Grundlagen Datenbanken

1: Einleitung

1.1 – Überblick

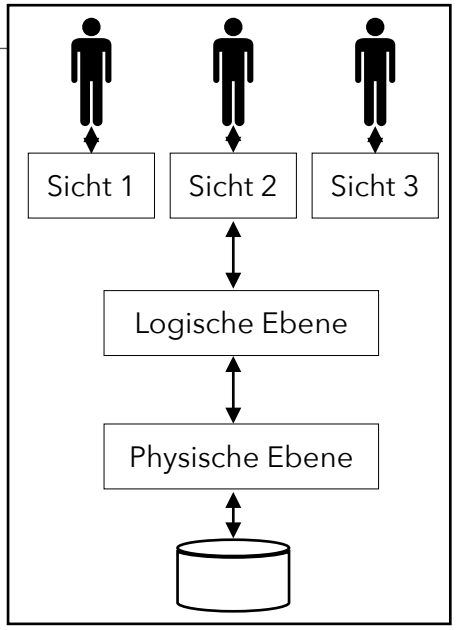
Was ist ein Datenbanksysteme?
Ein System zum Speichern und Verwalten von Daten.

Typische Probleme ohne DBS
- Redundanz und Inkonsizenz
- Verschiedene Datenformate
- Probleme bei Mehrnutzerbetrieb
- Verlust von Daten
- Integritätsverletung
- Sicherheitsprobleme
- Hohe Entwicklungskosten für Anwendungsprogramme

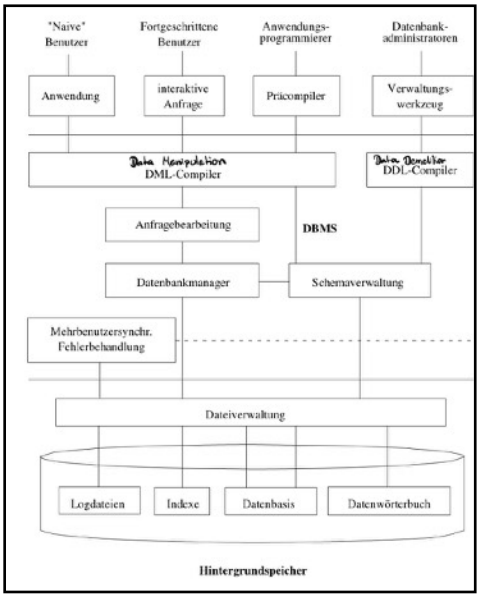
Gründe für den Einsatz von DBS
- Datenunabhängigkeit
- Deklarative Anfragensprache
  > Anfrage nach Daten, nicht Methodik des Datenabrufs. (> Datenunabhängigkeit)
- Mehrbenutzersynchronisation
  > DBS erlaubt gleichzeitigen Zugriff und verhindert Race Conditions.
- Fehlerbehandlung
  > Rekonstruktion von Daten nach Abstürzen, Logdateien
- Sicherstellung d. Datenintegrität
  > Erkennung von Benutzer- und Programmfehlern
- Effizienz und Skalierbarkeit
  > Konzipiert für große Anwendungen, gute Laufzeitverhalten $< O(n \log n)$

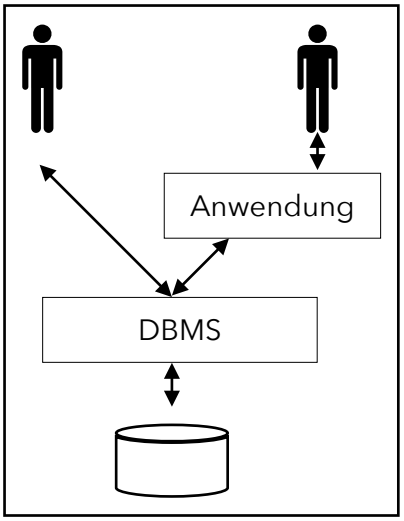
1.2 – Konzepte

Datenunabhängigkeit
- Entkopplung von Anwendung und Strukur der Datenspeicherung
- Logische Datenunabhängigkeit: Änderung auf logischer Ebene haben keinen Einfluss auf Anwendungen
  > schwierig
- Physische Datenunabhängigkeit: Änderung auf physischer Ebene haben keinen Einfluss auf Anwendungen
  > meist umgesetzt



Architektur





1.3 – Datenmodellierung

Datenmodellierung

- Datenmodell: welche Konstrukte zum Beschreiben von Daten
- Schema: konkrete Beschreibung einer konkreten Datensammlung

Datenmodelle

- Konzeptuelle Modelle
  - > Entity-Relationship-Modell (ER-Modell)
  - > Unified Modeling Language (UML)
- Logische Modelle
  - > Hierarchisches Modell
  - > Relationales Modell
  - > Objekt-orientiertes Modell, Objekt-relationales Modell

Schritte der Modellierung

1. Auschnitt aus der realen Welt
   > manuelle/intelektuelle Modellierung
2. Konzeptuelle Schemata (ER-Schema)
   > Halbautomatische Transformation
3. Relationales Schema/Netzwerkschema/objektorientieres Schema

2: Datenbankentwurf

2.1 – Anforderungsanalyse

Identifikation von Organisationseinheiten & Aufgaben

- Objektbeschreibung: Objekte (+ geschätzte Anzahl) & Attribute identifizieren
  > Attribute: Typ, Länge, Wertebereich, Anzahl Wiederholungen, Identifizierend (eindeutig?)
- Beziehungsbeschreibungen: beteiligte Objekte, Attribute, Anzahl
- Prozessbeschreibung: Häufigkeit, benötigte Daten, Priorität, Datenmenge

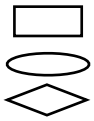
Spezifikation

Hier wird das ‚was' festgelegt (nicht das ‚wie').

- Bestandteile: Prozessbeschreibung, Use Cases, Benutzerschnittstellen, sonstige Schnittstellen
- Iterativer Prozess (oft auch politisch)
  a. Anwender beschreibt Entwickler, was er möchte
  b. Entwickler schreibt in seiner ‚Sprache' auf (was er verstanden hat)
  c. Anwender bekommt Entwurf und äußert Änderungswünsche
  d. Zurück zu b.

2.2 – Konzeptioneller Entwurf

Entity-Relationship Model



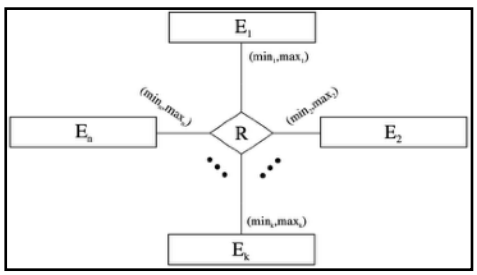
- Entitätstyp: Menge von Entitäten mit den gleichen Attributen
- Werttyp/Attribute: Eigenschaften von Beziehungen oder Entitäten
  > Nicht Mengenwertig > neue Entität
- Beziehungstyp: Beziehung zwischen Enitätstypen (Verben im infitiv)
  > Rollen: beschreiben in welcher Weise Entitäten an Beziehung teilnehmen
- Schlüssel: dient zur eindeutigen Identifizierung einer Entität
- Funktionalität: gibt Anzahl der verknüpften Objekte an
- Generalisierung: Vererbung der Eigenschaften an Untertyp
- Aggregation: ‚ist-Teil-von' Beziehung

Funktionalitäten

- Funktionalitäten drücken aus mit wieviel anderen Entitäten eine Entität eine Beziehung eingeht.

$$R \subseteq E_1 \times \ldots \times E_i \times \ldots \times E_n$$

- Für jedes $e_i \in E_i$ gibt es
  > Mindestens $\min_i$ Tupel der Art $(\ldots, e_i \ldots) \in R$
  > Höchstens $\max_i$ Tupel der Art $(\ldots, e_i \ldots) \in R$



Schwache Entitäten

- Schwache Entitäten können nicht als eigenständiges Gebilde existieren (vgl. Komposition)
- Braucht starke Identität in 1:n Beziehung, um identifiziert zu werden
  > Bsp. Gebäude, Zimmer

Entwurfsentscheidungen

- Entität vs. Attribut: Entitätstyp ist flexibler, Attribute sind einfacher
  > Entität bei Megenwertigkeit
  > Attribut in Bestandteile zerlegen > Entität mit Attributen (Bestandteile)
- Entität vs. Beziehung: Ebtutät ist flexibler, Beziehungen sind einfacher
  > Beziehung sind ohne Identität > diesselben Entitäten können nicht mehr als einmal eine Beziehung eingehen
- Binäre vs. Ternäre Beziehung: echte Ternäre Beziehung?
- Conncetion Trap: redundante zyklische Beziehungen vermeiden

Übersetzen von ER in relationales DBMS

- Übersetzen der Entitäten: jede Entität in eine Relation, Attribute der Entität werden Attribut der Relation (Schlüssel wird Primärschlüssel)
- Übersetzen von Beziehungen: jede Beziehung in Relation
  > N:M: Schlüssel setzt sich aus Kombination beider Entitätsschlüssel zusammen
  > 1:N: Schlüssel der N-Seite wird zum Schlüssel
  > 1:1: Schlüssel kann beliebig gewählt werden
  > Schemavereinfachung: Relationen mit dem gleichen Schlüssel in eine Relation
- Übersetzen von schwachen Entitäten: wie 1:N, aber zusammengesetzter Schlüssel
- Übersetzen von Generalisierung: keine äquivalente Übersetzung, 3 Möglichkeiten
  > Nur Obertyp, Problem: null Werte
  > Nur Untertyp, Problem: Obertyp kann nicht erstellt werden
  > Alle Typen, Problem: Informationen auf mehrere Relationen verteilt

3: Relationales Modell

3.1 – Definitionen

Definitonen

Eine relationale Datenbank enthält eine Menge von Relationen

- Instanz: Tabelle mit Zeilen (Tupel) und Spalten, aktueller Inhalt der Relation
- Schema: Spezifiziert Namen der Relation und Datentypen der Spalte (Attribut)
  > Assoziert mit jedem Attribut ist eine Domäne (Wertebereich)

```
Student(MatrNr:integer,
            Name:string,
            Geburtstag:date)
```

Formale Definiton

- Relation (Instanz): $R \subseteq D_1 \times D_2 \times \ldots \times D_n$
- Kardinalität: $|R|$ (Anzahl der Tupel)
- Schlüssel: eine minimale Menge von Attributen, deren Werte ein Tupel eindeutig indentifiziert

```
Student(MatrNr, Name, Geburtstag)
```

Anfragesprachen

- Relationenkalkül: deklarative Sprache, Grundlage für SQL
- Relationale Algebra: prozedual orientiert, Grundlage für DBMC-Plan (interne Verarbeitung)

3.2 – Relationale Algebra

Relationale Algebra

Alle Operatoren sind mengenorientiert und abgeschlossen.

- Operator bekommt als Eingabe eine/mehrer Mengen von Tupeln
- Gibt eine Menge von Tupeln aus

Projektion

Wählt Attribute $A_1, \ldots, A_n$ aus der Relation R und filtert alle anderen Attribute heraus.

- Duplikate werden eliminiert (Tupel sind eindeutig)

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

Selektion

Wählt alle Tupel aus R aus, die dsa Prädikat p erfüllen.

- Prädikate können mit logischen Operatoren ($\wedge$, $\vee$, $\neg$) und Vergleichsoperatoren ($=$, $<$, $>$, $\leq$, $\geq$) kombiniert werden

$$\sigma_p(R)$$

Kreuzprodukt

Zwei Relationen werden mit dem Kreuzprodukt verbunden (jeder mit jedem).

$$R_1 \times R_2$$

Join

Kombiniert Selektion und Kreuzprodukt zum sinnvollen Verknüpfen zweier Relationen.

- Joins sind kommutativ und assoziativ
- Attributnamen müssen eindeutig sein, das erfordert teilweise Umbenennung von Attributen (ρ)

  $R_1 \bowtie_{R_1.A_i=R_2.A_j} R_2 = \sigma_{R_1.A_i=R_2.A_j}(R_1 \times R_2)$ ρ newName $\leftarrow$ oldName

  $\bowtie_{A=B}$ Equi-Join (nur Gleichheit)

  $\theta_{A<B}$ Theta-Join (erlaubt Vergleichsoperatoren)
- Natürlicher Join: Equi-Join, der nur Attribute mit gleichen Namen vergleicht (und redundante Spalten wegprojiziert), geschrieben $A \bowtie B$

Weitere Joinarten

Hier gilt die Kommutativität und Assoziativität unter Umständen nicht.

- Äußerer Join: Tupel ohne Relation bleiben erhalten $\mathbin{⟗}$
- Linker äußerer Join: Tupel aus der linken Relation ohne Relation bleiben erhalten $\mathbin{⟕}$
- Rechter äußerer Join: Tupel aus der rechten Relation ohne Relation bleiben erhalten $\mathbin{⟖}$
- Linker Semi-Join: Joinbedingung wird geprüft, es werden aber nur Tupel aus der linken Relation erhalten $\ltimes$ ist auch als Rechter Semi-Join möglich $\rtimes$
- Linker Anti-Join: Prüft Joinbedinung, behält nur Tupel aus linker Relation, die die Bedingung nicht erfüllen (keinen Joinpartner haben) $\triangleright$ ist auch als Rechter Anti-Join möglich $\triangleleft$

Vereinigung, Schnitt und Differenz

Falls beide Seite dasselbe Schema (gleiche Reihenfolge) haben, können Relationen

- Vereinigt werden (konkateniert)

  $R_1 \cup R_2$
- Geschnitten werden (Tupel, die in beiden Relationen sind)

  $R_1 \cap R_2$
- Subtrahiert werden (Mengendifferenz) (Tupel die in R_1, aber nicht in R_2 sind)

  $R_1 \setminus R_2$

Relationale Division

Mit der relationalen Division kann man das Kreuzprodukt rückgängig machen.

$$R_1 \div R_2 = \pi_{(\mathscr{R}_1 \setminus \mathscr{R}_2)}(R_1) \setminus \pi_{(\mathscr{R}_1 \setminus \mathscr{R}_2)}(\pi_{(\mathscr{R}_1 \setminus \mathscr{R}_2)}(R_1) \times R_2 \setminus R_1)$$

3.3 – Relationenkalkül

Relationenkalkül

Der Relationenkalkül ist stärker deklarativ orientiert (es werden Ergebnistupel ohne Herleitungsvorschrift beschrieben). Es gibt zwei verschiedene Arten

> Relationaler Tupelkalkül
> Relationaler Domänenkalkül

Relationaler Tupelkalkül

Eine Anfrage im Relationenkalkül hat die Form $\{s \mid P(t)\}$, wobei t Tupelvariable und P Formel ist.

Bsp: $\{s \mid s \in$ Studenten
$\wedge \exists h \in$ hören(s . MatrNr $= h$. MatrNr
$\wedge \exists v \in$ Vorlesungen(h . VorlNr $= v$. VorlNr
$\wedge \exists p \in$ Professoren(p . PersNr $= v$. gelesenVon
$\wedge p$. Name = "Pythagoras"))))$\}$

Relationaler Tupelkalkül: Formale Definition

Atome
- $s \in R$, mit s Tupelvariable und R Relationenname
- $s . A \square t . B$, mit s und t Tupelvariable, A und B Attributnamen und $\square$ Vergleichsoperator
- $s . A \square c$ mit c Konstante

Formeln
- Alle Atome sind Formeln
- Ist P Formel, so auch (P) und $\neg P$
- Sind P_1 und P_2 Formeln, so auch $P_1 \wedge P_2$, $P_1 \vee P_2$ und $P_1 \rightarrow P_2$
- Ist $P(t)$ Formel mit freier Variable t, so auch $\forall t \in R(P(t))$ und $\exists t \in R(P(t))$

Relationaler Domänenkalkül

Eine Anfrage im Domänenkalkül hat die Form $\{[v_1, \ldots, v_n] \mid P(v_1, \ldots, v_n)\}$, wobei $v_1, \ldots, v_n$ Domänenvariable sind und P Formel ist.

Bsp: $\{[m, n] \mid \exists s([m, n, s] \in$ Studenten
$\wedge \exists p, v, g([m, p, v, g] \in$ prüfen
$\wedge \exists a, r, b([p, a, r, b] \in$ Professoren
$\wedge a =$ 'Sokrates'))))$\}$

Relationaler Domänenkalkül: Formale Definition

Atome
- $[w_1, \ldots, w_n] \in R$, mit m-stelliger Relation R und Domänenvariablen $w_1, \ldots, w_n$
- $x \square y$, mit x und y Domänenvariablen, $\square$ Vergleichsoperator
- $x \square c$, mit Konstante c

Formeln
- Alle Atome sind Formeln
- Sind P_1 und P_2 Formeln, so auch $P_1 \wedge P_2$, $P_1 \vee P_2$ und $P_1 \rightarrow P_2$
- Ist $P(v)$ Formel mit freier Variable v, so auch $\forall v \in (P(v))$ und $\exists v \in (P(v))$

Sicherheit

Bedingung für Sicherheit: Ergebnis des Ausdrucks muss Teilmenge der Domäne der Formel sein.

$\{s \mid \neg(s \in S)\}$ $\{[p] \mid \neg([p] \in P)\}$ unsicher!

4: SQL

4.1 – Einfache Anfragen

Structured Query Language (SQL)

SQL ist eine standardisierte deklarative Anfragesprache für relationale Datenbanksysteme.

- DRL: Data Retrieval Language
- DML: Data Manipulation Language
- DDL: Data Definition Language
- DCL: Data Control Language

Einfache Anfragen

```
select <Liste von Attributen> from <Liste von Relationen> where <prädikat>;
```

Duplikateliminierung

SQL eliminiert keine Duplikate. Wird Duplikateliminierung gewünscht, verwendet man das Schlüsselwort distinct:

```
select distinct <Attribute>
```

Prädikate

- Prädikate in der ‚where'-Klausel können logisch kombiniert werden (¬, ∨ , ∧ ; Bindungsstärke).
  > NOT ¬, OR ∨, AND ∧
- Vergleichsoperatoren: = , < , > , < = , = >
  > beetween <start wertebereich> and <ende wertebereich>
  > like <Stringvergleich mit ‚Jokern' _ (bel. Zeichen) & %(bel. Zeichenkette)>

Nullwerte

- SQL hat für unbekannte, nicht verfüg- oder anwendbare Werte den speziellen Wert null.
  ```
  select * from Student where Geburtstag is NULL;
  ```
- Nullwerte werden in arithmetischen & logischen Ausdrücken durchgereicht
  > ein Operand NULL > Ergebnis NULL
  > logische Ausdrücke können zu wahr, falsch und unbekannt evaluieren

Kommentare

SQL Abfragen kann mit ‚– –, Kommentieren

```
select * -- einzeiliger Kommentar
```

Qualifizierte Attributnamen

Um Attribute ihrer ursprünglichen Relation zuzuordnen, wird eine Punkt-Notation verwendet

```
select * from Student, besucht, Vorlesung
where Student.MatrNr = besucht.MatrNr and besucht.Nr = Vorlesung.Nr;
```

Relationen umbennen

Relationen können in der ‚from'-Klausel umbenannt werden

```
select * from Student S, <alter Name> <neuer Name>;
```

Sortierung

Die Ergebnistupel sind nicht (automatisch) sortiert. Mit der ‚order-by'-Klausel kann man nach einem (angeordneten) Attribut auf- oder absteigend sortieren.

```
select * from Student order by Geburtstag desc; (alt. asc)
```

4.2 – Mehrere Relationen

Kreuzprodukt

Die einfache Verknüpfung zweier Relationen ohne ‚where'-Klausel erzeugt ein Kreuzprodukt.

```
select *
from Vorlesung, Professor -- Kreuzprodukt;
```

Joins

- Joinprädikate werden in der ‚where'-Klausel angegeben

```
select *
from Vorlesung, Professor
where ProfPersNr = PersNr;
```

- Verschiedene Joinarten der relationalen Algebra sind auch in SQL möglich

```
select *
from R1 [cross|inner|natural|left outer|right outer|full outer] join R2
[on R1.A = R2.B];
```

Mengenoperationen

SQL verfügt auch über die üblichen Mengenoperationen (∩ , ∪ , \)

> **union** ∪, **intersect** ∩, **except** \ (, **union all**)
>
> Duplikateliminierung: bei union werden Duplikate eliminiert (außer mit union all)

4.3 – Geschachtelte Anfragen

Geschachtelte Anfragen

Anfragen können inneinander geschachtelt sein (d.h. es kann mehrere ‚select'-Klauseln geben).

- Beim Schachteln eines ‚select's in einer ‚select'-Klausel darf nur ein Tupel mit einem Attribut zurückgeliefert werden
- Das Schachteln von korrelierten Unteranfragen in ‚from'-Klauseln ist in vielen DBMS nicht erlaubt

Unorrelierte Unteranfrage

Unteranfrage wird einmal ausgewertet und für alle Tupel der äußeren Anfrage geprüft.

```
select s.Name from student s where s.MatrNr in (
      select b.MatrNr from besucht b where b.Nr = 5);
```

Korrelierte Unteranfrage

Unteranfrage hat für jedes Tupel der äußeren Anfrage verschiedene Werte (da auch auf Attribute aus dem äußeren Abruf zugegriffen wird) und muss deshalb jedes mal neu ausgewertet werden.

```
select distinct P.Name from Professor P, Assistent A
where A.Boss = P.PersNr and exists (select * from Assistent B
      where B.Boss = P.PersNr and A.Fachgebiet <> B.Fachgebiet);
```

4.4 – Aggregatfunktionen

Aggregatfunktionen

Attributwerte (oder ganze Tupel) können auf verschiedene Arten zusammengefasst werden

```
> count()  --Zählen
> sum()    --Aufsummieren
> avg()    --Durchschnitt bilden
> max()    --Maximum finden
> min()    --Minimum finden
    select count(*) from Student; select count(distinct GebTag) from Student;
```

Minimun & Maximum

Die Mini- und Maximum Funtkionieren reduzieren alle Werte einer Spalte zu einem einzigen Wert.

```
    select MatrNr, Name from Student where MatrNr = (
          select max(MatrNr) from Student);
```

Gruppieren

- Man kann Tupel in verschiedene Gruppen aufteilen und die Gruppen getrennt aggregieren.

```
    select Nr, count(*) as Anzahl from besucht group by Nr;
```

- Attribute, die nicht in der ‚group by'-Klausel auftauchen, dürfen nur aggregiert in der ‚select'-Klausel vorkommen

```
    select PersNr, ~~Titel,~~ count(*) as Anzahl from Vorlesung group by PersNr;
```

Having

Da die ‚where'-Klausel vor dem gruppieren ausgewertet wird, gibt es mit der ‚having'-Klausel (die später ausgewertet wird) ein ‚where' für Gruppen.

```
    select PersNr, count(Nr) as AnzVorl from Vorlesung group by PersNr
    having count(*) > 3 --Prof, die mehr als drei Vorlesungen halten
```

4.5 – Sichten

Sichten

Sichten gehören zur DDL und sind eine Art virtuelle Relation.

- Evaluation: Vereinfachen Zugriff, können Zugriffe einschränken (+), Änderungsoperationen sind teilweise nicht möglich (–)

```
    create view ÜberSchnittCredit as select Nr, ProfPersNr from Vorlesung
    where Credits > (select avg(Credits) from Vorlesung);
```

4.6 – Daten Manipulieren

Daten Einfügen

Daten können mit dem insert Befehl eingefügt werden

```
    insert into <Relation>
    values(<values>)
```

Die einzufügenden Daten können auch aus einer anderen Relation kopiert werden

```
    insert into <Relation>
    select <Abfrage>
```

Daten Ändern

Mit dem update-Befehl können Änderungen vorgenommen werden

```
update <Relation>
set <attrValue>
where <condForTupel>
```

Daten Löschen

Mit dem delete-Befehl löscht man Daten aus Relationen

```
delete from <Relation>
where <condForTupel> -- ohne where wird die gesamte Relation gelöscht
```

4.7 – Daten/Relationen Anlegen

Relationen Anlegen

Mit dem create-Befehl können Relationen angelegt werden

```
create table <name> (
 <varname> <datentyp>
 ...
);
```

Relationen & Sichten löschen

Mit dem drop-Befehl kann man Relationen, Sichten und Indexe wieder entfernen.

```
drop table <relation>
drop view <view>
drop index <index>
```

Schlüssel definieren

Für jede Relation kann/sollte ein Primärschlüssel definiert werden

```
create table <name> (
 <varname> <datentyp> primary key -- Primärschlüssel
 <varname> <datentyp> references <varname> -- Fremdschlüssel einer anderen
                                           -- Relation
 foreign key <varname> <datentyp> references <varname> - -
 ...
);
```

Integritätsbedingung

Im Datenbanksystem muss die Konsistenz der Daten sichergestellt werden. Dafür können Constraints angelegt werden.

```
not null          check(<cond>)
```

Referentielle Integrität

```
set null          cascade           restrict
```

Indexe

Indexe beschleunigen den Zugriff auf Relationen (verlangsamen aber Einfügen).

```
create [unique] index <indexname>
on table <relationName> (<attr> [asc|desc]);
```

5: Relationale Entwurfstheorie

5.1 – Funktionale Abhängigkeit

Formale Definition

- α und β sind Attributmengen eines relationalen Schemas $\mathscr{R}$
- Es gibt eine FD : $\alpha \to \beta \iff \forall$ Instanzen $\in \mathscr{R} : r, t \in R \quad r.\alpha = t.\alpha \Rightarrow r.\beta = t.\beta$

Funktionale Abhängigkeiten werden aus Hintergrundwissen über die Anwendung abgeleitet.

Armstrong Axiome

Für $\alpha, \beta, \gamma \in \mathscr{R}$

$\beta \subseteq \alpha \Rightarrow \alpha \to \beta$	Reflexivität
$\alpha \to \beta \Rightarrow \alpha \cup \gamma \to \beta \cup \gamma$	Verstärkung
$\alpha \to \beta \wedge \beta \to \gamma \Rightarrow \alpha \to \gamma$	Transitivität
$\alpha \to \beta \wedge \alpha \to \gamma \Rightarrow \alpha \to \beta \cup \gamma$	Vereinigungsregel
$\alpha \to \beta \cup \gamma \Rightarrow \alpha \to \beta \wedge \alpha \to \gamma$	Dekompositionsregel
$\alpha \to \beta \wedge \gamma \cup \beta \to \delta \Rightarrow \alpha \cup \gamma \to \delta$	Pseudotransitivitätsregel

Attributhüllen

Die Attributhülle einer Attributmenge α ist die Menge aller funktional abhängigen Attribute von α. $\text{AH}(\alpha) \stackrel{?}{=} \mathscr{R} \implies \alpha$ ist pot. Schlüssel

Schlüsselberechnung

Eigenschaften von Schlüsseln

1. $\kappa \subseteq \mathscr{R}$
2. $\kappa \to \mathscr{R}$ (Vollständigkeit)
3. $\neg \exists \kappa' \subset \kappa : \kappa' \to \mathscr{R}$ (Minimalität)

- Superschlüssel: Wenn nur Eigenschaften 1 & 2 gelten, wird κ Superschlüssel genannt.
- Kandidatenschlüssel: Wenn Eigenschaften 1, 2 & 3 gelten, wird κ Kandidatenschlüssel genannt.
  > Primärschlüssel: Ein Kandidatenschlüssel κ wird als Primärschlüssel gewählt.

5.2 – Schemaqualitätsprüfung

Schemaqualitätsprüfung

«*Data depends on the key [1NF], the whole key [2NF] and nothing but the key [NF3].*»
Normalformen ordnen Schemata nach ihrer Qualität.

1NF ⊂ 2NF ⊂ 3NF ⊂ BCNF ⊂ 4NF

Erste Normalform (1NF)

Ein relationales Schema ist in 1NF $\iff$ alle Attribute nehmen nur Atomare Werte an.

Zweite Normalform (2NF)

Ein relationales Schema ist in 2NF $\iff$ 1NF $\wedge$ jedes Nichtschlüsselattribut (NSA) ist voll funktional von jedem Schlüssel abhängig.

- β hängt voll funktional von α ab, falls $\neg \exists \alpha' \subset \alpha : \alpha' \to \beta$

Dritte Normalform (3NF)

Ein relationales Schema ist in 3NF $\iff$ 2NF $\wedge$ für jede funktionale Abhängigkeit gilt

$\alpha \to \beta$ ist trivial : $\beta \subseteq \alpha \ \vee \ \alpha$ ist Superschlüssel $\vee$ jedes Attribut in β ist in einem Schlüssel

- i.a.W.: keine transitiven Abhängigkeiten, in denen NSA von NSA abhängen

Boyce-Codd Normalform (BCNF)

Ein relationales Schema ist in BCNF $\iff$ 3NF $\wedge$ alle Attribut hängen direkt vom Schlüssel ab

$\alpha \to \beta$ ist trivial : $\beta \subseteq \alpha \ \vee \ \alpha$ ist Superschlüssel

- i.a.W.: es gibt auch keine NSA, die S bestimmen

Mehrwertige Abhängikeiten (MVDs)

Fähigkeiten		
PersNr	Sprache	ProgSprache
3002	Englisch	C
3002	Deutsch	C
3002	Englisch	Java
3002	Deutsch	Java
3005	Englisch	C
3005	Deutsch	C

In dem Schema (siehe rechts) gibt es mehrwertige Abhängigkeiten (PersNr $\twoheadrightarrow$ Sprache, PersNr $\twoheadrightarrow$ ProgSprache), einer Person werden mehrere Sprachen und auch mehrere Programmiersprachen zugeordnet.
$\alpha, \beta, \gamma \subseteq \mathcal{R} : \alpha \cup \beta \cup \gamma = \mathcal{R}$, $\alpha \twoheadrightarrow \beta$ gilt, wenn für jede Instanz in R gilt: für jedes Paar von Tupeln $t_1, t_2 \in R$ mit $t_1 . \alpha = t_2 . \alpha$ existiert $t_3 \in R$ mit $t_3 . \alpha = t_1 . \alpha, t_3 . \beta = t_1 . \beta, t_3 . \gamma = t_1 . \gamma$.

- i.a.W.: für jedes Tupel mit gleichen Wert für α kommen alle β, γ-Kombinationen vor.
- $\alpha \twoheadrightarrow \beta \Rightarrow \alpha \twoheadrightarrow \gamma \ : \ \gamma = \mathcal{R} - \alpha - \beta$

Vierte Normalform (4NF)

Ein relationales Schema ist in 4NF $\iff$ BCNF $\wedge$ für jede MVD $\alpha \twoheadrightarrow \beta$ werden ebenfall die Eigenschaften der BCNF angewandt:

$\alpha \twoheadrightarrow \beta$ ist trivial : $(\beta \subseteq \alpha \vee \alpha \cup \beta = \mathcal{R}) \ \vee \ \alpha$ ist Superschlüssel

---

5.3 – Schemazerlegung

Verlustlosigkeit

Beim Zerlegen soll Verlustlosigkeit gelten: die in der ursprünglichen Instanz R enthaltene Information muss aus den Instanzen $R_1, \ldots, R_n$ rekonstruierbar sein ($\forall R_i$).

- Hinreichende Bedingung für verlustlose Zerlegung: $(\mathcal{R}_1 \cap \mathcal{R}_2) \to \mathcal{R}_i \in \mathcal{F}^+_{\mathcal{R}} : i \in \{1,2\}$

Abhängigkeitsbewahrung

Alle Funktionalen Abhängigkeiten in $\mathcal{F}_{\mathcal{R}}$ sollten in den $\mathcal{F}_{\mathcal{R}i}$ bewahrt bleiben.

- Hinreichende Bedingung für Abhängigkeitsbewahrung
  - > $\mathcal{F}_{\mathcal{R}} \equiv (F_{\mathcal{R}1} \cup \ldots \cup F_{\mathcal{R}n})$
  - > $\mathcal{F}^+_{\mathcal{R}} = (F_{\mathcal{R}1} \cup \ldots \cup F_{\mathcal{R}n})^+$

Kanonische Überdeckung

$\mathcal{F}_C$ heißt kanonische Überdeckung von $\mathcal{F}$, wenn die folgenden Kritieren erfüllt sind:

> $\mathcal{F}_C \equiv \mathcal{F}$, d.h. $\mathcal{F}_C^+ = \mathcal{F}^+$
> In $\mathcal{F}_C$ existieren keine FDs $\alpha \to \beta$, bei denen α oder β überflüssige Attribute enthalten.
> Jede linke Seite einer FD in $\mathcal{F}_C$ ist einzigartig (sukzessive Anwendung d. Vereinigungsregel).

1. Überflüssige Attribute durch Links- und Rechtsreduktion entfernen
2. Entfernen aller funktionalen Abhängigkeiten $\alpha \to \emptyset$
3. Mit Vereinigungsregel alle $\alpha \to \beta_1, \ldots, \alpha \to \beta_n$ zu $\alpha \to \bigcup_{i=1}^{n} \beta_i$

3NF–Synthesealgorithmus

1. Forme ein Unterschema für jede Funktionale Abhängigkeit
2. Füge ein Schema $\mathcal{R}_\kappa$ mit einem Kandidatenschlüssel hinzu
3. Eliminiere redundante Schemata

Zerlegung in BCNF & 4NF

- Zerlegungen sind nicht immer abhängigkeitsbewahrend
- BCNF & 4NF bevorzugen Updateoperationen vor Leseoperationen (entspricht nicht der Nutzung von Datenbanken)

BCNF

1. Starte mit $Z = \{\mathcal{R}\}$
2. Solange es noch ein $\mathcal{R}_i \in Z$ gibt, dass nicht in BCNF ist
   > Finde eine FD $(\alpha \to \beta) \in F^+$ mit
   $\alpha \cup \beta \subseteq \mathcal{R}_i$
   $\alpha \cap \beta = \emptyset$
   $\alpha \to \mathcal{R}_i \notin F^+$
   > Zerlege $\mathcal{R}_i$ in $\mathcal{R}_{i1} := \alpha \cup \beta$ und $\mathcal{R}_{i2} := \mathcal{R}_i - \beta$
   > Entferne $\mathcal{R}_i$ aus Z und füge $\mathcal{R}_{i1}$ und $\mathcal{R}_{i2}$ ein
   $Z := (Z - \{\mathcal{R}_i\}) \cup \{\mathcal{R}_{i1}\} \cup \{\mathcal{R}_{i2}\}$

4NF

1. Starte mit $Z = \{\mathcal{R}\}$
2. Solange es noch ein $\mathcal{R}_i \in Z$ gibt, dass nicht in 4NF ist
   > Finde eine MVD $\alpha \twoheadrightarrow \beta \in \mathcal{F}^+$ mit
   $\alpha \cup \beta \subset \mathcal{R}_i$
   $\alpha \cap \beta = \emptyset$
   $\alpha \to \mathcal{R}_i \notin \mathcal{F}^+$
   > Zerlege $\mathcal{R}_i$ in $\mathcal{R}_{i1} := \alpha \cup \beta$ und $\mathcal{R}_{i2} := \mathcal{R}_i - \beta$
   > Entferne $\mathcal{R}_i$ aus Z und füge $\mathcal{R}_{i1}$ und $\mathcal{R}_{i2}$ ein
   $Z := (Z - \{\mathcal{R}_i\}) \cup \{\mathcal{R}_{i1}\} \cup \{\mathcal{R}_{i2}\}$

6: Physische Datenorganisation

6.1 – Speicherhierarchie

Speicherhierachie
Verschiedene Schichten der Speicherung, nach oben hin schneller, teurer und kleiner

CPU-Register ↔ Cache ↔ Hauptspeicher ↔ Festplatten ↔ Bänder

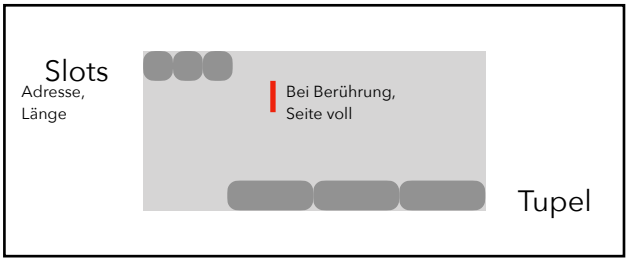
Aufbau einer Festplatte
Platte mit Arm und Lesekopf; Lesen
- Positionierung des Kopfes (Seek-Time)
- Rotation zum Anfang des Blockes auf Platte (Latenzzeit)
- Lesen des Blocks (Lesezeit)

Raid
- RAID0: Striping, Verteilung der Datensätze auf mehrere Festplatten
- RAID1: Spiegelung
- RAID0+1: Striping & Spiegelung
- RAID3/4: Striping mit Parity-Platte ($A, B, A \oplus B$)
- RAID5: Striping mit verteilter Parity (Ziel)

6.2 – Speicherung von Relationen

Datenbankpuffer
Daten werden in gleich große Seiten aufgeteilt, die in den Hauptspeicher geladen werden
- Pufferung: Daten werden bei Bedarf seitenweise in den Hauptspeicher geladen
- Verdrängung: Seiten werden aus vollem Datenbankpuffer verdrängt, wenn es keinen Platz mehr gibt (Least Recently Used, Hinting)

Speicherung von Relationen
- Tupel der Relationen werden auf Seiten im Hintergrundspeicher gespeichert
- Jedes Tupel kann über eine TID (Tupel-Identigikator) referenziert werden
- Verdrängung innerhalb von Seiten
  > Tupel passt nichtmehr auf Seite (nach update)
  > Tupel wird auf nächste freie Seite geschrieben, dabei bleibt der Slot gleich, das Tupel auf der alten Sleite zeigt auf den zusätzlichen Slot auf der neuen Seite



6.3 – Indexstrukturen

Indexstrukturen
Zum effizienten Zugriff auf Daten werden Indexstrukturen (Bäume/Hashing) angelegt.

Indexstruktur anlegen

```
CREATE INDEX <name> on <relation>(<atrCSV>);
```

Hierarchische Indexe: ISAM

- Tupel auf indexierten Attribut sortieren und eine Indexdatei darüber anlegen
- Vergleiche Daumenindex eines Buches, z.B. Telefonbuch
- Evaluation: (+) einfach, Bereichsanfragen möglich; (–) Indexinstandhaltung teuer

Hierarchische Indexe: B-Baum

Idee: Indexseiten für Indexseiten → B-Baum

- Jeder Pfad von der Wurzel zu einem Blatt hat die gleiche Länge
- Jeder Knoten (außer der Wurzel) hat mindestens i und höchstens $2i$ Einträge
- Einträge in jedem Knoten sind sortiert
- Jeder Knoten mit n Einträgen hat $n + 1$ Kinder

$[p_0, k_1, p_1, \ldots, k_n, p_n]$ p_j sind Zeiger, k_j Schlüssel $k_n < t < k_{n+1} : \forall t \in \text{SubTree}(p_n)$

Einfügealgorithmus

1. Finde den richtigen Blattknoten, um den neuen Schlüssel einzufügen
2. Füge Schlüssel dort ein
3. Falls kein Platz mehr da
   3.1. Teile Knoten und ziehe Median heraus
   3.2. Füge alle Knoten kleiner als Median nach links, alle größer als Median rechts hinzu
   3.3. Füge Median in Elternknoten und passe Zeiger an
4. Falls kein Platz in Elternknoten
   4.1. Falls Wurzelknoten: kreiere neuen Wurzelknoten und füge Median ein, passe Zeiger an
   4.2. Ansonsten wiederhole 3. mit Elternknoten

Löschalgorithmus

1. In einem Blattknoten kann ein Schlüssel einfach gelöscht werden
2. Falls kein Blattknoten, muss Invariante bestehen bleiben
   2.1. Nach nächstgrößerem/kleineren Schlüssel in Kindknoten suchen
   2.2. Schlüssel wird an Stelle des gelöschten Schlüssels geschrieben
3. Nach Löschen eines Schlüssels kann ein Knoten unterbelegt sein ($< i$)
   3.1. Knoten wird mit Nachbarknoten verschmolzen
   3.2. Prüfe Elternknoten, eventuell auch verschmelzen (Performance!)

Hierarchische Indexe: B⁺-Bäume

Performanceoptimierung: Hoher Verzweigungsgrad führt zu niedrigeren Bäumen.

- B⁺-Bäume speichern nur Referenzschlüssel in inneren Knoten, Daten in Blattknoten
- Blattknoten sind verkettet, damit sequentielle Suchen/Bereichszugriffe effizienter sind

Partitionierung: Hashing

- Hashing mit Chaining: Bie Kollisionen wird einfach ein weitere Bucket angehangen
- Erweiterbares Hashing: Hashfunktion erweitern, sodass übergelaufene Buckets weiter aufgeteilt werden

Ballung/Clustering

Daten, die häufig zusammen abgerufen werden (Relation), werden auf einer Seite gespeichert.

- Clustering geht mit der Zeit kaputt
- Verzahntes Clustering: Clustern von Relationen, die häufig gejoint werden

7: Anfragebearbeitung

7.1 – Übersetzung

Übersetzung

Die SQL (deklarativ) Anfrage muss möglichst effizient in relationale Algebra (prozedual) übersetzt werden.

Anfrage → Übersetzer → Ausführungsplan → Laufzeitsystem → Ergebnis

Standardübersetzung

Die Standardübersetzung ist häufig sehr ineffizient und bedarf deshalb einer Optimierung.

- Select: ganz oben als Projektion
- (order by)
- (Having: siehe Selektion, aber erst nach Gruppierung)
- (Aggregatfunktionen & group by: Gruppieren Γ und nicht gruppierte Attribute aggregieren)
- Where: in der Mitte als Selektion
- From: ganz unten als Kreuzprodukt aller Eingaberelationen

7.2 – Logische Optimierung

Potentiale

Transformation des Ausdruck (in relationaler Algebra) zu einem äquivalenten, der zu einem schnellerem Ausführungsplan führt > möglichst kleine Ausgaben von den einzelnen Operatoren.

- Aufbrechen von Selektionen ($\sigma_{p_1 \wedge \ldots \wedge p_n}(R) \equiv \sigma_{p_1}(\sigma_{p_2}(\ldots(\sigma_{p_n}(R))\ldots))$
- Verschieben von Seletionen (nach unten, sodass die Ergebnisse schneller klein werden)
- Selektion und Kreuzprodukte zu Joins zusammenfassen
- Joinreihenfolge optimieren
- Projektionen einfügen (möglichst früh)

Äquivalenzumformungen

$R_1 \bowtie R_2 \equiv R_2 \bowtie R_1$ $\quad$ $R_1 \cup R_2 \equiv R_2 \cup R_1$

$R_1 \cap R_2 \equiv R_2 \cap R_1$ $\quad$ $R_1 \times R_2 \equiv R_2 \times R_1$

$(R_1 \bowtie R_2) \bowtie R_3 \equiv R_1 \bowtie (R_2 \bowtie R_3)$ $\quad$ $(R_1 \cup R_2) \cup R_3 \equiv R_1 \cup (R_2 \cup R_3)$

$(R_1 \cap R_2) \cap R_3 \equiv R_1 \cap (R_2 \cap R_3)$ $\quad$ $(R_1 \times R_2) \times R_3 \equiv R_1 \times (R_2 \times R_3)$

$\sigma_{p_1 \wedge \ldots \wedge p_n}(R) \equiv \sigma_{p_1}(\sigma_{p_2}(\ldots(\sigma_{p_n}(R))\ldots)$ $\quad$ $\Pi_I(\sigma_p(R)) \equiv \sigma_p(\Pi_I(R))$, falls attr$(p) \subseteq I$

$\Pi_{I_1}(\Pi_{I_2}(\ldots(\Pi_{I_n}(R))\ldots) \equiv \Pi_{I_1}(R)$, mit $I_1 \subseteq I_2 \subseteq \ldots \subseteq I_n \subseteq \mathscr{R} = \text{sch}(R)$

7.3 – Physische Optimierung

Potentiale

Optimierung auf physischer Ebene kann durch die Verwendung von Indizes oder mittels Zwischenergebnisen realisiert werden.

Iteratorkonzept

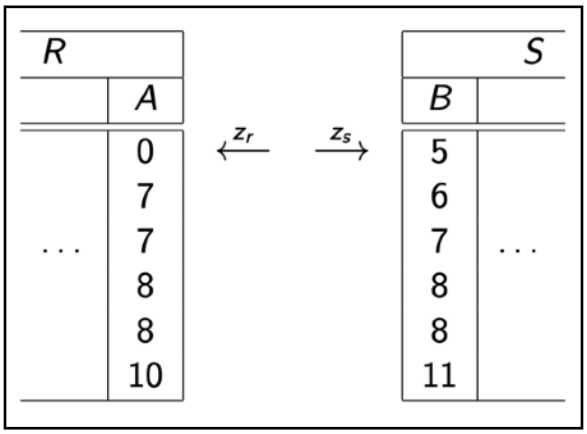
Jede Relation erhält einen Iterator mit den Operationen open, next, close, size und cost.

- open: initialisiert den Iterator
- next: gibt das nächste Element
- close: beendet die Iteration

NestedLoop-Join

- `Open`
  > Öffne die linke Eingabe
- `next`
  > Rechte Eingabe geschlossen
    > öffne sie
  > Fordere rechts solange Tupel an, bi Bedingung p erfüllt ist
  > Falls rechte Eingabe erschöpft
    > schließe rechte Eingabe
    > hole nächstes Tupel von linker Eingabe
    > starte `next` neu
  > Gib den Verbund von aktuellem linken und aktuellem rechten Tupel zurück
- `close`
  > Schließe beide Eingabequellen

(Sort-)Merge-Join (nur Equi-Join, input sortiert)



- `Open`
  > Öffne beide Eingaben
  > Setze ‚akt' auf linke Eingabe
  > Markiere rechte Eingabe
- `next`
  > Solange Bedingung p nicht erfüllt
    > setze ‚akt' auf Eingabe mit dem kleinsten anliegenden Wert im Joinattribut
    > rufe `next` auf ‚akt' auf
    > markiere andere Eingabe
  > Verbinde linkes und rechtes Tupel
  > Bewege andere Eingabe vor
  > Ist Bedingung nicht mehr erfüllt oder andere Eingabe erschöpft
    > bewege ‚akt' vor
    > Wert des Joinattributs in ‚akt' verändert?
    $\Longrightarrow$ markiere die andere Eingabe
    $\not\Longrightarrow$ setze andere Eingabe zurück auf Markierung
- `close`
  > Schließe beide Eingabequellen

Index-Join

- `Open`
  > Öffne die linke Eingabe
  > Hole erstes Tupel aus linker Eingabe
  > Schlage Joinattributwert im Index nach
- `next`
  > Bilde Join, falls Index (weiteres) Tupel zu diesem Wert liefert
  > Sonst, bewege linke Eingabe vor und schlage Joinattributwert im Index nach
- `close`
  > Schließe die Eingabe

Iterator Selektion

Ohne Index

- `open`
- `next`
  > Hole solange nächstes Tupel, bis eines die Bedinung p erfüllt, gib dieses Tupel zurück
- `close`

Mit Index

- `open`
  > Schlage im Index das erste Tupel nach, dass die Bedingung erfüllt, öffne Eingabe
- `next`
  > Gib solange das nächste Tupel zurück, bis die Bedingung p nicht mehr erfüllt ist
- `close`

Grace-Hash-Join

- Bei Relationen mittels Hashfunktionen partitionieren, sodass die Partitionen in den Hauptspeicher passen.
- Auf Partitionen Hauptspeicherindexstrukturen anwenden (Hashing)

N-Wege sortieren

1. Soviel wie möglich in den Hauptspeicher laden und sortieren > Run
2. Je zwei Runs zusammenmergen (müssen nicht komplett in Hauptspeicher)

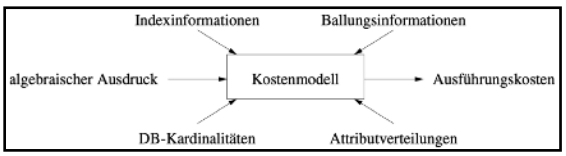
Replacement Selection

Ausgabe						Speicher				Eingabe						
					10	10	20	30	40	25	73	16	26	33	50	31
					10	20	25	30	40	73	16	26	33	50	31	
				10	20	25	30	40	73	16	26	33	50	31		
			10	20	25	(16)	30	40	73	26	33	50	31			
		10	20	25	30	(16)	(26)	40	73	33	50	31				
	10	20	25	30	40	(16)	(26)	(33)	73	50	31					
10	20	25	30	40	73	(16)	(26)	(33)	(50)	31						
					16	26	31	33	50							

Neue Eingabe im Hauptspeicher führt zur Verdrängung, Ausgabe aber in sortierter Reihenfolge.

7.4 – Kostenmodelle

Kostenmodell



Selektivitäten

Anteil der qualifizierenden Tupel einer Operation

$$\text{sel}_p := \frac{|\sigma_p(R)|}{|R|} \qquad \text{sel}_{RS} := \frac{|R \bowtie S|}{|R \times S|}$$

Abschätzungen der Selektivität

$\text{sel}_{R.a=c} = \frac{1}{|R|}$ falls a Schlüssel von R

$\text{sel}_{R.a=c} = \frac{1}{i}$ falls i die Anzahl der Attributwerte von $R.a$ ist

$\text{sel}_{R.a=S.b} = \frac{1}{|R|}$ bei Equijoin von R mit S über Fremdschlüssel in S

Joinreihenfolge

Joinreihenfolge bestimmen ist NP-vollständiges Probelm und hat großen Einfluss auf Performanz.

- Greedy-Heuristiken
  - > Beginne mit einer Relation
  - > Joine die günstigste dazu (kleinstes erwartetes Ergebnis)
  - > Wiederhole bis alle gejoint sind
- Dynamisches Programmiern
  - > löse Teilprobleme optimal ($|P_1| = 1, \ldots, |P_n| = n$)
  - > baue daraus Lösungen für komplexere Probleme

8: Transaktionsverwaltung

8.1 – Transaktionverwaltung

Transaktionsverwaltung

Transaktionsverwaltung besteht aus
- Recovery: Behebung von Fehlersituationen
- Synchronisation: sicherer Mehrbenutzerzugriff

Operationen

Beginn einer Transaktion

```
BEGIN TRANSACTION
```

Erfolgreiche Beendigung einer Transkation, persistente Speicherung

```
COMMIT
```

Selbstabbruch bzw. Zurücksetzen auf Zustand zur Beginn der Transaktion (immer erfolgreich)

```
ABORT
```

Sicherungspunkte definieren, zu dem sich Transaktion zurücksetzten lässt

```
DEFINE SAVEPOINT
```

Zurücksetzen der Transaktionen auf den letzten aktuellen Sicherungspunkt

```
BACKUP TRANSACTION
```

Systemabsturz

Im Falle eines Systemabsturzes müssen alle abgeschlossenen Transaktionen in der DB erhalten bleiben. Alle nicht abgeschlossenen Transaktionen müssen vollständig zurückgesetzt werden.

Transaktionen und SQL

Transaktion beginnen

```
START TRANSACTION
```

Beende Transaktion und schreibe Änderungen fest, kann z.B. bei Verletzung von Integritätsbedingungen fehlschlagen

```
COMMIT (WORK)
```

Beende Transaktion und setze alle Änderungen zurück, muss immer erfolgreich sein

```
ROLLBACK (WORK)
```

ACID-Eigenschaften

- Atomicity: ‚alles oder nichts' bei einer Transaktion
- Consictency: konsitente Datenbank bleibt auch nach Transaktion konsitent (z.B. referentielle Integrität)
- Isolation: keine Seiteneffekte, es fühlt sich an, als sei man der einzige Nutzer
- Durability: alle mit commit festgeschriebenen Änderungen müssen bestehen bleiben

9: Recovery

9.1 – Überblick

Recovery Mechanismen
- Sicherungspunkte: Backups des gesamten Datenbestands
- Log-Dateien: Transaktionsdaten, Protokollierung aller Änderungen

MEDIA Recovery
Vollständiger Verlust (Hintergrundspeicher): Backup mit Log in aktuellen Stand überführen.

Fehlerklassifikation
- R1 Recovery: Lokale Fehler, die noch nicht commitet sind
- R2 Recovery: Hauptspeicherverlust, abgeschlossende TAs erhalten
- R3 Recovery: Hauptspeicherverlust, nicht abgeschlossene TAs müssen zurückgesetzt werden
- R4 Recovery: Hintergrundspeicherverlust, MEDIA-Recovery

Ersetzungsstrategien
- ¬steal: so lange nicht commited ist, wird nicht auf Hintergrundspeicher geschrieben
- steal: jede nicht fixierte Seite kann verdrängt werden, erforderd kompliziertes R3

Speicherstrategien
- force: nach Commit Änderungen direkt auf den Hintergrundspeicher schreiben
- ¬force: geänderte Seiten bleiben bis Verdrängung im Hauptspeicher und werden erst dann im Hintergrundspeicher persistiert

Auswirkung auf Recovery

	force	¬force
¬steal	Kein Redo, kein Undo	Redo, kein Undo
steal	Kein Redo, Undo	Redo, Undo

Einbringungsstrategien
Einbrinungsstrategien definieren, wie Daten auf den Hintergrundspeicher geschrieben werden.
- Update in Place: Direktes Überschreiben der Seiten
- Twin-Block-Verfahren: Für jede Seite liegt eine Kopie vor, von der die Änderung beim commiten übertragen werden
- Schattenspeicherkonzept: Nur geänderte Seiten werden dupliziert, sonst wie Twin-Block

9.2 – ARIES

ARIES-Protokoll
Das ARIES-Protokoll ist ein weit verbreitetes Protokoll zur Fehlerbehandlung. Log-Datei enthält
- Reder-Informationen
- Undo-Informationen

WAL-Prinzip

Write Ahead Log

- Bevor eine Transaktion commited werden kann, müssen alle zugehörigen LOG-Einträge in die Log-Datei (für Hauptspeicher, schnell) ausgeschrieben werden.
- Bevor eine modifizierte Seite ausgelagert (verdrängt) werden darf, müssen alle Log-Einträge in das Log-Archiv (für immer, langsam) ausgeschrieben sein

M.a.W: Erst Loggen, dann Schreiben!

Phasen des Wiederanlaufs

Nach einem Absturz werden folgende Phasen zur Wiederherstellung der Daten durchlaufen.

1. Analyse: Ermittlung von Winnern (TAs fertig, müssen vollständig nachvollzogen werden) und Loosern (TAs nicht fertig, müssen rückgängig gemacht werden)
2. Wiederholung der Historie: Alle Log Einträge zu Winnern & Loosern werden ge-Redo't
3. Undo der Looser: Looser TAs werden rückgängig gemacht

Struktur der Log-Einträge

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

- LSN (Log Sequence Nummer): Kennung des Log-Eintrags, monoton steigend
- TA: Transaktionskennung der Transaktion
- PageID: eine Seitenkennung von einer veränderten Seite (Atomar, sonst mehrer Log-Einträge!)
- Redo: physisch after image, logisch before → after
- Undo: physisch before image, logisch after → before
- PrevLSN: Zeiger auf vorhergehenden Log-Eintrag (Effizienzgründe)

9.3 – Sicherungspunkte

Sicherungspunktarten

- Transaktionskonsistent: Während der Erstellung des Sicherungspunktes werden keine TAs ausgeführt.
- Aktionskonsistent: System wird kurz angehalten um aktuelle TA Zustände zu speichern
  > Analyse & Redo nur ab Sicherungspunkt
  > Undo potentiell bis zum Beginn aller TAs, die zurückgesetzt werden
- Unscharf (fuzzy): Sicherungspuntk wird im Regelbetrieb erstellt, parallel werden Kennungen modifizierter Seiten gespeichert, um Dirty Pages zu identifizieren
  > Analyse beginnt ab Sicherungspunkt
  > Redo muss alle Dirty Pages beachten, da hier eventuell noch TAs neu hinzugekommen sind
  > Undo potentiell bis zum Beginn aller TAs, die zurückgesetzt werden (s.o.)

10: Mehrbenutzersynchronisation

10.1 – Probleme

Lost Update
Ergebnis einer Transaktion geht verloren, weil eine parallele Transaktion das Ergebnis überschreibt.

Dirty Read
Lesen von Daten, die später über *abort* wieder zurückgesetzt werden.

Non-Repeatable Read
Die Veränderung in den Daten hat zur Folge, dass bei zwei Abfragen unterschiedliche Ergebnise gelesen werden.

Phantom Problem
Erweiterung des Datenbestands führt zur Veränderung bei weitere Anfrage, obwohl man selber nichts neues hinzugefügt hat (Isolation!).

10.2 – Isolation Levels

Isolation Levels

	Lost update	Dirty read	Non-repeatable Read	Phantom problem
Read uncommitted	✓			
Read commited	✓	✓		
Repeatable read	✓	✓	✓	
Serializable	✓	✓	✓	✓

Zugriffmodi
- Read only: ermöglicht parallelen Zugriff
- Read write: größerer synchronisations Overhead

10.3 – Historien

Transaktionen TA
- Elemente einer TA
  > $r_i(A)$: Lesen des Datenobjekts A; $w_i(A)$: Schreiben des Datenobjekts A
  > a_i: Abbruch; c_i: erfolgreicher Commit
- TA's sind partielle Ordnungen der Ordnungsrelation $<_i$
  > a_i oder c_i sind letzte Operation
  > Zwei Operationen auf demselben Datenobjekt sind immer geordnet
- Transaktionen können als gerichete azyklische Graphen (DAGs) dargestellt werden

Historien
Historien geben an, wie Operationen aus verschiedenen TAs relativ zueinander ausgeführt werden. Operationen können parallel ausgeführt werden > partielle Ordnung.
Es kann auch Abhängigkeiten zwischen teilen einzelner Transaktionen geben.

Konfliktoperationen
Zwei Operationen stehen im Konflikt miteinander, wenn sie auf dasselbe Datenobjekt arbeiten und mindestens eine der Operationene eine Schreiboperation ist.

Konfliktäquivalenz
Zwei Historien H und H' sind (konflikt-)äquivalent, wenn
- sie die gleichen TAs enthalten
- Konflikoperationen der erfolgreichen TA in derselben Reihenfolge sind

Serialisierbarkeit
- Serielle Historien: Alle TAs werden strikt nacheinander ausgeführt
- Serialisierbare Historien: Historien, die (konflikt-)äquivalent zu einer seriellen Historie sind
  > Serialisierbarkeitsgrapgh $SG(H)$ ist azyklisch; Knoten sind erfolgreich absgeschlossene TAs aus H, Kante zwischen $T_i \rightarrow T_j$ wird eingetragen falls es Konfliktoperationen mit $p_i <_H q_j$ gibt

Rücksetzbarkeit
Historien sind rücksetzbar, falls alle Transaktionen T_k, von denen T_i liest, vor T_i beendet sind.
- Kaskadierendes Rücksetzen: Eine Historie vermeidet kaskadierendes Rücksetzen, wenn für TAs gilt: Wenn T_i von T_j liest, dann gilt $c_j <_H r_i[x]$

Striktheit
Eine Historie ist strikt, wenn nur von commiteten TAs gelesen oder überschrieben wird.
> Erlaubt physikalische Protokollierung beim Recovery (‚before image von T: x')

10.4 – Datenbank Scheduler

Datenbank Scheduler
Der Datenbank Scheduler nimmt eingehende Transaktionen vom Transaktionsmanager entgegeen und ordnet sie so an, dass eine serialisierbare und rücksetzbare Historie entsteht.

Pessimistische und Optimistische Scheduler
- pessimistisch: verzögert, beste Reihenfolge festlegen (Bsp: Sperrbasiert)
- optimistisch: möglichst schnelle Ausführung, Schäden im nachhinein beheben

Sperren
Äquivalent zu synchronized, Sperren von einzelnen Datenobjekten (Tupeln).
- 2PL: R/W-Lock; 1. Sperren anfordern > 2. Sperren freigeben ~~> 3. Sperren anfordern~~
  > Strenges 2PL: alle Sperren bis zum Ende halten > ACA & Strikt
  > Preclaiming: Zur Vermeidung von Deadlocks erst alle Sperren anfordern
- MGL: Sperren auch auf Seiten oder ganze Relation (zur Verhinderung des Phantomproblems)
  > shared, für Leser; exclusive, für Schreiber; intention share/exclusive, für beabsichtigtes L/S

Deadlocks

TAs sollten nicht ewig auf eine Sperre warten.

- Time-Out: TAs nach zu langer Wartezeit abbrechen
- Wartegraphen: gerichtete Graphen mit den jeweiligen Warteabhängigkeiten > Zyklus = Deadlock.

Deadlock können durch Preclaiming (alle Sperren am Anfang anfordern) verhindert werden.
Priority Queues mit Zeitstempeln verhindern Livelocks.

_