Informationssysteme - Zusammenfassung

Patrick Pletscher

13. September 2004

1. Benutzung eines Datenbanksystems)

1.1. Allgemeines

Typen und Subtypen

```
type contact
(  name : string;
   phone: string;
   fax  : string;
   email: url;
   www  : url;
)
```

Vererbung von Eigenschaften. In folgendem Beispiel ist `person` ein Subtyp von `contact`.

```
type person subtype of contact
(  title       : string;
   photo       : uri;

   work_places : ()->
    (locations : set of location);
)
```

Association

Eine Source Collection verbunden mit einer Target Collection, z.Bsp. `Situated_at` das `Contacts` und `Locations` assozieren kann. Dabei wird die Notation `(x:y)` für die Art der Bindung benutzt, `x` ist dabei die minimale Anzahl (meistens `0` oder `1`) und `y` die maximale Anzahl (meistens `*`).
Dies ist eine Binary Collection.

1.2. Schema Definition

Typen

```
struct phone_struct
(  description : string;
   number      : string;
);

type private subtype of person
(  birthdate   : date;
   phone       : set of phone_struct;
   music       : uri;
   transport   : uri;
   age         : () -> (years:integer);
```

Unäre Collections

```
collection Contacts   : set of contact;
```

Binäre Collections

```
struct works_for_struct
(  employee : person;
   employer : organisation;
)
collection Works_for : set of works_for_struct;
```

Constraints

```
constraint Works_for association from
  Persons (0:*) to Organisations (0:*)
...
constraint Persons subcollection of Contacts;
constraint Organisations subcollection
 of Contacts;
...
constraint (Persons and Organisations)
  partition Contacts;
```

1.3. AQL Query Language

Numerische Werte

Arithmetische Ausdrücke:

```
r"3.14"+i"6"
```

Unterstützt Typen: Integer(i) und Real(r).

Unterstützte Operationen: `+, i, *, /, mod, ^, abs, integer, float, floor, sin, cos, atan, sqrt, log, exp`

Boolsche Werte

```
l"true", l"false"
```

Boolsche Operatoren: `not, and, or, xor`

Vergleiche: `=, <, >, =<, >=, <>` wobei `<>` ungleich ist.

Operationen auf Objekte

Objekt Attribute und Methoden:

```
t"eth_person"(o"0805").office

t"private"(o"0804").age()
```

Operationen auf Collections

Mengenoperationen: `union, intersection, minus`

```
c"Persons" union c"Organisations"
```

Kardinalität:

```
count c"Persons"
```

Selektieren

```
all P in c"Persons" having (P.title = "Prof")
```

Binäre Collections

Alle bisherigen Collections Operationen sind auch darauf anwendbar, zusätzlich gibt es die `domain/range` Funktionen, die `domain_restriction/ range_restriction` und die `domain_subtraction/ range_subtraction`
Beispiel um alle Associations ohne ETHZ contacts auszuwählen:

```
c"Situated_at"
domain_subtraction
( all C in c"Contacts"
  having (C.name = "ETH Zurich")
)
```

Inverse

Um Associations umzukehren:

```
inverse c"Works_for"
```

Compose

Um Associations zu verbinden.

Z.Bsp: Wie finde ich die möglichen Arbeitsorte von Personen?

```
c"Works_for" compose c"Situated_at"
```

Closure

Repetiertes Compose mit sich selber. Also der transitive Abschluss.

```
closure c"Part_of"
```

Map

Wende eine Operation auf jedes Objekt in einer Collection an.

```
map P in c"Persons" by (P.name)
```

Element Extraction

```
first c"Persons".name
last c"Persons".name
the 3 in c"Persons".name
```

Reduce

Z.Bsp. ein Totalwert berechnen

```
reduce P in c"Privates"
aggregate A by ((P.age()).years + A)
default 0
```

Nest

Gruppieren von Associations. Also wenn z.Bsp. mehrere Personen für die selbe Firma arbeiten, so wird danach nur noch eine Association erscheinen mit einem Set von Personen.

```
nest( inv(c"Works_for"))
```

Division

Z.Bsp. wer arbeitet für jede Organisation?

```
c"Works_for" div c"Organisations"
```

2. Operationen in OMS

2.1. Objekte

Alle Daten werden als Objekte gespeichert, auch Typen usw.

Informationen über Typen werden als Objekte repräsentiert, diese Typen sind Werte vom Typ `ptype`. Attribute von Typen sind andere Typen entweder als `uni` oder `set`.

2.2. Methoden

Methoden sind Operationen die an Objekttypen gebunden sind, sie werden für individuelle Objekte aufgerufen.

Deklaration

Navigation:

```
type test1(
 ...
 work_places: () -> (locations: set of locations);
)
```

Hergeleitete Attribute:

```
type test2(
 ...
 age: () -> (years: integer);
)
```

Operationen:

```
type test3(
 ...
 send_email: () -> ();
)
```

2.3. Implementierung von Operationen

Definiert als eine Regel der Form:

```
method(ParameterList, ResultList) :- method body.
```

Der `method body` besteht aus mehreren Bedingungen, welche erfüllt werden müssen, sie können entweder mit `,` (und) oder `;` (oder) verknüpft werden. Prolog versucht immer die Bedingungen zu erfüllen, und benutzt dazu Backtracking. Mit `cond1, !, cond2` kann man das Backtracking unterbinden.

self

```
self(-Self)
```

Gibt einem das aktuelle Objekt zurück.

AQL Prädikat / get_ext

```
aql(+Query, -Type -Object)
```

Führe ein AQL-Query aus. Falls die Abfrage eine Collection ausgibt und man gerne ein `set` zurückgeben möchte, so muss man die Collection noch in ein Set verwandeln, dies geschieht mit `get_ext(?CollObj, ?CollExt)`.

get_coll

```
get_coll(?Object, ?Name, ?Type)
get_coll(?Object, ?Name, ?Type, ?Extent)
```

Collection holen, einmal einfach die Collection nach Object, das andere Mal den Extent nach Extent.

findall / member

```
findall(?Template, :Goal, ?Result)
```

selektiert Elemente für die das Prolog-Goal True ist, wählt dabei alle Template-Variablen und speichert alle erfüllenden Belegungen in der Result Liste.

```
member(?Element, ?List)
```

Ist Element in List vorhanden?

Beispiel

```
method([],[Locations]) :-
  self(Self),
  get_coll(_,'Works_for',_,Works_for),
  get_coll(_,'Situated_at',_,Situated_at),
  findall(L,(member((Self,O),Works_for),
             member((O,L),Situated_at)),
          Ls),
  list_to_set(Ls,Locations).
```

2.4. Macros

Macros sind "Applikationen" die mit Datenbanken verknüpft sind. Wie Methoden werden auch Macros als Objekte in der Datenbank repräsentiert.

2.5. Triggers

Werden aufgerufen, falls ein Ereigniss (`create, update, delete, commit`) auftritt.

Beispiel

```
type contact add(init: trigger on create);

contact::init :-
  self(Self),
  get_coll(C,'Contacts',_),
  add_obj_to_coll(Self,C).
```

3. Datenmodellierung

3.1. Metadata

Daten über Daten.

Vom System definierte Typen

type	beschreibt	Beispiel
ptype	persistent types	person
tisa	subtype relationships	person subtype contact
collection	collections	Persons
cisa	subcollection relationsships	Persons subcollection Contacts
association	associations	Works_for
method	methods	age()

Extension Operation

```
ext t"person"
% collection of all objects of type person

ext t"collection"
% collection of all objects of type collection

ext t"ptype"
% collection of all objects of type ptype
```

Metadata Queries

Finde die Namen von allen Typen, welche direkte Subtypes von contact sind.

```
(
  (all T in (ext t"tisa")
    having ((T.supertype).name = "contact")
  )
).name
```

3.2. Data Modellierung

Disjoint

Die Subcollections einer Collection sind disjunkt, keiner kann also beide Typen haben.

$\mathsf{ext}(\mathsf{Secretaires}) \cap \mathsf{ext}(\mathsf{Technicals}) = \emptyset$

Cover

Jeder Member einer Collection muss in einer der Subcollections sein.

$\mathsf{ext}(\mathsf{Programmers}) \cup \mathsf{ext}(\mathsf{Managers}) = \mathsf{ext}(\mathsf{Technicals})$

Partition

Jeder Member einer Collection genau in einer Subcollection.
$\mathsf{ext}(\mathsf{Females}) \cap \mathsf{ext}(\mathsf{Males}) = \emptyset$
$\mathsf{ext}(\mathsf{Females}) \cup \mathsf{ext}(\mathsf{Males}) = \mathsf{ext}(\mathsf{Persons})$

Intersect

Jeder Member von allen Subcollections ist auch in der Collection.

3.3. Design Database

Form von Collections

Set Collection: Keine Duplikate, keine Ordnung
Bag Collection: Duplikate, keine Ordnung
Sequence Collection: Duplikate, Ordnung
Ranking Collection: Keine Duplikate, Ordnung

Collections

```
// Unary Collections
collection Employees: set of employee;

// Binary Collections
collection Works_for : set of
 (secretary, technical);
```

3.4. Object Evolution

dress und strip Operationen auf Objekte, migrate zwischen Collections.

Kinds und Roles

Kinds: fundamentale, fixierte Klassifikation.
Roles: verändern sich während dem Lebenszyklus.

$C_1 \preccurlyeq C_2$ bezeichnet dass C_1 Subcollection von C_2 ist.

Für eine Collection C:

$$\text{kinds } C = \{K \mid \text{K ist ein kind und } C \preccurlyeq K\}$$

$$\begin{aligned} \text{roles } C \quad = \quad & \{R \mid R \text{ ist eine role und } C \preccurlyeq R\} \\ & \{R \mid C \preccurlyeq R \text{ und } R \text{ ist kein kind }\} \end{aligned}$$

Annahmen:

- Jede Klassifikation hat einen einzigen Root
- Ein Root ist ein Kind
- Wenn $C \preccurlyeq C_1$ und es kein C_2 gibt mit $C_1 \preccurlyeq C_2$ dann ist C_1 die maximale Collection von C.
- Jede Collection hat eine einzige maximale Collection.

Migration

`x :: C_1 -> C_2` gültig wenn:

1. x nicht zu einer Subcollection von C_1 gehört
2. x kann nur einen Kind verlieren, wenn es die kontextuelle Role von diesem Kind verliert

   $\forall K \in (\text{kinds } C_1 - \text{kinds } C_2) \exists R \in \text{ roles } K \text{ s.t. } R \notin \text{ roles } C_2$

4. Datenmodelle

4.1. Datenmanagement Anforderungen

Die drei E's des effektiven Datenmanagement:

Effectual Wirksam, macht gewünschte Funktionalität.

Expedient Einfach zu benutzen.

Efficient Befriedigende Performance.

4.2. Datenmodell

Spezifiziert Menge von Konstrukten und Operationen.

1. Konzeptuelles Modellieren
2. Design & Prototyping
3. Implementation

Beispiele

- OM Modell (Konzeptuelles Modellieren und Daten Management)
- Entity-Relationship Modell (Konzeptuelles Modellieren)
- Relationelles Datenmodell (Daten Management)

Datenmodell Definition

Im Allgemeinen 3 Komponenten:

Strukturen Konstrukte, in denen ausgedrückt Konzepte beschrieben werden und wie Information dargestellt wird.

Auflagen Constraints.

Operationen Einige Modelle haben volle Operationsmodelle, andere limitierte oder gar keine.

4.3. Entity-Relationship Datenmodell

Attribute

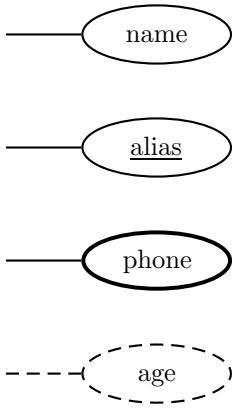

Abbildung 1: Verschiedene Arten von Attributen

Keys

Ein Key für eine Entity-Menge E ist eine Menge K von einem oder mehreren Attributen, so dass zwei disjunkte Entities e_1 und e_2 in E, e_1 und e_2 können nicht identische Werte haben für alle die Attribute.

4.4. Relationelles Modell

Modellinformationen als Tabellen von Daten. Kollektionen von Records. Der Key spielt eine wichtige Rolle um Tupel zu identifizieren.

5. DBMS Features & Architektur

Ein DBMS ist ein Softwaresystem für das effiziente Erstellen und Managen von grossen Datenmengen und das dauerhafte Speichern für lange Zeit.

Dauerhaftes Speichern Memory, Buffer Management und Datenstrukturen, welchen schnellen Zugriff auf grosse Datenmengen ermöglichen.

Programmierinterface Kräftige Sprachen für den User und die Applikationen um Daten zu definieren (DDL), manipulieren (DML) und Query (QL).

Transaktions Management Unterstützt gleichzeitigen Zugriff von mehreren Benutzern und Recovery von Fehlern.

5.1. Query Processing

Query Compiler

Parser, Optimiser und generieren von Execution Plan.

Execution Engine

Auslösen von Anfragen nach Daten, unternimmt entsprechenden Operationen.

5.2. Storage und Buffer Manager

Der Storage Manager kontrolliert das Platzieren von Daten auf der Disk und den Transfer von Daten zwischen Disk und Memory. Der Buffer Manager partioniert das Memory in Buffer. Der Ganze Zugriff auf Daten, die auf der Disk gespeichert sind geschieht über den Buffer Manager.

Was für Daten?

Daten Applikationsdaten, Datenbankinhalt

Metadata Datenbank Schemas

Indexes Datenstrukturen um den effizienten Zugriff auf Daten zu ermöglichen

Statistiken Informationen über die Charakteristiken (z.B. Grösse von Relationen oder die statistische Verteilung von Attributwerten), welche benutzt werden um die Abfragen zu beschleunigen.

5.3. Beschleunigen von Zugriff auf Secondary Storage

- Platzieren von Blöcken, auf die zusammen zugegriffen wird in dem gleichen Disk Zylinder.
- Einführen von Parallelismus (RAID)
- Spiegelung von Daten
- Effizientes Caching
- Prefetching von Disk Blocks

5.4. Transaktionsmanagement

Eigenschaften

Atomacity Alles-oder-Nichts. Transaktionen sind entweder komplett erfolgreich oder versagen.

Consistency Garantieren, dass Datenbank konsistent, bevor man eine Transaktion commited. Constraints müssen geprüft werden.

Isolation Der Effekt von Transaktionen ist anderen nicht sichtbar, bis commited.

Durability Recovery in jedem Fall, somit müssen die Logs an sicherem Speicherplatz gespeichert werden.

Probleme bei Parallelen Transaktionen (gelöst durch Isolation)

Definition 1 (Lost Update Problem). Eine Transaktion T_i updated ein Objekt in der Datenbank. Eine zweite Transaktion T_j updated dasselbe Objekt basierend auf den Werten von T_i.

Definition 2 (Dirty Read Problem). Eine Transaktion T_i liest Datenbankobjekte welche durch eine Transaktion T_j verändert wurden, aber T_j hat noch nicht commited.

Definition 3 (Phantom Problem). Eine Transaktion T_i erstellt Datenbankobjekte (Phantome), welche zu einer Arbeitsdomäne von einer anderen parallelen Transaktion T_j gehören.

6. Relationenmodell

6.1. Relationen

Domäne

Der Wertebereich, eine Menge *atomarer* Werte, normalerweise sind dies bekannte *Basistypen* wie z.B. String oder Integer.

Relationen Schema

Eine endliche Menge von Attributen $\{A_1, A_2, \ldots, A_n\}$ wobei jedem Attribut eine Domäne D_i zugeordnet ist.

Relation

Eine Untermenge des kartesischen Produkts $D_1 \times D_2 \times \ldots D_n$. Es wird $r(R)$ benutzt um zu bezeichnen, dass r eine Relation auf R ist.

Tupel

Ein *Element* einer Relation r wird als Tupel bezeichnet. Wenn $t = \langle d_1, d_2, \ldots, d_n \rangle \in r(R)$, so bezeichnet $t[A_i]$ den Attributwert d_i eines Attributs A_i von einem Tupel t. Kann für Mengen von Attributen $X = \{A_1, \ldots, A_k\}$ erweitert werden, so dass $t[X] = \langle d_1, \ldots, d_k \rangle$.

Grad und Kardinalität

Die Kardinalität von r ist die Anzahl Tupel in r und der Grad von r ist die Anzahl von Attributen in seinem Relationen Schema R.

Relationales Datenbank Schema $\mathcal{R}$

Ein Relationales Datenbank Schema $\mathcal{R}$ ist eine Menge von Relationen Schemas $\{R_1, R_2, \ldots, R_m\}$.

Relationale Datenbank auf $\mathcal{R}$

Eine Relationale Datenbank auf $\mathcal{R}$ ist eine Menge von Relationen $\{r_1(R_1), r_2(R_2), \ldots, r_m(R_m)\}$.

Key

Ein Key von $r(R)$ ist eine Untermenge X von R, so dass für verschiedene Tupel t_1, t_2 von r gilt: $t_1[X] \neq t_2[X]$. Ein *minimaler* Key ist ein Key welcher keine Keys enthält.

6.2. Relationale Algebra

Mengenoperationen

Für zwei Relationen r und s auf demselben Schema R, sind die normalen Mengenoperationen definiert.

$$\begin{aligned} r \cup s &= \{t \mid t \in r \vee t \in s\} \\ r \cap s &= \{t \mid t \in r \wedge t \in s\} \\ r - s &= \{t \mid t \in r \wedge t \notin s\} \end{aligned}$$

Bemerkung. Die entstehenden Schemata sind gleich wie R.

Kartesisches Produkt $\times$

Gegeben seien zwei Relationen $r(R)$ und $s(S)$, wobei $R \cap S = \emptyset$ (was durch Umbenennen der Attributnamen immer erreicht werden kann), dann ist das Kartesische Produkt definiert als:

$$r \times x = \{t \mid t[R] \in r \wedge t[S] \in s\}$$

Bemerkung. Das Schema von $r \times s$ ist $R \cup S$.

Projektion π

Die Projektion einer Relation $r(R)$ auf $X \subseteq R$ ist gegeben durch:

$$\pi_X(r) = \{t[X] \mid t \in r\}$$

Wir wählen also gewisse Attribute einer Relation aus.

Bemerkung. Das Schema von $\pi_X(r)$ ist gleich X.

Selektion σ

Gegeben sei die Relation $r(R)$, wir möchten die Tupel selektieren, welche ein bestimmtes Prädikat P auf ihren Attributwerten erfüllen. Dies kann durch den Selektionsoperator σ_P erreicht werden, welcher definiert ist durch:

$$\sigma_P(r) = \{t \mid t \in r \text{ und } P(t)\}$$

Bemerkung. Das resultierende Schema bleibt das gleiche.

Natürlicher Join $\bowtie$

Gegeben seien zwei Relationen $r(R)$ und $s(S)$, der *natürliche Join* von r und s, geschrieben $r \bowtie s$ ist definiert als:

$$r \bowtie s = \{t \mid \exists t_1 \in r, t_2 \in s \text{ so dass } t_1 = t[R] \text{ und } t_2 = t[S]\}$$

Falls also $R \cap S = \emptyset$ so ist $r \bowtie s = r \times s$ und für den Fall $R \cap S \neq \emptyset$ so ist jedes $t \in r \bowtie s$ eine Kombination von Tupeln von r und Tupeln von s mit gleichen $(R \cap S)$-Werten.

Bemerkung. Das Schema von $r \bowtie s$ ist $R \cup S$.

θ-Join

Wenn $X \subseteq R, Y \subseteq S$ und X und Y θ-vergleichbar sind, so ist der θ-Join von $r(R)$ und $s(S)$ definiert als:

$$r \bowtie_{X\theta Y} s = \{t \mid t[R] \in r \text{ und } t[S] \in s \text{ und } t[X]\theta t[Y]\}$$

Zwei Mengen von Attributen sind θ-vergleichbar gdw. sie die gleiche Zahl von Attributen enthalten, für jedes $A_i \in X$ gibt es ein $V_i \in Y$ mit der selben Domäne D_i und der Relationale Operator θ ist auf D_i definiert. θ ist meistens einer der relationalen Operatoren: $=, \neq, >, \geq, <, \leq$.
Falls der Operator $=$ ist, so spricht man von einem *Equi-Join*.

Bemerkung. Das Schemata von $r \bowtie s$ ist $R \cup S$.

Division $\div$

Gegeben seien zwei Relationen $r(R)$ und $s(S)$ wobei $S \subset R$ und $R' = R - S$. Dann ist r dividiert durch s, geschrieben $r \div s$, die Relation

$$r'(R') = \{t \mid \forall t_s \in s \; \exists t_r \text{ mit } t_r(R') = t \text{ und } t_r(S) = t_s\}$$

Bemerkung. Das Schema von $R \div S$ ist $R - S$. Die Division ist die einfachste Art, Anfragen der Art "... für alle ..." auszudrücken.

Zusammenhang zwischen Natürlichem Join und kartesischem Produkt

Gegeben seien $r(R)$ und $s(S)$, es sei $Q = R \cap S$. Der Natürliche Join kann ausgedrückt werden als:

$$r \bowtie s = \pi_{R \cup S - \{s(Q)\}}(\sigma_{r(Q)=s(Q)} r \times s)$$

Zusammenhang zwischen Division und Natürlichem Join

Gegeben seien $r(R)$ und $s(S)$, wobei $S \subset R$ und $R' = R - S$. Die Division kann ausgedrückt werden als:

$$r \div s = \pi_{R'}(r) - \pi_{R'}((\pi_{R'}(r) \bowtie s) - r)$$

6.3. Datenabhängigkeiten

Gewisse semantische Eigenschaften der Daten können durch Datenabhängigkeiten ausgedrückt werden.

Funktionale Abhängigkeit (FD)

Gegeben sei eine Relation $r(R)$ und $X, Y \subseteq R$, dann erfüllt r die funktionale Abhängigkeit $X \to Y$ wenn für beliebige Tupel $t_1, t_2 \in r,\ t_1[X] = t_2[X]$ impliziert, dass $t_1[Y] = t_2[Y]$. Wir sagen also, dass $X \to Y$ gilt, falls irgendein X Wert gegeben, der Y Wert eindeutig bestimmt ist.

Folgerungsregeln für FDs

Gegeben eine Relation $r(R)$ und $W, X, Y, Z \subseteq R$ so gilt:

F1: (Reflexivität) $X \to X$

F2: (Vergrösserung) $X \to Y \Rightarrow XZ \to Y$

F3: (Addivität) $X \to Y \wedge X \to Z \Rightarrow X \to YZ$

F4: (Projektierung) $X \to YZ \Rightarrow X \to Y$

F5: (Transitivität) $X \to Y \wedge Y \to Z \Rightarrow X \to Z$

F6: (Pseudotransitivität) $X \to Y \wedge YZ \to W \Rightarrow XZ \to W$

F1, F2, F6 - Armstrong's Regeln.

Mehrwertige Abhängigkeit (MVD)

Eine mehrwertige Abhängigkeit $X \to\!\!\to Y$ gilt für eine Relation $r(XYZ)$, wobei X, Y, Z disjunkte Mengen von Attributen sind, gdw. für alle Tupel t_1 und t_2 in r welche denselben Wert für Attribute X haben, also $t_1[X] = t_2[X]$, r auch immer t_3 und t_4 (nicht notwendigerweise disjunkt) enthält, so dass gilt:

$$\begin{gathered} t_1[X] = t_2[X] = t_3[X] = t_4[X] \\ t_3[Y] = t_1[Y] \text{ und } t_3[Z] = t_2[Z] \\ t_4[Y] = t_2[Y] \text{ und } t_4[X] = t_1[Z] \end{gathered}$$

oder in anderen Worten wenn t_1 und t_2 gegeben sind durch:

$$t_1 = (X, Y_1, Z_1) \text{ und } t_2 = (X, Y_2, Z_2)$$

so müssen auch Tupel existieren, so dass

$$t_3 = (X, Y_1, Z_2) \text{ und } t_4 = (X, Y_2, Z_1)$$

Die obigen Bedingungen verlangen, dass Y und Z durch X alleine bestimmt sind, und dass kein Zusammenhang zwischen Y und Z gilt, da alle Y und Z in jeder möglichen Paarung auftreten, somit repräsentieren diese Paare keine Information.

Folgerungsregeln MVDs

Gegeben eine Relation $r(R)$ und $W, X, Y, Z \subseteq R$ so gilt:

M1: (Reflexivität) $X \to\!\!\to X$

M2: (Vergrösserung) $X \to\!\!\to Y \Rightarrow XZ \to\!\!\to Y$

M3: (Addivität) $X \to\!\!\to Y \wedge X \to\!\!\to Z \Rightarrow X \to\!\!\to YZ$

M4: (Projektierung) $X \to\!\!\to Y \wedge X \to\!\!\to Z \Rightarrow X \to\!\!\to Y \cap Z, X \to\!\!\to Z - Y \wedge X \to\!\!\to Y - Z$

M5: (Transitivität) $X \to\!\!\to Y \wedge Y \to\!\!\to Z \Rightarrow X \to\!\!\to Z - Y$

M6: (Pseudotransitivität) $X \to\!\!\to Y \wedge YW \to\!\!\to Z \Rightarrow XW \to\!\!\to Z - YW$

M6: (Komplementierung) $X \to\!\!\to Y \wedge Z = R - XY \Rightarrow X \to\!\!\to Z$

FM1: $X \to Y \Rightarrow X \to\!\!\to Y$

FM2: $X \to\!\!\to Y \wedge Z \to W$, wobei $W \subseteq Y \wedge Y \cap Z = \emptyset \Rightarrow X \to W$

6.4. Normalformen

Erste Normalform

Alle Felder eines Tupels müssen atomare Werte enthalten.

Zweite und Dritte Normalform

Ein Nicht-Schlüsselfeld muss ein Fakt über den Schlüssel, den ganzen Schlüssel und nichts als den Schlüssel liefern.

Enthalten Aussagen über FDs.

Zweite Normalform

Verletzt wenn ein Nicht-Schlüsselfeld ein Fakt über eine Untermenge vom Schlüssel ist.

Beispiel 6.4.1

$$\{\underline{part, warehouse}, quantity, warehouse\_address\}$$

Muss unterteilt werden in:

$$\{\underline{part, warehouse}, quantity\}$$
$$\{\underline{warehouse}, warehouse\_address\}$$

Dritte Normalform

Verletzt wenn ein Nicht-Schlüsselfeld ein Fakt über ein anderes Nicht-Schlüsselfeld ist.

Beispiel 6.4.2

$$\{\underline{employee}, departement, location\}$$

Muss unterteilt werden in:

$$\{\underline{employee}, departement\}, \{\underline{departement}, location\}$$

Vierte und Fünfte Normalform

Enthalten Aussagen über MVDs.

Vierte Normalform

Ein Tupel sollte nicht mehrere Mehrwertige Fakten beinhalten.

Beispiel 6.4.3

$$\{\underline{employee, skill, language}\}$$

Muss unterteilt werden in:

$$\{\underline{employee, skill}\}, \{\underline{employee, language}\}$$

Fünfte Normalform

Der Informationinhalt einer Relation kann nicht rekonstruiert werden von mehreren kleineren Relationen (nicht alle mit dem selben Schlüssel).

Beispiel 6.4.4
Ein Agent verkauft ein Produkt und repräsentiert eine Firma, dann verkauft er die Produkte für diese Firma. Dann könnte man

$$\{agent, company, product\}$$

wiefolgt in drei Relationen aufteilen

$$\{agent, company\} \quad \{agent, product\} \quad \{company, product\}$$

6.5. Design

- Entwickle ein konzeptuelles Modell mit
  - Entity-Relationship Model
  - Object Data Model
- Übertrage es in ein Relationales Schema
- Verhindert viele Fallgruben und Komplexitäten des Relationalen Datenbank Designs

Mapping von konzeptuellem Modell zu Relationenmodell

Typen Jeder Objekttyp wird auf eine Relation gemappt, in welcher jedes Objekt als Tupel repräsentiert wird. Falls das Objekt schon einen eindeutigen Key hat, so kann man diesen als Primary Key wählen, sonst muss man einen eindeutigen Identifier hinzufügen.

Subtypen Ein Subtyp wird wie andere Typen auf eine Relation gemappt. Aber hier hat die Relation einen Foreign Key für den Supertyp.

Collections Im Allgemeinen werden Collections auf unäre Relationen gemappt mit einem Foreign Key auf das entsprechende Tupel, falls es aber Eins-zu-Eins Korrespondenz zwischen Typ und Collection gibt, so ist es nicht nötig die Collection seperat darzustellen.

Subcollections Werden wie andere Collections als Relation dargestellt.

Associations Normalerweise eine neue Relation mit zwei Foreign Key zu der Domain und Range, falls es aber zumindest ein Kardinalitätsconstraint von 1 gibt, so kann die entsprechende durch Foreign Keys eingebunden werden.

7. SQL

7.1. Übersicht

Generelle Query Struktur

```
SELECT {attr to be included in answer | * for
  all attr}
FROM {relation(s) involved in query}
WHERE {selection condition};
```

Mit `SELECT DISTINCT ...` können Duplikate eliminiert werden.

Aggregation

```
SUM
AVG
MIN
MAX
COUNT
```

Stringvergleich

`s LIKE p`, wobei `p` ein Pattern ist, mit speziellen Chars (% für irgendwelche 0 oder mehr Chars oder _ für einen Char).

Null Values

Attribute können den Wert NULL haben. Vergleiche, die NULL beinhalten geben UNKNOWN zurück. Eine arithmetische Operation mit NULL gibt NULL zurück. NULL ist aber *keine Konstante*: Attribute können den Wert NULL haben, NULL kann aber nicht in SQL Ausdrücken vorkommen. Für einen Check, ob ein Wert NULL ist, gibt es `IS NULL`.

Ordnen von Resultaten

```
...
ORDER BY surname DESC, forename ASC;
```

Gruppieren

Ähnlich einem nest in OMS; wird v.a. im Zusammenhang mit Aggregationsfunktionen benötigt. Die Syntax ist wiefolgt:

```
SELECT column,SUM(column)
FROM table GROUP BY column
```

Wenn man z.B. eine Tabelle Sales hat mit Firmen und deren Verkäufen

Company	Amount
IBM	5600
Microsoft	7800
IBM	8300

und man möchte deren Gesamtwert haben, so benutzt man `GROUP BY` wiefolgt:

```
SELECT Company, SUM(Amount) FROM Sales
```

dies ergibt

Company	Amount
IBM	13900
Microsoft	7800

Tupel Variablen

```
SELECT p1.surname, p2.surname
FROM Persons p1, Persons p2
WHERE p1.phone = p2.phone;
```

Subqueries

```
SELECT surname
FROM Persons
WHERE phone =
(
  SELECT phone
  FROM Organisations
  WHERE name='ETH Zurich'
);
```

Conditions für Relationen

`EXISTS R`	true gdw. `R` nicht leer
`v IN R`	true gdw. `v` gleich wie ein Wert in `R` ist, wobei `R` unär ist.
`v > ALL R`	true gdw. `v` grösser als alle Werte in `R`, wobei `R` unär ist.
`v > ANY R`	true gdw. `v` grösser als ein Wert in `R`, wobei `R` unär ist.

Mengenoperationen

`UNION` kann über zwei Subqueries gemacht werden. Die Relationen müssen aber über die gleichen Attribute sein.

`MINUS` durch `WHERE NOT EXISTS`

`INTERSECT` durch `WHERE EXISTS`

7.2. JOIN Operationen und Kartesisches Produkt

Kartesisches Produkt

```
r1 CROSS JOIN r2
```

Das Kartesische Produkt zwischen Relation `r1` und `r2`.

Inner Join

```
r1 INNER JOIN r2 ON p
```

`p` ist dabei ein Vergleich zwischen Spalten von `r1` und Spalten von `r2`. Das Resultat sind alle Tupel wo es einen Match gibt. Wenn es Tupel von `r1` welche keinen Match in `r2` haben, werden die nicht angezeigt und umgekehrt.

Left Join

```
r1 LEFT JOIN r2 ON p
```

Gleich wie Inner Join nur werden hier die Tupel von `r1`, welche keinen Match in `r2` haben, auch ausgegeben.

Right Join

```
r1 RIGHT JOIN r2 ON p
```

Gleich wie Inner Join nur werden hier die Tupel von `r2`, welche keinen Match in `r1` haben, auch ausgegeben.

Full Join

```
r1 FULL JOIN r2 ON p
```

Behält alle Tupel.

Sonstige Joins

Natürlicher Join:

```
r1 NATURAL JOIN r2 ON p
```

θ-Join:

```
r1 JOIN r2 ON p
```

Outer Join:

```
r1 {LEFT|RIGHT} OUTER JOIN r2 ON p
```

Dieser Typ von Join ergibt alle Tupel einer Relation und nur die Tupel der zweiten, wo die Joined Felder gleich sind.

7.3. Datenmanipulation/ Integrität

Datenmanipulation

```
INSERT INTO Persons
(ahv,surname,forename,phone,fax)
VALUES
(123,'Smith','John','27534','27532');

DELETE FROM Persons
WHERE ..

UPDATE Persons
SET surname='Schmidt'
WHERE surname='Smith';
```

Daten Definitions Statements

```
CREATE TABLE Persons(
   ahv      CHAR(14),
   surname  VARCHAR(20)
);
```

`CHAR(14)` muss genau 14 Zeichen enthalten, `VARCHAR(20)` kann beliebige Anzahl kleiner 20 enthalten.

Domain Integrität

Attribute müssen bekannt sein:

```
surname VARCHAR(20) NOT NULL
```

Allgemeine Attribut Constraints Regeln:

```
CHECK (quantity > 0)
CHECK (title IN ('Mr', 'Miss', 'Dr'))
CHECK (orgname IN (SELECT orgname
  FROM Organisations))
```

Entity Integrität

Primary Key

```
CREATE TABLE Persons(
  ahv CHAR(14) PRIMARY KEY,
  ..
)

// oder als relation constraint
CONSTRAINT primary_person
 PRIMARY KEY (ahv)

// fuer zusammengesetzen Key
CONSTRAINT primary_person
 PRIMARY KEY (ahv,orgname)
```

Unique.
UNIQUE erlaubt NULL, während PRIMARY KEY das nicht erlaubt.

Referential Integrität

```
CREATE TABLE contacts.Worksfor (
  ahv     CHAR(14),
  orgname VARCHAR(40),

  CONSTRAINT pk_worksfor
   PRIMARY KEY (ahv,orgname),

  CONSTRAINT fk_worksfor_per FOREIGN KEY (ahv)
   REFERENCES Persons(ahv) ON DELETE CASCADE,

  CONSTRAINT fk_worksfor_orgn
   FOREIGN KEY (orgname)
   REFERENCES Organisations(orgname)
   ON DELETE CASCADE
);
```

Views

'Virtuelle Relationen'

```
CREATE VIEW PublicPerson AS
 SELECT surname,forename,www
 FROM Persons;
```

7.4. Rekursion

Rekursion am Beispiel Microsoft SQL Server 2005 "Yukon". Finde die Anzahl von Personen die für die ETH arbeiten.

```
WITH Part_of_closure(ext_dom, ext_ran)
AS
( -- Anchor Organisation
  SELECT po.ext_dom, po.ext_ran
  FROM Part_of po
   JOIN contact c ON po.ext_ran = c.contact_id
  WHERE c.name LIKE 'ETH%'
  UNION
  -- Recursive Suborganisations
  SELECT po.ext_dom, po.ext_ran
```

```
  FROM Part_of po
   JOIN Part_of_closure po_closure ON po.ext_ran =
   po_closure.ext_dom
)
SELECT COUNT(p.person_id) AS "Total"
FROM person p JOIN Works_for wf
 ON p.person_id = wf.ext_dom
 JOIN organisations o
 ON wf.ext_ran = o.organisation_id
WHERE o.organisation_id IN (
  SELECT c.contact_id FROM contact c
  WHERE c.name LIKE 'ETH %'
  UNION
  SELECT po_closure.ext_dom FROM
   Part_of_closure po_closure
)
```

8. Query Verarbeitung und Optimierung

8.1. Query Verarbeitung

Query

- Deklarativ in Natur
  Spezifiziert mehr WAS getan wird, als WIE
- Optimierung
  Wähle zwischen verschiedenen möglichen Ausführungsplänen
  Wähle eher eine GUTE als eine OPTIMALE
- Code Generierung
  Generiere Code um den gewählten Ausführungsplan auszuführen
  Kompiliert oder interpretiert
- Batch Queries
  Gruppiere Queries und optimiere sie über die ganze Arbeitslast

Übersicht über Verarbeitung

1. *Parser* baut einen Query-Tree
2. *Optimiser* entscheidet sich für einen Ausführungsplan und generiert Code dafür
3. *Evaluator* wertet den Ausführungsplan aus

Query Preprocessing

- Semantik überprüfen
- Ersetze Referenzen zu Views durch ihre Parsebäume

Query Trees

Das Query muss in einen Tree konvertiert werden, welcher einen logischen Queryplan mit Operationen der Relationalen Algebra repräsentiert. Jedes Stadium der algebraischen Query-Optimierung wird einen logischen Query Tree in einen anderen umformen. Am Ende wird der Tree in Code umgewandelt.

Von SQL zur Relationalen Algebra

- Produkt von allen Relationen in FROM Klausel
- Selektieren von Bedingungen in WHERE Klausel
- Projektion über Attribute spezifiziert in SELECT Klausel

Eine WHERE Klausel die ein Subquery enthält kann als eine zwei Argument Auswahl verarbeitet werden. Wir können es als Select Node (σ) mit keinen Parametern und zwei Unterbäumen behandeln:

- Rechtes Kind repräsentiert die Relation R auf welcher die Auswahl ausgeführt wird
- Linkes Kind repräsentiert die Bedingung angewandt auf jedes Tupel von R

Die entstandenen Intermediate trees machen weitere Transformationen nötig, so dass nur noch Operationen der Relationalen Algebra enthalten sind.

Heuristiken

Generell für Relationale Systeme

- Reduziere Grösse der Operandenrelationen so weit wie möglich, so früh wie möglich
- Verschiebe Selektion und Projektion nach unten im Query Tree
- Führe die am meisten verkleinernden Selektion und Join Operation zuerst aus

8.2. Algebraische Optimierung

Man benutzt eine Menge von Umschreibregeln für algebraische Ausdrücke. Eine Auswahlfunktion welche Regeln wann und wie anwendet.

Kommutativität

- $\sigma_P(\pi_X(r)) \rightarrow \pi_X(\sigma_P(r))$
- $\sigma_{P1}(\sigma_{P2}(r)) \rightarrow \sigma_{P2}(\sigma_{P1}(r))$
- $\pi_{X1}(\pi_{X2}(r)) \rightarrow \pi_{X2}(\pi_{X1}(r))$ if $X1 = X2$
- $\pi_X(\sigma_P(r)) \rightarrow \sigma_P(\pi_X(r))$ if $\text{attr}(P) \subseteq X$
- $r\ op\ s \rightarrow s\ op\ r$ where $op \in \{\cup, \times, \bowtie\}$

Assoziativität

$(r\ op\ s)\ op\ u \rightarrow r\ op\ (s\ op\ u)$ where $op\ \in \{\cup, \times, \bowtie\}$

Bemerkung: Assoziativität kann nicht generell auf θ-Join angewendet werden.

Idempotenz

- $\pi_{X1}(r) \rightarrow \pi_{X1}(\pi_{X2}(r))$ where $X1 \subseteq X2$
- $\sigma_P(r) \rightarrow \sigma_{P1}(\sigma_{P2}(r))$ if $P = P1 \wedge P2$

Distributivität

- $\sigma_P(r\ op\ s) \rightarrow \sigma_P(r)\ op\ \sigma_P(s)$ where $op\ \in \{\cup, -\}$
- $\sigma_P(r\ op\ s) \rightarrow \sigma_{P1}(r)\ op\ \sigma_{P2}(s)$ where $op\ \in \{\times, \bowtie\}$ if $\exists P1, P2.(P = P1 \wedge P2)$ and $\text{attr}(P1) \subseteq\ \text{attr}(P2) \subseteq \text{attr}(s)$
- $\pi_X(r \cup s) \rightarrow \pi_X(r) \cup \pi_X(s)$
- $\pi_X(r \times s) \rightarrow \pi_{X1}(r) \times \pi_{X2}(s)$ where $X1 = X2 -\ \text{attr}(s)$ and $X2 = X - \text{attr}(r)$
- $\pi_X(r \bowtie s) \rightarrow \pi_{X1}(r) \bowtie \pi_{X2}(s)$ where ..

Faktorisierung

- $\sigma_P(r) \cup \sigma_P(s) \rightarrow \sigma_P(r \cup s)$
- $\sigma_{P1}(r) - \sigma_{P2}(s) \rightarrow \sigma_{P1}(r - s)$ if $P1 \Rightarrow P2$
- $\sigma_{P1}(r)\ op\ \sigma_{P2}(s) \rightarrow \sigma_P(r\ op\ s)$ where $op \in \{\times, \bowtie\}$ and $P = P1 \wedge P2$
- $\pi_X(r) \cup \pi_X(s) \rightarrow \pi_X(r \cup s)$
- $\pi_{X1}(r)\ op\ \pi_{X2}(s) \rightarrow \pi_X(r\ op\ s)$ where $op \in \{\times, \bowtie\}$ and $X = X1 \cup X2$

8.3. Implementation

Abschätzen der Kosten

Der physikalische Ausführungsplan wird selektiert, der die geschätzen Kosten für eine Query Evaluation minimiert. Die Grösse der Zwischenrelationen hat hier einen grossen Einfluss. Dafür werden Mittelwerte der Anzahl Tupel in Zwischenresulaten benötigt, dies wird z.B. über statistische Schätzungen gemacht.

Abschätzen der Grösse von Auswahlen

$T(R)$ ist die Anzahl Tupel in R, $V(R, A)$ die Anzahl Attribute A in R. Wenn $S = \sigma_{A=c}(R)$ dann schätzt man im Allgemeinen

$$T(S) = T(R)/V(R, A)$$

Select

Linear Search wird nur gewählt, falls es keine bessere Möglichkeit gibt. Sonst werden Zugriffsstrukturen wie Index Tabellen oder Hash Tabellen benutzt. Für Range Queries über die Schlüsselattribute ist es sehr gut, wenn man nur auf erstes zugreifen muss und die restlichen Tupel durch sequentiellen Zugriff bekommen kann (darum ist hier normales Hashing ungünstig; B-Tree sind dafür z.Bsp. gut geeignet).

Join

Eine der teuersten Operationen, darum versucht man diese Operation in andere umzuformen.

Für einen *Nested Loop* greift man meist auf einen Blockzugriff und nicht einen Tupelzugriff zurück. Im Allgemeinen ist es vorteilhaft so viele Blocks wie möglich von der Outer-Loop Relation in den Buffer zu lesen. Die Relation mit den wenigsten Blöcken sollte als die Outer-Loop Relation gewählt werden.

Mengenoperationen

Im Allgemeinen aufwändig, v.a. das kartesische Produkt.

Operationen verbinden

Operationen können verbunden werden, so dass nicht so viele temporäre Dateien generiert werden müssen.

9. Transaktionsmanagement

- Einheit von Arbeit
- Sequenz von Operationen auf einer Datenbank
- Ein konsistenter Zustand wird in einen anderen konsistenten Zustand überführt

Sei $DB = \{x_1, \ldots, x_m\}$ eine Menge von Datenbankobjekten und $OP = \{R, W\}$ die Menge von Datenbankoperationen. Eine Transaktion $T = \langle A_1, \ldots A_n \rangle$ ist eine Sequenz von Aktionen, wobei $A_i,\ i = 1 \ldots n$ entweder $R(x_j)$ oder $W(x_j),\ j = 1 \ldots m$.

9.1. Serielle Transaktions Ausführung

Alle Transaktionen werden von verschiedenen, unabhängigen Benutzern ausgeführt. Für alle dieser Benutzer ist irgendeine sequentielle Ausführung von diesen Transaktionen zulässig. Jede Aktion innerhalb einer Transaktion ist atomically. Wenn eine Menge von Transaktionen $T_1, \ldots, T_n$ seriell ausgeführt wird, so wird es als korrektes Resultat angesehen.

9.2. Parallele Transaktions Ausführung

Parallele Ausführung erhöht die Anwortzeit. Es gibt aber Probleme.

Nebenläufigkeits (Concurrency) Kontrolle

Parallele Ausführung von Transaktionen ist zulässig, falls es äquivalent zu einer Serie von sequentiell ausgeführten und commiteden Transaktionen ist.

9.3. Schedule

Eine Schedule ist eine Menge von verzahnt geordneten Transaktionen $T_1, \ldots, T_n$. Ein Schedule ist serialisierbar, wenn sein Effekt derselbe ist, wie wenn die Transaktionen seriell ausgeführt würden.

Ein Schedule S ist ein Tripel $(T, A, <)$ mit $T = \{T_1, \ldots, T_n\}$
A ist eine Menge von Datenbankoperationen von allen Transaktion in T
$<$ ist eine partielle Ordnung auf der Aktionenmenge A und für jedes Aktionspaar $(a, b) \in A \times A$ gilt:

- $<$ erhält die Ordnung der Aktionen innerhalb einer Transaktion.
- Wenn a und b von verschiedenen Transkationen sind und auf die diesselben Daten zugreifen und mindestens eines eine Schreiboperationen ist, so müssen sie mit Rücksicht auf $<$ geordnet sein. Dies sind sogenannte Konfliktpaare.
- Die übrigen Operationen können parallel ausgeführt werden.

Korrektheit

Zwei Schedules sind (konflikt vermeidend) äquivalent, sofern sie die selben Mengen T und A haben und die Konfliktpaar gleich geordnet sind.

Ein Schedule ist korrekt wenn er äquivalent zu einem seriellen Schedule ist.

9.4. Test für Serialisierbarkeit

Dependency Graph

Sei $S = (T, A, <)$. Der Dependency Graph von S ist ein gerichteter Graph mit Transaktionen als Knoten und einer Kante von T_i nach T_j wenn es ein Konfliktpaar $\langle a, b\rangle$ in S gibt, wobei a zu T_i und b zu T_j gehört und $a < b$.

Theorem 1. *Ein Schedule ist serialisierbar, genau dann, wenn sein Dependency Graph nicht zyklisch ist.*

Algorithmus für seriellen Schedule

- Konstruiere einen Dependency Graph vom Schedule.
- Führe darauf einen topological sorting aus. Entferne also immer einen Knoten ohne incoming edges und entferne alle seine ausgehenden Kanten.
- Wenn der Graph Zyklen enthält, so terminiert der Algorithmus schon bevor alle Knoten entfernt sind.

9.5. Locking

lock(Transaction T_i, Object x, Mode m)
unlock(Transaction T_i, Object x, Mode m)
verschiedene Typen von Locking: X, S. Ein Objekt das mit S gelockt ist, darf auch noch mit S zugegriffen werden von anderem, sonst verboten.

Dies garantiert aber keine serialsierbarkeit.

9.6. Two-Phase Locking

Einschränkung zum Locking: Sobald eine Transaktion ein Objekt unlocked, so können keine weiteren Locks mehr beantragt werden. Es gibt also einen Punkt ab dem nur noch unlock möglich ist.

Theorem 2. *Das 2PL Protokoll garantiert serialsierbare Schedules.*

Ist aber nicht Deadlock frei und es gibt Probleme bei abgebrochenen Transkationen.

Deadlock

Zwei Transaktionen T_i und T_j warten in einem Zyklus auf einen Lock von einer anderen Transkation.

Strict Two-Phase Locking

Alle Locks werden bis zum Schluss gehalten und zusammen freigegeben. So gibt es keine Probleme mit abgebrochenen Transkationen mehr. Wird in allen grösseren Datenbanksystemen implementiert.

9.7. Timestamp Ordering

Jeder Transaktion T_i wird eine eindeutiger Timestamp $ts(T_i)$ in einer strikt monotonen Sequenz zugeordnet. Wenn $o_i(x)$ im Konflikt zu $o_j(x)$ so müssen sie geordnet sein $o_i(x) < o_j(x)$ genau dann wenn $ts(T_i) < ts(T_j)$. Für jeden Zugriff auf ein Objekt muss der Scheduler überprüfen, ob schon eine spätere Transaktion auf das Objekt zugegriffen hat: $maxts(mode\ m, object\ o)$, maximaler Timestamp einer Transkation, die auf Objekt o mit dem Mode m zugriff.

- T_i will Objekt x lesen
  - $ts(T_i) < maxts(w, x)$
    T_i will alte Daten lesen $\rightarrow$ reset T_i
  - $ts(T_i) > maxts(w, x)$
    Führe Leseoperation aus und aktualisiere $maxts(r, x)$
- T_i will Objekt x schreiben
  - $ts(T_i) < maxts(r, x)$
    Es gibt eine jüngere Transaktion, die x gelesen hat $\rightarrow$ reset T_i
  - $ts(T_i) < maxts(w, x)$
    Schreiboperation ist überflüssig
  - $ts(T_i) \geq maxts(r, x) \wedge ts(T_i) \geq maxts(w, x)$
    Führe Schreiboperation aus und aktualisiere $maxts(w, x)$

10. RDMS Architektur

10.1. Buffer Management

Beim Buffer Management geht es um das Laden und Unladen von Pages vom Zweitspeicher in den Hauptspeicher. Stellt Primitiven für den Zugriff auf den Buffer zur Verfügung: fix, use, unfix, flush und force. Dies generiert I/O Operationen als Antwort auf diese Primitiven solange der Zugriff durch den Scheduler erlaubt wird.

fix Fordere Zugriff auf Page an und lade sie in den Buffer. Ausführung der Primitive muss nur auf Zweitspeicher zugreiffen, falls Page noch nicht im Buffer.

use Benutzt durch Transaktion um Zugriff zur Page die vorher geladen wurde zu erhalten.

unfix Transaktion zeigt Buffer Manager an, dass die Benutzung der Page beendet ist, sie ist nicht mehr länger gültig.

force Transferiere eine Page vom Buffer in den Zweitspeicher. Die anfrangende Transaktion wird suspended bis zum Ende der Primitive.

flush Vom Buffer Manager selber benutzt um inaktive oder ungültige Pages in den Zweitspeicher zu transferieren.

Policies

Steal Policy

- Benutzt während fix Operation, erlaubt Buffer Manager andere aktive Page zu selektieren mit anderer Transaktion als Opfer.
- Wird durch no-steal Policy verboten

Force Policy

- Benötigt, dass alle aktiven Pages einer Transaktion in den Zweitspeicher geschrieben werden, wenn die Transaktion ein commit ausführt.
- Wird durch no-force Policy verboten

Kombination von no-steal/no-force wird von meisten DBMS verwendet.

10.2. Index Strukturen

Primary Index immer über Unique Attribute Field, Secondary Index über Non-Unique Attribute Field.

B^+-Tree

Datenpointer nur in Leaf Nodes. Leaf Nodes sind meist zusammengelinkt. Gut für dynamische Daten, Key Values; teilweise auch für Secondary Index, kommt auf Anwendung draufan.

Hashing

Durch Extensible Hashing können mehrere Hash-Tabelleneinträge auf den gleichen Bucket zeigen. Ein Bucket hat lokale Tiefe d - die Anzahl Bits auf welcher Inhalt basiert. Falls nun ein Bucket erweitert werden soll, wird dessen d um eins erhöht und der Inhalt auf zwei Buckets aufgeteilt.

B-Trees in Oracle

Oracle erstellt automatisch B-Bäume für alle Attribute mit Constraint

```
PRIMARY KEY
UNIQUE
```

Index Strukturen können auch explizit erstellt/gelöscht werden

```
CREATE INDEX person_names ON persons(name);
DROP INDEX person_names;
```

Bitmap Index in Oracle

Selektiert binär welche Tupel welche Attribute haben. Häkchen bei Attribut, falls vorhanden.

Tablespace in Oracle

Benutzt um Metadata, Schemas von Application Data und Index Strukturen zu trennen.

10.3. Oracle Objekte

Unterstützen kein Polymorphism oder inheritance.

```
CREATE TYPE sales.part_type AS OBJECT (
 id INTEGER,
 descrption VARCHAR2(50),
 quantity INTEGER,
 reorder_level INTEGER,
 MEMBER FUNCTION
  parts_on_hand(part_id IN INTEGER) RETURN INTEGER,
 MEMBER PROCEDURE
  order_part(part_id IN INTEGER, quantity IN INTEGER)
);
```

Benutzerdefinierte Datentypen

```
CREATE TABLE sales.customers (
 ..,
 address pub.address_type
);

INSERT INTO sales.customers VALUES (
..,
pub.address_type('IFW', ...));
```

Nested Relationen

```
CREATE TYPE sales.item_type AS OBJECT (
 item_id INTEGER,
 quantity INTEGER
);

CREATE TYPE sales.itme_list AS TABLE OF sales.item_type;

CREATE TABLE sales.orders(
 ..,
 line_items sales.item_list,
 NESTED TABLE line_itmes STORE AS items
);
```

Oracle erstellt so zwei logische Tabellen (orders, items), Daten werden aber in einem physischen Datensegment gespeichert.

11. XML für Informationsrepräsentation

11.1. Baumstruktur von XML Dokumenten

- Root node
- Element nodes
- Attribute nodes
- Text nodes

- Comment nodes
- Processing instruction nodes
- Namespace nodes

11.2. Well-Formedness und Validity

Ein XML Dokument ist well-formed, wenn es den Regeln von XML folgt.

Ein XML Dokument ist gültig entsprechend seiner Document Type Definition (DTD).

Ein XML Dokument ist gültig, wenn es komplett selbsterklärend über seine Struktur und Inhalt ist durch den Dokumentinhalt und zusätzliche Dateien, auf welche im Dokument referenziert wird.

11.3. Übersicht über XML Technologien

- SAX und DOM. XML Programmierungs API (für das Parsen eines XML Dokuments). SAX ist Event-based, also wenn z.Bsp. Start-Tag auftritt, wird ein Event ausgelöst. DOM stellt einem einen Baum im Memory her.
- XPath und XPointer. XPath ist eine Sprache die benutzt wird um bestimmte Elemente in einem XML Dokument zu adressieren, dabei werden versch. location steps mit / getrennt. XPointer benutzt XPath um auf Punkte oder Ranges in XML Dokumenten zu zeigen.
- XSL. Für die Transformation von XML Dokumenten.
- XQuery. Abfragen an XML Dokumente.

11.4. Datenrepräsentation

Es gibt im Groben 2 unterschiedliche Möglichkeiten der Datenrepräsentation in XML, Element-basiert, oder Attribut-basiert.

11.5. Document Type Definition (DTD)

Definiert Struktur des Dokuments.

Elements

```
<!ELEMENT ElementName Type>
```

`Type` kann `Empty` sein, `(#PCDATA)` oder wiederum ein anderes Element. Man kann auch mehrere Typen angeben. Man kann die Kardinalität steuern:

- `?` optional, (0 oder 1)
- `*` irgendeine Anzahl von Vorkommen (0 oder mehr)
- `+` benötigt (1 oder mehr)
- default ist genau 1 mal

```
<!ELEMENT person (name, title?, phone*)>
<!ELEMENT mixed (#PCDATA | itemA | itemB)*>
```

Achtung: `#PCDATA` muss erstes Element in Liste sein.

Attribute

```
<!ATTLIST ElementName AttrName AttrType Default>
```

Wobei `Default` sein kann:

- `#REQUIRED`
- `#IMPLIED` optional
- `#FIXED value` Attribut hat einen fixen Wert
- `value` Attribut hat einen Default-Wert, von Hochkomma umschlossen

`AttrType` ist:

- `CDATA` irgendein String wird akzeptiert, sofern wellformed
- `NMTOKEN, NMTOKENS` eines oder mehrere name tokens getrennt durch a whitespace
- `ENUMERATION` Liste von möglichen Werten, z.B. `(Article | Proceedings | ...)`
- `ENTITY, ENTITIES` Name von ungeparsder entity oder List von ungeparsder Entities getrennt durch whitespaces
- `ID` XML Name, welcher eindeutig ist in einem einzigen XML Dokument
- `IDREF, IDREFS` XML Name, welcher auf einen oder mehrere ID Type Attribute im gleichen Dokument verweist.
- `NOTATION` Attribut enthält den Namen einer Notation

```
<!ATTLIST price pricetype
 (cost | sale | retail) 'sale'>
<!ATTLIST inventory colour CDATA #IMPLIED>
```

References

```
<products>
 <company id="celtic">
  ...
 </company>
 <special company_id="celtic">
  ...
 </special>
</products>
```

```
<!ELEMENT company (company_name, product*)>
<!ATTLIST company id ID #REQUIRED>

<!ELEMENT special (#PCDATA)>
<!ATTLIST special company_id IDREF #REQUIRED>
```

Entities

Generelle Entities, können durch `&entityName` im XML Dokument benutzt werden.

```
<!ENTITY globis "Global Information
 Systems Group">
```

Parameter Entities, können nur im DTD benutzt werden, intern oder extern

```
<!ENTITY % gender "(male | female)">
```

Notations

Einbinden von nicht-XML Daten in dem man das Format beschreibt.

```
<!NOTATION jpeg SYSTEM "image/jpeg">

<!ENTITY globisLogo
   SYSTEM "http://myurl/g.jpg"
   NDATA jpeg>
```

11.6. XML Schema

Ersetzt DTD, benutzt selbst auch XML Syntax.

Datentypen

Primitive Datentype sind die Basis für alle anderen Datentypen. Sie können keine Child-Elemente enthalten und sind built-in.
Derived Datentype sind durch einen bereits existierenden Datentypen und können Elemente und Attribute enthalten.

Elemente welche keine Child Elemente und keine Attribute haben werden als simple types bezeichnet. Alle Attribute müssen simple types sein.

```
<?xml version="1.0">
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="title" type="xs:string" />
 ...

</xs:schema>
</xml>
```

Komplexe Typen. Elemente mir Child Elementen oder Attributen sind definiert durch complex types. Complex type Definitionen enthalten eine Menge von Attribut- und Elementdeklaration, es gibt versch. Möglichkeiten: sequence (geordnet), all (nicht geordnet) und choice (Auswahl eines Elementes)

```
<xs:complexType name="personType">
 <xs:sequence>
  <xs:element name="surname" type="xs:string"/>
  <xs:element name="forename" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
```

Die so definierten complex types können auch wieder an anderen Orten für den Typ eines Elements benutzt werden.

Attribut Deklarationen

```
<xs:element name="publication">
 <xs:attribute name="type" type="pubtype">
 </xs:attribute>
</xs:element>
```

Man kann auch Constraints einfügen, wie z.B. die Anzahl Vorkommen. Dafür gibt es `minOccurs` und `maxOccurs`.

Userdefinierte Simple Types

Restriction

```
<xs:simpleType name="month">
 <xs:restriction base="xs:integer">
   <xs:minInclusive value="1" />
   <xs:maxInclusive value="12" />
 </xs:restriction>
</xs:simpleType>


<xs:simpleType name="pubtype">
 <xs:restriction base="xs:string">
  <xs:enumeration value="article" />
  <xs:enumeration value="book" />
  ...
 </xs:restriction>
</xs:simpleType>
```

Derivation, z.Bsp. Listen von Basetypes.

```
<xs:simpleType name="ListOfIntegers"
 base="integer" derivedBy="list">
 <xs:minLength value="1" />
 <xs:maxLength value="10" />
</xs:simpleType>
```

Schema Zusammenfügen

```
<include schemaLocation=
  "http://www.macbain.inf.ethz.ch/customer.xsd"/>

<redefine schemaLocation=
  "http://www.macbain.inf.ethz.ch/customer.xsd"/>
  <simpleType name="nameType">
  <restriction base="string">
    <maxLength value="40"/>
  </restriction>
 </simpleType>
</redefine>
```

12. Metamodell

Ein Metamodel ist ein Model von einem Modell.

12.1. Metadata und Metaschema

Schema und Metaschema sind die Beschreibung der Datenbank, Metadata und Data die eigentliche DB.

Das Datenbankschema kann auch als Werte in der Datenbank gespeichert sein - Metadata. Metaschema ist eine Beschreibung der Metadata. Ein Metaschema bietet die Regeln und Grammatiken für eine bestimmte Sprache.

13. Distributed Databases

13.1. Fragmentation

Horizontal Fragmentation

Man kann z.B. die Datenbank einer grosse Organisation, so fragmentieren, dass alle Geschäftssitze die Daten ihrer Stadt haben.

Vertical Fragmentation

Employees z.B. in zwei Datensätze aufteilen: name, grade, address und andererseits salary, taxcode.

Derived Horizontal Fragmentation

Unterteilung nach einer anderen Distribution. Z.B. Employees nach Städten fragmentieren, soferen Organisation nach Städten fragmentiert. Dafür benutzt man ein Semi-Join.

$$Employees \triangleright\!< \sigma_{city=Zurich}(Organisations)$$

13.2. Distribution

Ein Globales Schema und versch. verteilte Datenbanken mit unterschiedlichen Schemas. Bei einem Hetereogenen DDB werden die Queries gemappt von dem globalen Schema auf ein "kleineres" Modell, der User muss nicht über die Verteilung wissen. Bei einem Multi-Database System weiss der User aber über die versch. DB und es gibt kein Mapping.

A. Prolog

A.1. Einführung

[] entspricht dabei leerer Liste, $[A|R]$ einer Liste mit Kopfelement A, und Restliste R und $[a, b, c]$einer Liste mit den genannten Elementen. `app(X,Y,A)` hängt die Listen X und Y zu A zusammen, `rev(X,Y)` dreht X in Y um. `list(L)` ist eine Liste (ein geschlossener Term). `select(X,L1,L2)` löscht das erste Vorkommen von X in L1 und gibt L2 zurück. `delete(X,L1,L2)` löscht alle Vorkommen.

```
app([], L, L).
app([X|L1],L2,[X|L3]) :- app(L1,L2,L3).

rev([], []).
rev([X],[X]).
rev(L1,L2) :- app([X|LA], [Y], L1),
              app([Y|LB], [X], L2),
rev(LA, LB).

/*2te Variante*/
rev([],[]).
rev([X|L],M):-
 rev(L,N),
 app(N,[X],M).

/*1ter und letzter Eintrag loeschen*/
r(L1,L2) :- app([_|L2],[_],L1).

/*Permutation*/
perm([],[]).
perm([X|L1],[Y|L2]):-
 sel([X|L1],Y,L3),
 perm(L3,L2).

sel([X|L],X,L).
sel([Y|L1],X,[Y|L2]):-
 sel(L1,X,L2).

suffix(L1,L2) :- append(_,L1,L2).
prefix(L1,L2) :- append(L1,_,L2).

list([]).
list([_|L]) :- list(L).

sublist(L1,L2) :-
 suffix(L3,L2),
 prefix(L1,L3).

member(X,[X|_]).
member(X,[_|L]) :- member(X,L).

select(X,[X|L],L).
select(X,[Y|L1],[Y|L2]) :- select(X,L1,L2).

delete([],_,[]).
delete([Y|L1],X,L2) :-
 ( X=Y ->
  delete(L1,X,L2)
 ; L2=[Y|L3],
  delete(L1,X,L3)
 ).

/*loescht den zweiten Eintrag*/
second(X,[_,X|L]).

/*X letztes Element der Liste?*/
final(X,L) :- append(L1,[X],L).

/*oder*/
final(X,[X]).
final(X,[_|L]):-final(X,L).

/*L1 identisch mit L2, bis auf
Vertauschung von letztem und
erstem? */
swapf1([],[]).
swapf1([X],[X]).
equalminuslast([X],[Y]).
equalminuslast([X|L1],[X|L2]) :-
 equalminuslast(L1,L2).
swapf1([X|L1],[Y|L2]) :-
 final(X,[Y|L2]),
 final(Y,[X|L1]),
 equalminuslast(L1,L2).

/*auf 2x vorkommen testen*/
```

```
twice(X,L) :-
 append(_,[X|L1],L),
 member(X,L1).
```

A.2. Deklarative Konstrukte in Prolog

Rule = Atom ':-' Goal'.'

Goal = 'true' | 'fail' | Atom | Term '=' Term | Goal ',' Goal
| Goal ';' Goal | '\+' Goal | Goal '->' Goal ';' Goal

= : "gleich"
, : "und"
; : "oder"
\+ : "nicht"
->; : "if-then-else"