Berechtigung für S, TS sieht beide Tupel, mit Berechtigung C ein und mit Berechtigung U kein Tupel.

Angenommen ein Benutzer mit Berechtigung C möchte das Tupel `(101, Picante, Scarlet,` C`)` einfügen.

- Wird das Einfügen durchgeführt, wird die Schlüsselbedingung verletzt.
- Wird das Einfügen abgelehnt, dann kann der Benutzer schließen, daß ein Tupel mit `BId = 101` und einer höheren Sicherheitsstufe in der Relation enthalten ist!

Lösung: Erlaube die Einfügung und füge die Sicherheitsklasse zum Schlüssel hinzu:

BId	BName	Farbe	Sicherheitsklasse
101	Salsa	Red	S
101	Picante	Scarlet	C
102	Pinto	Brown	C

Statistische Datenbanken

Eine statistische Datenbank erlaubt lediglich statistische Anfragen, wie `MIN, MAX, AVG, ...`, und keine Anfragen nach individuellen Tupeln.

Es kann jedoch unter Umständen Information über Individuen aus den Antworten zu statistischen Anfragen abgeleitet werden:

Segler `Sneaky Pete` möchte die Bewertung von Admiral `Horntooter` wissen, dem Ehrenpräsident des Vereins. Er weiß, daß `Horntooter` das älteste Mitglied ist. `Pete` macht das so:

- Er stellt solange mit größer werdendem X die Anfrage “Wieviele Segler existieren, deren Alter größer als X ist?” bis die Antwort “1” ist. Sei $X = 65$.
- Er stellt die Anfrage: “Was ist die maximale Bewertung aller Segler mit Alter größer 65?” und weiß was er will.

Ausweg: Verlange, daß jede Anfrage eine Mindestanzahlzahl N von Tupeln betrifft.

`Sneaky` ist jedoch nicht dumm und ändert seine Strategie:

- Er stellt solange die Anfrage "Wieviele Segler existieren, deren Alter größer als X ist?" bis das System die Anfrage zurückweist. Sei bis dahin $X = 55$. Er stellt dann zwei weitere Anfragen:
- "Was ist die Summe der Bewertungen aller Segler mit Alter größer 55?" Die Antwort sei A_1.
- "Was ist die Summe der Bewertungen aller Segler mit Alter größer 55 zuzüglich der Bewertung von `Sneaky` und abzüglich der Bewertung von `Horntooter`?" Die Antwort sei A_2.
- Jetzt ist `Pete` fertig; der gesuchte Wert ist $A_1 - A_2 +$ Bewertung von `Sneaky`.

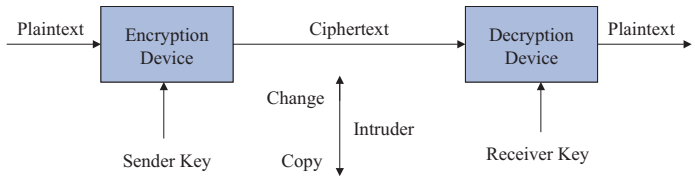
Ausweg: Wenn die Antwortmengen zu zwei Anfragen einen nichtleeren Schnitt haben, dann verlange, daß dieser Schnitt eine Mindestanzahlzahl M von Tupeln enthält.

Um jetzt individuelle Information zu erfragen ist eine Anzahl von Anfragen erforderlich, die proportional zu N/M wächst.

letzter Ausweg: Protokolliere alle Anfragen und versuche rechtzeitig gefährliche Muster zu entdecken.

Sicherheit: Verschlüsselung

Annahme: Verschlüsselung und Entschlüsselung ist jeweils durch einen Schlüssel gesteuert.



Es gilt:

$$ciphertext = K_{sender}[plaintext]; \;\; plaintext = K_{receiver}[ciphertext],$$

wobei $K_{sender}, K_{receiver}$ Schlüssel der Ver-, bzw. Entschlüsselung.

- Ein potentieller Eindringling mag den Ver/Entschlüsselungsalgorithmus kennen,
- jedoch ohne Kenntnis der Schlüssel ist die Entschlüsselung so aufwendig, daß eine erfolgreiche Entschlüsselung extrem unwahrscheinlich ist.

- **symmetrische Verschlüsselung:** Für Ver- und Entschlüsselung wird derselbe Schlüssel verwendet: $K_{sender} = K_{receiver}$
- **asymetrische Verschlüsselung: Public-Key-Verschlüsselung**.
  - Ein Teilnehmer C besitzt einen öffentlichen Schlüssel K_C^{pub} zur Verschlüsselung der zu ihm gesendeten Nachrichten.
  - C besitzt einen privaten Schlüssel K_C^{priv} zur Entschlüsselung der zu ihm mittels K_C^{pub} verschlüsselt gesendeten Nachrichten.
  - Sei M die Nachricht:
$$M = K_C^{priv}[K_C^{pub}[M]]$$

Public-Key-Verschlüsselung basiert typischerweise auf dem RSA-Algorithmus. Sie ist rechenaufwendiger als symmetrische Verschlüsselungsverfahren und wird in der Regel für kurze Informationsblöcke innerhalb von Kommunikationsprotokollen verwendet.

Anforderungen an Public-Key-Verschlüsselung

1. Für einen Client C ist es mit geringem Rechenaufwand möglich, ein Paar (K_C^{pub}, K_C^{priv}) zu generieren.
2. Es ist für einen Sender A mit geringem Rechenaufwand möglich, in Kenntnis von K_C^{pub} zu einer Nachricht M die Verschlüsselung $K_C^{pub}[M]$ zu berechnen.
3. Es ist für C mit geringem Rechenaufwand möglich, unter Anwendung von K_C^{priv} zu einer verschlüsselten Nachricht $K_C^{pub}[M]$ die Entschlüsselung $K_C^{priv}[K_C^{pub}[M]]$zu berechnen.
4. Es ist für einen Gegner praktisch unmöglich, in Kenntnis des öffentlichen Schlüssels K_C^{pub} den privaten Schlüssel K_C^{priv} zu berechnen.
5. Es ist für einen Gegner praktisch unmöglich, in Kenntnis des öffentlichen Schlüssels K_C^{pub} und der verschlüsselten Nachricht $K_C^{pub}[M]$ ohne Kenntnis von K_C^{priv} die Entschlüsselung M zu berechnen.
6. Verschlüsselung und Entschlüsselung kommutieren:
$$K_C^{pub}[K_C^{priv}[M]] = K_C^{priv}[K_C^{pub}[M]] = M.$$

Eine Anwendung: Digitale Signaturen

Sei M eine zu verschlüsselnde Nachricht.

Ein Teilnehmer C kann eine von ihm gesendete Nachricht M digitalen authorisieren, indem er $K_C^{priv}[M]$ sendet.

Wegen

$$M = K_C^{pub}[K_C^{priv}[M]]$$

und der Offenlegung von K_C^{pub} weiß der Empfänger, dass die Nachricht nur von C stammen kann.

Um möglichst kurze zu verschlüsselnde Nachricht zu erhalten, kann als Signatur $K_C^{priv}[f(M)]$ gewählt werden, wobei $f(M)$ ein sogenannter **Message Digest**. Die Message-Digest-Funktion f ist öffentlich und hat die Eigenschaft, dass die Berechnung von M aus $f(M)$ praktisch unmöglich ist.

Auswertung relationaler Operatoren

Techniken zur Auswertung relationaler Operatoren bilden die Grundlage eines **Anfrageoptimierers**, dessen Aufgabe die Bildung eines möglichst optimalen **Auswertungsplans** für die einzelnen Anfragen ist. Ziel in der Realität ist die Vermeidung von ungünstigen Plänen.

Grundlegende Techniken:

Iteration: Alle Tupel in den Eingaberelationen werden iterativ berücksichtigt. Sofern möglich, kann anstatt der Tupel ein entsprechender Index verarbeitet werden.

Indexverwendung: Im Falle einer Selektions- oder Verbundbedingung wird ein Index verwendet, um die die Bedingung erfüllenden Tupel zu bestimmen.

Partitionierung: Eine Menge von Tupeln wird partitioniert, damit Operationen durch eine Reihe von effizienteren Operationen realisiert werden können. Partitionierungen können mittels Sortierung oder Streuung erreicht werden.

Grundlage für Auswahlentscheidungen sind Informationen über die Relationen und ihre Indexstrukturen, die im sogenannten **Katalog** gehalten werden.

- Kataloge enthalten Informationen über
  - Anzahl Tupel und Anzahl Seiten jeder Relation,
  - Anzahl unterschiedlicher Schlüsselwerte und Anzahl Seiten für jeden Index,
  - Höhe, minimalen/maximalen Schlüsselwert für jeden Baum-Index.
- Kataloge werden periodisch aktualisiert.
- Unter Umständen enthalten sie auch statistische Informationen über Werteverteilungen, beispielsweise in Form von Histogrammen.

- Ein **Zugriffspfad** ist eine Methode, um Tupel einer Relation auf dem externen Speicher zu lokalisieren: sequentieller Durchlauf (**scan**) einer Datei, bzw. Verwenden eines zu einem Selektionskriterium **passenden** Index.
  - Ein **Baumindex** paßt zu einem (aus konjunktiv verknüpften Termen gebildeten) Selektionskriterium, wenn die verwendeten Attribute einen Prefix des betreffenden Suchschlüssels ergeben.
    Sei ein Baumindex über $< a, b, c >$ gegeben. Der Index paßt zu $a = 5 \wedge b = 3$ und $a = 5 \wedge b > 3$, aber nicht zu $b = 3$.
  - Ein **Hashindex** paßt zu einem (aus konjunktiv verknüpften Termen gebildeten) Selektionskriterium, wenn die verwendeten Attribute ausschließlich in Gleichheitstermen auftreten und gerade den Attributen des betreffenden Suchschlüssels entsprechen.
    Sei ein Hashindex über $< a, b, c >$ gegeben. Der Index paßt zu $a = 5 \wedge b = 3 \wedge c = 6$, aber nicht zu $a = 5 \wedge b = 3$, oder $b = 3$, oder $a > 5 \wedge b = 3 \wedge c = 6$.
- Die **Selektivität** eines Zugriffspfades wird an der Anzahl der für die Durchführung einer Indexoperation benötigten zu lesenden Seiten (Index + Daten) gemessen.
  Die Selektivität ist um so größer je kleiner ihr Wert ist.

Selektion

Annahme: die Selektion hat die Form: $\sigma[R.field\ \mathbf{op}\ value]R.$

Alternativen:

- **kein Index und unsortierte Daten:** Scan der Datei.
- **kein Index und sortierte Daten:** lokalisiere den Satz mit Suchschlüsselwert $value$ durch binäres Suchen und verarbeite die restlichen die Suchbedingung erfüllenden Sätze ausgehend von diesem.
- **B-Baum:** lokalisiere den Satz mit Suchschlüsselwert $value$ und verarbeite die restlichen die Suchbedingung erfüllenden Sätze ausgehend von diesem. Großer Effizienzgewinn bei Ballung.
- **gestreute Speicherung:** geignet sofern **op** Gleichheit. Großer Effizienzgewinn bei Ballung für den Fall, daß mehrere qualifizierende Sätze existieren.

Annahme: die Selektion enthält eine allgemeine Selektionsbedingung in konjunktiver Normalform, d.h. als Konjunktion einer Menge von Disjunktionen.

(a) Selektion ohne Disjunktion:

(a1) Verwende einen Scan der Datei oder der selektivsten auf ein oder mehrere Terme passenden Indexstruktur, wobei alle verbleibenden Terme auf die gelesenen Sätze angewendet werden.

(a2) Verwende mehrere passende Indexstrukturen. Enthalten die Dateneinträge Adressen zu den eigentlichen Sätzen, dann bilde vor der weiteren Verarbeitung den Durchschnitt dieser Mengen.

(b) Selektion mit Disjunktion:

(b1) Wenn ein Term einer Disjunktion einen Datei-Scan benötigt, dann ist - ohne Betrachtung anderer Konjunktionen - unabhängig von allen anderen Termen dieser Disjunktion ein Datei-Scan erforderlich.

(b2) Wenn eine Disjunktion konjunktiv mit einem Term, bzw. einer Konjunktion von Termen verknüpft ist, zu dem/der eine passende Indexstruktur existiert, dann kann die Gesamtmenge der Sätze zunächst mit diesem Index eingeschränkt werden.

(b3) Wenn zu allen Termen einer Disjunktion eine passende Indexstruktur existiert, dann qualifizieren gerade diejenigen Sätze die Disjunktion, die sich aus der Vereinigung der die einzelnen Terme qualifizierenden Mengen von Sätzen ergeben.

Projektion

Annahme: die Projektion hat die Form $\pi[field1, field2, ..., fieldm]R$.

(a) Projektion mittels Sortierung:

- Führe einen Scan der Datei durch und erzeuge eine Datei mit Sätzen der gewünschten Felder.
- Sortiere die resultierende Datei; der Sortierschlüssel besteht aus allen Feldern.
- Führe einen Scan der resultierenden Datei durch, vergleiche benachbarte Sätze und entferne Duplikate.

Zur Effizienzsteigerung können die obigen Schritte in ein externes Sortierverfahren integriert werden.

(b) Projektion mittels Streuung:

- Führe einen Scan der Datei durch, entferne nicht benötigte Felder und streue die resultierenden Sätz durch Anwendung einer Streufunktion h_1 auf alle Felder. Duplikate liegen jetzt in derselben Kachel.
- Lese die Seiten einer Kachel, wende auf alle Felder der betreffenden Sätze eine von h_1 verschiedene Streufunktion h_2 an und erzeuge eine Hauptspeicher-Streutabelle. Wenn ein Satz auf denselben Eintrag gestreut wird wie ein vorheriger Satz, teste auf Duplikate und ignoriere ihn, bzw. füge ihn dem Eintrag hinzu.
- Schreibe die jetzt duplikat-freien Sätze aus der Hauptspeicher-Streutabelle in die gewünschte Datei.

Verbund

Annahme: Der Verbund hat die Form $R \bowtie_{R.A=S.B} S$.

nested-loop-join

sort-merge-join

hash-join

(A) nested-loop-join

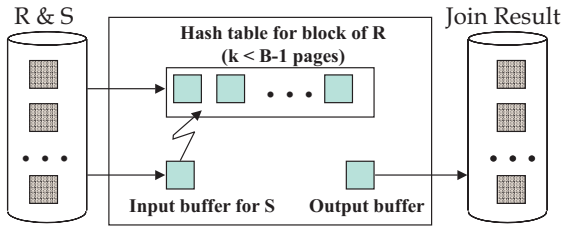
```
foreach tuple r ∈ R do
    foreach tuple s ∈ S do
        if r_A = s_B then add [r, s] to result
```

Sehr ineffizientes Verfahren; kann erheblich verbesert werden durch

- Parallelisierung: Partitioniere eine der beiden Relationen und weise jede Partition einem Prozessor zu,
- seitenorientierte Implementierung, in der alle Tupel der Seiten betrachtet werden, die sich zu R und S gerade im Hauptspeicher befinden.

block-nested-loop-join:

Wähle als äußere Relation R die kleinere Relation (damit wird der Einleseaufwand minimiert) und versuche sie im Hauptspeicher zu halten. Geht dies nicht, so unterteile sie in Blöcke aus mehreren Seiten, so daß jeder Block im Hauptspeicher gehalten werden kann. Unterstütze den Zugriff zu den Sätzen der Blöcke durch dynamischen Aufbau einer Streuungstabelle.



index-nested-loop-join:

Falls zu einer der beteiligten Relationen ein Index über dem Verbundattribut existiert, wähle diese als innere Relation S. Alternativ kann beim ersten Durchlauf durch die innere Relation S **on-the-fly** ein Index aufgebaut werden.

(B) sort-merge-join

```
if R not sorted on attribute A, sort it;
if S not sorted on attribute B, sort it;

Tr = first tuple in R;
Ts = first tuple in S;
Gs = first tuple in S;

while Tr ≠ eof and Gs ≠ eof do {
    while Tr ≠ eof and Tr.A < Gs.B do
        Tr = next tuple in R after Tr;

    while Gs ≠ eof and Tr_A > Gs_B do
        Gs = next tuple in S after Gs;

    while Tr ≠ eof and Tr_A = Gs_B do {
        Ts := Gs;
        while Ts ≠ eof and Ts_B = Tr_A do {
            add [Tr, Ts] to result;
            Ts = next tuple in S after Ts;}
        Tr = next tuple in R after Tr;
        }
    Gs := Ts;
}
```

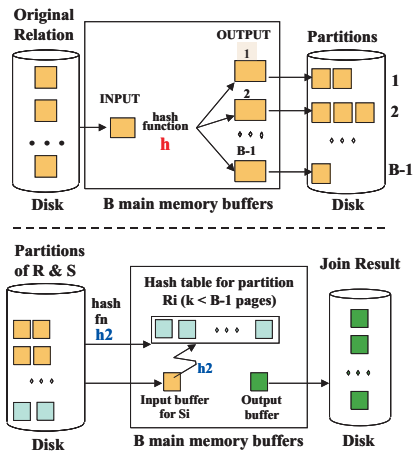
(C) hash-join

partitioning (building) phase: Partitioniere die kleinere Relation R durch Anwendung einer Hash-Funktion $h1$. Partitioniere dann die größere Relation S durch Anwendung derselben Funktion $h1$.

matching (probing) phase: Die potentiellen Joinpartner sind in "gegenüberliegenden" Partitionen. (Annahme: jede Partition von R kann vollständig im Hauptspeicher gehalten werden.) Bilde dann den Join unter zuhilfenahme einer Hauptspeicher-Streuungsfunktion $h2$, indem

- die Partitionen zu R nacheinander in den Hauptspeicher gelesen und mittels $h2$ gestreut werden,
- zu jeder (vollständig im Haupspeicher befindlichen) Partition von R die Seiten der entsprechenden Partition zu S nacheinander in den Hauptspeicher geholt werden und für jeden Satz die korrespondierenden Sätze mittels $h2$ bestimmt werden.

hash-join:



Mengen-Operationen: Vereinigung, Differenz, Durchschnitt, Produkt

- Durchschnitt und (kartesisches) Produkt sind Spezialfälle der Verbundoperation.
- Vereinigung und Differenz mittels Sortierung: analog sort-merge-join.
- Vereinigung und Differenz mittels Streuung: analog hash-join.

Aggregierung: SUM, AVG, COUNT, MIN, MAX

- Die gegebene Relation wird durchlaufen um benötigte Zustandsinformation zu extrahieren. GROUP BY kann mittels Sortierung oder Streuung unterstützt werden.
- Ein Index zu der gegebenen Relation kann anstatt der eigentlichen Sätze der Relation verarbeitet werden, wenn der Suchschlüssel alle für die Aggregierung benötigten Attribute enthält.
- Wenn die Attribute einer GROUP BY Klausel Präfix des Schlüssels eines baumförmigen Index sind, dann kann die Sortierung der gegebenen Relation vermieden werden.

Optimierung

Anfragen an eine Datenbank werden **deklarativ** gestellt. Eine **effiziente Antwortberechnung** setzt eine **Optimierung** durch das DBS voraus.

(I.) Semantische Optimierung: Unter der Annahme, daß die DB alle Integritätsbedingungen erfüllt, kann ihre Kenntnis zur Vereinfachung der Anfrage verwendet werden.

Beispiel: Gilt, daß kein Angestellter mehr verdienen darf als sein direkter Vorgesetzter, so braucht eine Anfrage nach allen Angestellten, die eben dies tun, gar nicht erst berechnet werden.

(II.) Logische Optimierung (algebraische Optimierung): Wandle die SQL-Anfrage in einen äquivalenten Ausdruck der relationalen Algebra um (kanonische Form) und forme diesen äquivalenzerhaltend um gemäß einer geeigneten Heuristik.

(III.) Physische Optimierung: Ordne den Operatoren des Algebraausdruckes gemäß einer geeigneten Heuristik sogenannte Iteratoren zur Bildung eines **Auswertungsplans** zu. Ein **Iterator** realisiert in Form eines abstrakten Datentyps einen Operator oder auch eine Indexstruktur.

Äquivalenz von Algebra-Ausdrücken:

Ausdrücke Q, Q' heißen **äquivalent**, $Q \equiv Q'$, genau dann, wenn für jede Instanz $\mathcal{I}$ der betreffenden Datenbank gilt: $\mathcal{I}(Q) = \mathcal{I}(Q')$.

Sei attr(α) die in einer Selektionsbedingung α verwendete Menge von Attributen und seien $Q, Q1, Q2, \ldots$ Ausdrücke mit Formaten X, X_1, ...

- $Z \subseteq Y \subseteq X \Longrightarrow \pi[Z](\pi[Y](Q)) \equiv \pi[Z](Q)$.
- $Z, Y \subseteq X \Longrightarrow \pi[Z](\pi[Y](Q)) \equiv \pi[Z \cap Y](Q)$.
- $\sigma[\alpha_1](\sigma[\alpha_2](Q)) \equiv \sigma[\alpha_2](\sigma[\alpha_1](Q))$.
- attr$(\alpha) \subseteq Y \subseteq X \Longrightarrow \pi[Y](\sigma[\alpha](Q)) \equiv \sigma[\alpha](\pi[Y](Q))$.
- $Q1 \bowtie Q2 \equiv Q2 \bowtie Q1$.
- $(Q1 \bowtie Q2) \bowtie Q3 \equiv Q1 \bowtie (Q2 \bowtie Q3)$.
- attr$(\alpha) \subseteq X_1$, attr$(\alpha) \cap X_2 = \emptyset \Longrightarrow \sigma[\alpha](Q1 \bowtie Q2) \equiv \sigma[\alpha](Q1) \bowtie Q2$.
- attr$(\alpha) \subseteq X_1 \cap X_2 \Longrightarrow \sigma[\alpha](Q1 \bowtie Q2) \equiv \sigma[\alpha](Q1) \bowtie \sigma[\alpha](Q2)$.

Auswertungsplan

- Die Bestimmung des optimalen Auswertungsplans zu einem Ausdrucks basiert auf einem datenabhängigen Kostenmodell.
- In der Regel muß mit Heuristiken gearbeitet werden.

Grundlage für die Auswahl eines Auswertungsplans ist Information über Indexstrukturen, Ballungen, Kardinalitäten und Verteilungsannahmen von Werten.

Die **Selektivität** läßt Rückschlüsse auf die Größe von Zwischenergebnissen machen.

- Selektion mit Bedingung p:

$$sel_p := \frac{|\sigma[p](R)|}{|R|}$$

  Relativer Anteil der das Selektionskriterium erfüllen Tupel.
  Ist p gerade $R.A = c$ und A Schlüssel von R, so gilt $sel_p = 1/|R|$.
  Existieren i unterschiedliche Werte von $R.A$ und sind diese gleichverteilt, so haben jeweils $\frac{|R|}{i}$ denselben Wert. Es gilt dann $sel_p = 1/i$.

- Join von R und S:

$$sel_{RS} := \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie S|}{|R| \cdot |S|}$$

  Relativer Anteil der Ergebnistupel bzgl. dem kartesischen Produkt.
  Gilt $R \bowtie_{R.A=S.B} S$ und A ist Schlüsselattribut, so ergibt sich $|R \bowtie_{R.A=S.B} S| \leq |S|$ und damit $sel_{RS} \leq 1/|R|$.

Operatorberechnung mittels Pipelining:

Besteht eine Anfrage aus mehreren Operatoren, sollte die Berechnung und Materialisierung von temporären Relationen als Zwischenergebnisse möglichst vermieden werden. **Pipelining** bedeutet die direkte Weiterreichung von Zwischenresultaten an Folgeoperatoren, ohne daß diese materialisiert werden.

Beispiele:

- $\sigma[A = 5 \wedge B > 6]R$:
  Existiere eine passende Indexstruktur, die lediglich die Bedingung $A = 5$ unterstützt.
  Berechnung ohne Pipelining: Berechne zunächst $\sigma[A = 5]R$ unter Verwendung der Indexstruktur und materialisiere das Resultat als Relation R'. Berechne dann $\sigma[B > 6]R'$.
  Berechnung mit Pipelining: Berechne $\sigma[A = 5]R$ unter Verwendung der Indexstruktur und teste dann **on-the fly** jedes qualifizierende Tupel direkt bzgl. $\sigma[B > 6]$.

- $(R \bowtie S) \bowtie T$:
  Berechnung ohne Pipelining: Berechne zunächst $R \bowtie S$, materialisiere das Resultat als Relation RS und berechne dann $RS \bowtie T$.
  Berechnung mit Pipelining: Setze $Q := (R \bowtie S)$ und betrachte $Q \bowtie T$. Sei Q die äußere Relation einer nested-loop-Auswertung.
  Zur Auswertung von $Q \bowtie T$ fordert Pipelining eine Seite von Q-Tupeln bei Bedarf an. Für jede erhaltene Seite wird die gesamte Relation T betrachtet.
  Eine jeweils angeforderte Seite von Q-Tupeln kann selbst mit nested-loop berechnet werden.

Pipelining ist ein wichtiges Grundprinzip für die Implementierung von Iteratoren. Ob Pipelining zur Anwendung kommen kann, hängt von der implementierten Strategie ab (Pipelining einer auf einem Hash-Join basierten Berechnung?).