Zusammenfassung für die mündliche Diplomprüfung im Fach Datenbanksysteme

Naja von Schmude

3. August 2009

Inhaltsverzeichnis

Das Dokument versucht die Foliensätze[1] zur Datenbankvorlesung im Sommersemester 2009 mit der 7. Auflage des Buchs „Datenbanksysteme - Eine Einführung“ von Kemper/ Eickler in Einklang zu bringen, natürlich nur in dem Maße, wie auch der Stoff in der Vorlesung behandelt wurde. Die Unterteilung in diesem Dokument stimmt mit den Kapiteln in den Foliensätzen überein.

1 Intro

- Ein DBS bietet Abstraktion von der physikalischen Repräsentation der Daten und Operationen (es ist also kein blödes IO Gedöns mehr notwendig). Zudem bietet es Sicherheit, nebenläufigen Zugriff und Fehlertoleranz.

- Datenunabhängigkeit wird durch Abstraktionslevel sichergestellt:

  physical data independence Modifikation der physischen Speicherstruktur belässt Datenbankschema invariant, und somit muss auch nicht das Anwendungsprogramm verändert werden

  logical data independence Veränderungen des Schemas sollen keine Auswirkungen auf das Programm haben. Natürlich geht das nur in kleinem Rahmen, bie großen Änderungen funktioniert das nicht. Ein wenig kann man durch Sichten ausgleichen

  Die Abstraktionslevel sind

  Physiche Ebene wie Daten auf dem Hintergrundspeicher abgelegt werden

  konzeptuelle (logische) Ebene durch Datenbankschema festgelegt, welche Daten gespeichert sind

  Sichten verschiedene Benutzergruppen/ Anwendungen bekommen nur bestimmte Teilmengen der Informationen bereitgestellt

- DBS Systeme basieren auf einem *Datenmodell*, das die Infrastruktur für die Modellierung der realen Welt zur Verfügung stellt. Das ist eine Sprache, die zwei Teilsprachen umfasst:

  Data Definition Language Zum Definieren des Schemas

  Data Manipulation Language Zum Abfragen und Ändern der Daten

[1] Die Foliensätze sind unter folgender URL aus dem VPN des Fachbereichs erreichbar: `http://www.inf.fu-berlin.de/lehre/SS09/DBS-Intro/material.html`

Man kann Datenmodelle auf den verschiedenen Abstraktionsschichten betrachten

konzeptuell Abgrenzung des zu modellierenden Ausschnitts der Welt. Mit ER-Modell, UML usw. (hier wird höchstens DDL verwendet)

logisch relationales DM, XML, Netzwerkmodell, hierarchisches Modell ...

physikalisch Deklarative Beschreibung des Implementierungsschemas

- Struktur der abspeicherbaren Datenobjekte werden festgelegt entsprechend des logischen Datenmodells (Metadaten). Diese Struktur nennt man das *Datenbankschema*.
- Der Satz von Datenbjekten, die einem Schema genügen, also Daten in DB zu einem bestimmten Zeitpunkt, werden als *Datenbankausprägung* (Zustand) bezeichnet.
- Das *DB(M)S* ist ein Softwaresystem zum Verwalten von Daten
- Ein *Computerized Information System* beschreibt die DB und alle Dinge drumherum; auch Geschäftsprozesse, Kommunikation usw.
- Technische Anforderungen an ein DBS:

  Konsistenz / Integrität zwischen DB und Realität muss immer bestehen

  Mehrbenutzerbetrieb Keine Interferenz von gleichzeitigen Operationen mehrere User

  Fail-Safe Systemfehler darf keinen inkonsistenten Zustand hinterlassen

  Effizienz

  Datensicherheit

2 Entwurf

- Der Prozess des Definierens einer Gesamtstruktur auf den unterschiedlichen Abstraktionsebenen wird als *Datenbank-Entwurf* bezeichnet.
- Entwurfschritte
  1. In der *Anforderungsanalyse* werden die notwendigen Informationen durch Interview, Diskussion gesammelt und anschließend nach relevanten Aussagen gefiltert, Redudanzen und Unstimmigkeiten ausgemerzt, Lücken gefüllt usw. Man versucht Objekte, Beziehungen und Attribute rauszufiltern.
  2. Beim *konzeptuellen Entwurf* wird die Realität auf Type Level beschrieben, meist mit ER-Modell oder UML. ES GIBT IMMER MEHRERE MÖGLICHKEITEN!!!!
  3. Beim *Entwurf des logischen Schemas* (Implementationsentwurf) wird der konzeptioneller Entwurf in ein Datanbankschema überführt

4. Der *physische Entwurf* dient der Effizienzsteigerung

- Eine *Integritätsbedingung* ist eine Invariante der Datenbankausprägung, die immer gelten muss. Es gibt z.B.

  Attributbedingungen

  Kardinalitäten (Funktionalitäten, Multiplizitäten in UML) Beschränkt wie viele Entitätinstanzen (Zeilen) von E_1 in Beziehung mit Entitätsinstanzen (Zeilen) von E_2 stehen dürfen. Es gibt $1:N$, $1:1$ und $N:M$, oder $(\min, \max)$.

- Die *schwache Entität* ist existentiell abhängig von der normalen Entität. Bei schwachen Entitäten ist die Kardinalität der starken Entität IMMER 1

- Bei der *Generalisierung* werden Eigenschaften ähnlicher Entititypen herausfaktorisiert und einem gemeinsamem Obertyp zugeordnet. Man unterscheidet

  diskunkte Generalisierung , bei der die Unterklassen paarweise diskunkt sind und

  vollständige Generalisierung , wo die Oberklasse die Vereinigung aller Unterklassen ist und jedes Datum gehört nur zu einer Unterklasse.

- Bei der *n-ären Beziehung* sind n Entitäten in die Beziehung involviert. Man kann sie darstellen, in dem man eine neue schwache Entität für die Beziehung einführt und zu dieser n binäre Beziehungen einfügt. Semantisch allerdings etwas unterschiedlich.

- Bei der *Aggregation* formen verschiedene Entitäten in ihrere Gesamtheit ein neue

- Bei der *Sichtenintegration* werden mehrere konzeptuelle Modelle, die zusammengehörig sind, so vereinigt, dass sie ein konsistentes Gesamtmodell bilden

3 Relationales Datenmodell

- Die Eigenschaften des relationalen Modells sind folgende: Keine Duplikate, da Relation eine Menge ist, keine Reihenfolge unter den Tupeln, Attribute haben primitive Typen, atomare Werte, Integritätsbedingungen müssen immer gelten, Attribute mit eindeutigen Namen

- Ein *Schlüssel* ist eine minimale Teilmenge der Attribute, die jedes Tupel eindeutig identifizieren. Minimal in dem Sinne, dass man nicht ein Attribut aus der Teilmenge entnehmen kann, so dass es noch ein Schlüssel bleibt. Es kann mehrere potentielle Schlüssel geben (Kandidatenschlüssel), von denen man dann einen zum Primärschlüssel macht. Oft fügt man künstliche Schlüssel ein (Personalnummer etc.)

- Ein *Fremdschlüssel* bezeichnet ein oder mehrere Attribute in der Relation S, wenn die Attribute dieselben Typen haben wie die Attribute des Schlüssels in Relation R (muss nicht Primärschlüssel sein). Zudem muss der Wert des Fremdschlüssels in R auftauchen oder `NULL` sein

- Die *referentielle Integrität* ist zugesichert, wenn alle Fremdschlüsselbedingungen eingehalten werden
- Bei der Umsetzung vom konzeptuellen Modell in relationales Modell (Schema) benutzt man folgende Regeln:
  - bei Entitäten: Attribute werden Attribute im RDM, Schlüssel wird Primärschlüssel
  - bei schwache Entitäten: Primärschlüssel der starkten Entität wird Teil des Schlüssel in der schwachen Entität
  - bei Beziehungen: Neue Relation mit den Schlüsseln der beteiligten Relationen und Attributen der Beziehung. Schlüssel setzt sich entsprechend der Kardinalität aus den Fremdschlüsseln zusammen ($1:1$ einer der Fremdschlüssel kann ausgewählt werden, $1:N$ Schlüssel der N-seitigen Entität, $N:M$ Schlüssel sind alle Schlüssel der N und M Relation)
  - bei Generalisierung hat man drei Möglichkeiten:
    1. Alles in eine große Relation rein packen, daraus folgen vermutlich viele NULL Werte
    2. Für Obertyp und Untertypen eigene Relationen. Schlüssel des Obertyps ist Teil des Schlüssels in den Untertypen
    3. Jeden Untertyp um die Attribute des Obertyps erweitern
- Hat man das gemacht, kann man das relationale Schema vereinfachen, in dem man Relationen mit gleichem Schlüssel zu einer Relation zusammenfasst. Es gelten folgende Regeln:
  - $1:N$ eindeutig, aber optional. Kann zu vielen NULL Werten führen, abwägen
  - $1:1$ Beziehungsrelation nach belieben mit einer der Entitätrelation vereinigen
  - $N:M$ Niemals vereinigen!
- Bedingungen sind
  - Wertbedingungen auf Attributen
  - Kardinalitäten
  - Semantische Bedingungen
  - referentielle Bedingungen durch `ON DELETE`, `ON UPDATE` bewahren
  - in SQL-DDL: Column constraint (nur auf eine Spalte bezogen) und table constraint (auf mehrere Spalten bezogen)

    PRIMARY KEY Nur einmal in einer Tabelle!

    NOT NULL Spalte darf nicht NULL sein, muss also immer mit einem Wert belegt sein

Default Values `<attName> <attType> DEFAULT <value>`

UNIQUE Wert nur einmal pro Tabelle (wie Primary Key)

CHECK

- Das *Henne-Ei-Problem*, d.h. dass referentielle Abhängigkeiten so gesetzt sind, dass sie einen Kreis bilden, kann wie folgt gelöst werden, in dem das Constraint-Checking ans Ende der Transaktion verschoben wird: `[DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIALLY IMMEDIATE]` bei `CONSTRAINT` Angabe
- Eine *Zusicherung* ist eine Integritätszsbedingung, die unabhängig von der Tabellendefinition ist (vergleichbar mit `CHECK`)
- *Trigger*: Wenn das Prädikat wahr ist, dann wird die zugeordnete Aktion ausgeführt. Man kann auch Aktionen ausserhalb der DB auslösen (z.B. Emailversand ...). Werden ausgelöst durch Datenbankevents: `INSERT`, `UPDATE` oder `DELETE`, Veränderung vom Schema, User Logon, Fehler ...
- SQL Konzepte

  Namespace Catalogue, Database, Schema, Table, Column

  Datentypen `NUMERIC`, `INTEGER`, `DECIMAL`, `SMALLINT`, `REAL`, `DOUBLE PRECISION`, `CHAR`, `VARCHAR` ...

  Generated Column Werte werden automatisch berechnet und zugewiesen, und zwar jedesmal, wenn eine Zeile eingefügt wird

  Domain Benannte Menge von Werten und Wertrepresentation. Ähnlich wie `TYPE` Definition

  Metadaten-Management Alle Definitionen und andere Daten über Daten werden Metadaten genannt. Die Datenstruktur für Metadaten wird im Datenwörterbuch abgespeichert, meist auch als Tabellen. So können Metadaten genauso behandelt werden wie normale Daten!

  Views Eine virtuelle Tabelle, die eine Definition hat, aber keinen Inhalt. Der View wird fest in das Schema integriert. Die Definition wird ausgeführt, wenn auf die Tabelle zugegriffen wird

  `CREATE VIEW <name> AS <SQL SELECT>`

4 Funktionale Abhängigkeiten und Normalformen

- Man braucht irgendeinen formalen Maßstab, um die Güte eine relationalen Schemas festzustellen, also ob es gut ist. Informale Richtlinien währen
  - Redundanzen vermeiden
  - Nicht mehr als ein reales Objekt in eine Entität / Relation packen

  Bei Nichteinhaltung können so genannte *Anomalien* auftreten:

Updateanomalie Bei Redundanzen muss man bei einem Update alle Daten ändern, die redundant sind.

Löschanomalie Beim Löschen von Tupeln aus Entitäten, die eigentlich aus zwei Entitäten bestehen, kann es dazu kommen, dass versehentlich Informationen einer Entität verloren gehen.

Einfügeanomalie Beim Einfügen in eine Relation, bei der man Informationen zweier Entitäten zusammengemischt hat, kann es vorkommen, dass man nur Informationen für einen der beiden Typen hat und so die übrigen Werte mit `NULL` belegt werden.

- *Funktionale Abhängigkeiten* werden benutzt um formal Integritätsbedingungen von Attributen und Beziehungen auszudrücken. Sie verallgemeinern quasi das Schlüsselkonzept. Sie stellen also semantische Konsistenzbedingungen dar, die aus der Anwendungssemantik hervorgehen.

- $\alpha \to \beta$ bedeutet, dass β funktional abhängig von α ist. Dies sagt aus, dass für zwei Tupel r, s einer Relation die Werte der Attributmengen α und β sich wie folgt verhalten müssen: Wenn $r.\alpha = s.\alpha$, dann folgt $r.\beta = s.\beta$. Folgende triviale Abhängigkeiten gelten (*Armstrong-Axiome*):

  Reflexivität $\beta \subseteq \alpha$, dann gilt $\alpha \to \beta$

  Verstärkung Falls $\alpha \to \beta$ gilt, dann gilt auch $\alpha\gamma \to \beta\gamma$

  Transitivität Falls $\alpha \to \beta$ und $\beta \to \gamma$ gilt, dann gilt auch $\alpha \to \gamma$

- Die Armstrong-Axiome sind *vollständig* und *korrekt* (sound). Korrekt sagt aus, dass ich nur diese FDs aus einer Menge F von FDs ableiten lassen, die von jeder Relationsausprägung erfüllt sind, für die auch F erfüllt sind. Vollständig besagt, dass sich auch alle FDs ableiten lassen, die logisch von F impliziert werden.

- Ein Attribut ist *prim*, wenn es Teil eines Schlüssels (auch Kandidatenschlüssels) ist. Ansonsten wird das Attribut als *nicht prim* bezeichnet

- Man kann folgende Abhängigkeiten unterscheiden...

  Schlüsselabhängigkeit

  Partielle Abhängigkeit Ein Attribut ist von einem Teil des (Kandidaten)Schlüssel funktional abhängig

  Abhängigkeiten zwischen nicht-schlüssel Attributen

  Abhängigkeiten zwischen Kandidatenschlüsseln

- *Normalisierung* soll Relationen so aufspalten, dass die oben aufgeführten Anomalien vermieden werden. Dabei müssen folgende Korrektheitskriterien beachtet werden:

  Verlustlosigkeit Die ursprüngliche Relationsausprägung muss aus den aufgespaltenen Schemata wieder herstellbar sein (über Joins). Formal muss entweder $R_1 \cap R_2 \to R_1$ oder $R_1 \cap R_2 \to R_2$ gelten. Dies besagt, dass die gemeinsamen Attribute von R_1 und R_2 ein Schlüssel oder Superschlüssel sein müssen

Abhängigkeitserhaltung Die funktionalen Abhängigkeiten des Ursprungsschemas muss auf die aufgesplitteten Schemen übertragbar sein. D.h. jede Funktionale Abhängigkeit ist einer Aufsplittung zuzuordnen

- Bei einer *mehrwertigen Abhängigkeit* (MVD) $\alpha \twoheadrightarrow \beta$ müssen zu zwei Tupel a und b, die die selben α Werte haben, alle Tupel existieren, bei denen man beliebig die β Werte vertauschen kann.
- Eine Relation ist in *1. Normalform*, wenn alle Attribute atomar sind
- Eine Relation ist in *2. Normalform*, wenn kein nicht-prim Attribut funktional von nur einem Teil des Schlüssels abhängig ist. Anders gesagt: Jedes nicht-prim Attribut ist funktional abhängig von jedem Kandidatenschlüssel
- Eine Relation ist in *3. Normalform*, wenn nicht-prim Attribute nicht transitiv vom Schlüssel abhängen. Oder keine funktionalen Abhängigkeiten zwischen nicht-prim Attributen existieren. Wenn eine Relation in 3. NF ist, dann ist sie automatisch auch in 2. NF.
- Eine Relation ist in *BCNF*, wenn aus einer nicht-trivialen Abhängigkeit $X \rightarrow a$ folgt, dass X ein Superschlüssel ist. BCNF garantiert zwar die Verlustlosigkeit, aber nicht unbedingt die Abhängigkeitserhaltung.
- Eine Relation ist in *4. Normalform*, wenn für jede MVD $\alpha \twoheadrightarrow \beta$ gilt, dass $\beta \subseteq \alpha$ oder $\beta = R \backslash \alpha$ oder α ein Superschlüssel ist.
- Eine Relation, die in 3. NF ist und bei dem alle Kandidatenschlüssel aus nur einem Attribut bestehen, sind automatisch in BCNF.
- Zwei Ansätze, wie man zu Normalformen kommen kann

  Dekomposition Für eine gegebene Menge an Relationen zerlege die Relationen, die nicht normalisiert sind, zu normalisierten Relationen.

  Synthese Versucht eine minimale kanonische Überdeckung der funktionalen Abhängigkeiten zu finden und daraus die normalisierten Relationen zu bilden

5 Relationale Algebra

- Prozedural orientierte Sprache, die wohldefiniert ist. Zudem deklarativ.
- Relationale Algebra ist abgeschlossen, d.h. Ergebnisse der Anfragen sind wieder Relationen
- R und S können *vereinigt* werden ($R \cup S$), wenn die Schema die selben sind. Da es sich um Mengen handelt, gibt es keine Duplikate.
- Die *Differenz* kann wie die Vereinigung gebildet werden. $R \backslash S$ ist die Menge der Tupel, die in R aber nicht in S vorkommen

- Das *Kreuzprodukt* $R \times S$ liefert die Menge aller möglichen Paarekombinationen von R und S. Bei gleichbenannten Attributen wird Umbenennung erzwungen durch das Voranstellen der Relationsnamen.
- Attribute und Relationen können *umbenannt* werden. Für Attribute:

  $$\rho_{<attributname> \leftarrow <neuerAttributname>}(< Relation >)$$

  und für Relationen $\rho_{<neuerRelationsname>}(< Relation >)$
- Die *Projektion* $\pi_B(R)$ projiziert alle Tupel von R nur auf die durch B ausgewählte Attributmenge. Wieder ohne Duplikate, da es eine Menge ist! Es gilt $|R| \geq |\pi_B(R)|$. Wenn B ein Schlüssel enthält, dann gilt die Gleichheit.
- Die *Selektion* $\sigma_P(R)$ selektiert alle Zeilen aus R, die das Prädikat P erfüllen
- Joins

  Natural Join $R \bowtie S$ ist ein Equijoin über alle gleichbenannten Spalten gleichen Typs von R und S Zudem werden doppelte Spalten rausgefiltert.

  Theta-Join Join über zwei Relationen R und S unter Angabe eines Join-Prädikates Θ, dass sich auf die Attribute bezieht. Man schreibt $R \bowtie_\Theta S$.

  Equijoin Prädikat bezieht sich auf die Gleichheit vom Spaltenwerten

  Left / Right / Full Outer Join $]\ \bowtie$, $\bowtie\ [$, $]\ \bowtie\ [$ Tupel der jeweiligen Argumentrelation bleiben erhalten (falls es keinen Gegenpart gibt, wird dort NULL eingetragen)

  Left / Right Semi Join $\ltimes$, $\rtimes$ Join wird auf die entsprechende Seite projeziert
- Die *Division* $R \div S$ enthält nur die Attribute der Argumentreltation R, die nicht in S enthalten sind. Zudem werden nur die Tupel aus R genommen, die für alle Einträge der Attribute aus S einen Gegenpart haben und bei denen jeweils die Attribute von $R \backslash S$ gleich bleiben.
- Ein *Ausdruck* der Relationalen Algebra wird aus mehreren Algebraausdrücken zusammengesetzt. Zulässich ist die Vereinigung, Differenz, Kreuzprodukt, Selektion, Projektion und Umbenennung.
- Die *Basis* der relationalen Algebra besteht aus Projektion, Selektion, Kreuzprokukt, Differenz, Vereinigung und Umbenennung.
- Unterschied Relationale Algebra und SQL: RA Ergebnisse sind Mengen, also keine Duplikate, in SQL Ergebnissen können Duplikate vorkommen. Duplikate können per `DISTINCT` eliminiert werden
- Eine DB-Sprache wird *relational vollständig* genannt, wenn jeder Ausdruck der relationalen Algebra in ihr ausgedrückt werden kann. Die transitive Hülle kann nicht mit einer endlichen Anzahl von RA Ausdrücken ausgedrückt werden, deshalb ist keine Rekursion möglich

- Man kann einen RA Ausdruck mit einem Operatorbaum darstellen
- Der systematische Austausch von Operationen entsprechend der Gesetzmäßigkeiten der RA wird als *algebraische Optimierung* bezeichnet

6 Tupelkalkül

Nicht prüfungsrelevant laut Herrn Schweppe, es sei denn, man hat besonderes Interesse daran!

7 SQL

- *Löschen* von Tupeln mit `DELETE FROM <table> WHERE <predicate>;`. Löschen von Tabellen mit `DROP TABLE <table>;`
- *Ändern* von Tupeln mit

  ```
  UPDATE <table> SET <attribute> = <value> [,<attribute> = <value>]* WHERE <predicate>;
  ```

  Ändern von Tabellen mit `ALTER TABLE ...`
- *Einfügen* von neuen Datensätzen mit

  ```
  INSERT INTO <table> (<attribute>, <attribute> ...) VALUES (<value>, <value> ...);
  ```

  Man kann auch Daten, die man mithilfe eines `SELECT` Befehls erhält, in eine Tabelle einfügen: `INSERT INTO <tabel> (SELECT ...);`. Natürlich müssen hierbei die Attribute übereinstimmen. Einfügen von Tabellen über `CREATE TABLE <name> (...);`
- SQL ist relational abgeschlossen, enthält aber noch zusätzliche Konzepte wie *Grouping* und *arithmetische Operationen*
- Einfache Suchprädikate sind z.B. `<attribute> BETWEEN <value1> AND <value2>` oder `<attribute> IS (NOT) NULL`, oder `<attribute> IN <Menge>`. Für Strings kann man auch mit `LIKE` suchen. Es gibt auch viele verschiedene Funktionen wie `SOUNDEX`, die einem das Leben erleichtern.
- Joins kann man entweder über das Kreuzprodukt mit anschließender Selektion (`SELECT * FROM <table1>, <table2> WHERE <predicate>;`) oder direkt mit einem Join-Befehl machen (`SELECT * FROM <table1> JOIN <table2> ON <predicate>`. Natürlich gibt's auch `(LEFT, RIGHT, FULL) OUTER JOIN`, und `NATURAL JOIN`.
- Man kann SQL-Anfragen *verschachteln*. Dabei unterscheidet man

  korrelierte Anfragen , bei denen sich die Unterabfrage auf die äußere Anfrage über Attributwerte bezieht oder

  unkorrelierte Anfragen , wo die Anfragen nichts miteinader zu tun haben.

Unkorrelierte Anfragen brauchen auch nur einmal ausgewertet werden, wohingegen man die korrelierten Anfragen für jedes Tupel der äußeren Anfrage neu berechnen muss. Jede Unterabfrage muss in Klammern eingeschlossen werden.

- `EXISTS` liefert `TRUE`, falls die Unterabfrage mind. ein Element enthält, ansonsten `FALSE`.
- *Grouping* fasst über den Befehl `GROUP BY <attribute>, <attribute> ...` Tupel mit den gleichen Werten der entsprechenden Attribute zu einem Tupel in der Ausgaberelation zusammen. Dabei kann noch mit `HAVING <predicate>` an das `GROUP BY` eine Bedingung an die Gruppierung gestellt werden. Gruppierung ist besonders nützlich im Zusammenhang mit *Aggregatsfunktionen*, die Operationen auf Tupelmengen durchführen und die Menge von Werten auf einen einzelnen Wert reduzieren. Funktionen sind z.B. `SUM`, `AVG`, `MIN`, `MAX`, `COUNT`. Wichtig: Da eine Gruppe in der Ausgaberelation nur mit einem Tupel repräsentiert wird, können in der `SELECT`-Klausel nur Aggregatsfunktionen vorkommen oder Attribute, die auch in der `GROUP BY`-Anweisung stehen.
- Eine *quantifizierende Anfrage* ist eine Anfrage, die aus Vergleichsoperatoren verknüpft mit `ANY` oder `ALL` bestehen. `ANY` testet dabei, ob es mind. ein Element in der Unterabfrage gibt, dessen Vergleich mit dem linken Argument des Vergleichs erfüllt wird, und `ALL` testet, ob der Vergleich für alle Elemente der Unterabfrage erfüllt ist. `IN` ist demnach äquivalent zur Anfrage `= ANY`. `ALL` nicht mit dem Allquantor verwechseln, den gibt es in SQL nicht!! Kann natürlich mit Hilfe des Existensquantors `EXISTS` ausgedrückt werden ...
- *Rekursionen* sind schwierig in SQL, da man die *transitive Hülle* nicht berechnen kann. Demnach ist SQL nicht turingvollständig. Oracle bietet allerdings Unterstützung über `CONNECT BY` und `START WITH`.
- Anfragen kann man ein wenig über *temporäre Tabellen* oder `WITH <name> AS (SELECT ...)` strukturieren.

8 Weitere SQL Features: Views, Trigger, Functions ...

- Eine *Sicht* ist ein benannter SQL-Ausdruck, der als virtuelle Tabelle Teil des Schemas wird. Virtuell bedeutet hierbei, dass es sich um keine echte Tabelle handelt, sondern der Ausdruck bei jedem Zugriff auf die Sicht neu ausgewertet wird. Eine Sicht kann sich auf eine Tabelle oder auf andere Sichten beziehen. Eine Sicht wird definiert über

  ```
  CREATE VIEW <name>(<attribute1>, <attribute2> ...) AS (SELECT ...);
  ```

- Sichten sollen allgemein den Zugriff für bestimmte Gruppen erleichtern und beschränken.
- Eine Sicht ist update-fähig unter folgenden Bedingungen:

  - Sicht enthält keine Aggregatfunktionen, oder `DISTINCT`, `GROUP BY`
  - Im `SELECT` der Sicht stehen nur eindeutige Spaltennamen und ein Schlüssel der Basisrelation
  - Sicht verwendet nur eine Tabelle (Sicht oder Basisrelation), die auch veränderbar ist

- *Methoden* und *Funktionen* erlauben komplexere Berechnungen direkt in SQL (PL/SQL).

- *Trigger* machen Datenbanksysteme proaktiv, in dem sie auf Datenbankveränderungen reagieren können. Mit Hilfe von Triggern lassen sich sehr komplexe Integritätsbedingungen ausdrücken.

```
CREATE TRIGGER <name> BEFORE / AFTER UPDATE / INSERT / DELETE
ON TABLE <table> FOR EACH ROW (<command>)
```

- In neueren SQL Versionen kann man *komplexe Datentypen* bauen, *Arrays* benutzen etc.

9 Einbettung von SQL in Programmiersprachen

- Um GUIs oder Turingvollständigkeit bereitzustellen, ist es notwendig SQL in eine Wirtsprache einzubetten. Dabei müssen folgende Probleme bewältigt werden:
  - Impedance Mismatch: Keine Mengenverarbeitung in normalen Programmiersprachen möglich, hier werden Daten iterativ verarbeitet. Hier führt man dann das *Curser-Konzept* ein, in dem der Curser immer auf ein Tupel der Ergebnisrelation zeigt, das dann bearbeitet werden kann. Der Curser wird dann iterativ eine Position weitergesetzt.

- *Embedded SQL* wird SQL-Code in ein den normalen Programmcode eingebettet. Über Spezielle Präfixe wird dieses markiert, so dass der Präcompiler diese Blöcke entsprechend umwandeln kann. Beispiel ist SQLJ oder ESQL. SQLJ arbeitet dabei über den JDBC-Treiber.

- ODBC und im speziellen JDBC bieten eine API für den Datenbankzugriff in einer bestimmten Programmiersprache. JDBC ist dabei unabhängig vom Hersteller des DBS. Man muss Verbindung aufbauen, SQL-Statement bauen, Query ausführen usw. Mit `PreparedStatemet` können die selben Anfragen mehrfach ausgeführt werden (mit z.B. unterschiedlichen Parametern).

- *OR-Mapping* versucht Details des DB-Schemas vor den Anwendungsprogrammierern zu verbergen, in dem man Relationen auf Objekte „mappt“. Das macht z.B. Hilbernate über ein XML-Dokument.

10 Data Warehouse

- Als *OLTP* (Online Transaction Processing) bezeichnet man DB-Anwendungen, die das „operative Tagesgeschäft“ eines Unternehmens abwickelt. Dies umfasst eigentlich die ganz normalen DB-Anwendungen.
- Als *OLAP* (Online Analytical Processing) bezeichnet man hingegen Anwendungen, die entscheidungsunterstützend (analytisch) arbeiten, in dem sie auf der ganzen Datenhistorie arbeiten um Rückschlüsse zu ziehen.
- Analytische Anfragen auf einem OLTP-Schema auszuführen ist ziemlich kompliziert (viele `JOIN`s, `UNION`s usw.), daher muss man es in ein für diese Aufgaben adäquates Schema überführen. Diese neue Datenbank, in der alle Daten in geeigneter Form eingefügt werden, nennt man *Data Warehouse*. Der Vorgang, um von Datenbanken (OTLP) zu einem Data Warehouse zu kommen, wird als *ETL* (Extract-Transform-Load) bezeichnet.
- Ein Data Warehouse soll die *retrospektive* Analyse von Daten ermöglichen.
- In einem DWH werden die Daten entsprechend unterschiedlicher Dimensionen aggregiert. Die verschiedenen Dimensionen kann man in einem *Hyperwürfel* anordnen, in dem man sich dann verschiedene Ausschnitte ansehen kann („slice and dice“).
- Als Schema des DWH hat sich das *Sternschema* durchgesetzt, das sich aus einer normalisierten *Faktentabelle* und vielen meist nicht normalisierten *Dimensionstabellen* zusammensetzt. Dabei sind die Attribute der Faktentabelle jeweils Fremdschlüssel auf die Dimensionstabellen. Wenn man die Dimensionstabellen normalisiert, ergibt sich ein *Schneeflockenschema*.
- Das Sternschema macht in der Abfrage *Sternjoins* notwendig, bei denen Gruppierungen und Aggregationen eingesetzt werden. Der Verdichtungsgrad wird dabei durch die Attribute, die in der `GROUP BY`-Klausel stecken, angegeben. Je mehr Attribute dort angegeben werden, umso geringer ist die Verdichtung, man spricht von *drill-down*. Je weniger Attribute man angibt, umso verdichteter sind die Ergebnisse, man sagt dann *roll-up*.
- Der `GROUP BY ROLLUP (<attribute1>, <attribute2> ... <attributen-1>, <attributen>)`-Operator in SQL ermöglicht es, eine ganze Reihe von Aggregationen auf einmal zu erzeugen, nämlich werden alle Ergebnisse folgender Gruppierungen vereinigt: `GROUP BY <attribute1> ... <attributen>`, `GROUP BY <attribute1> ... <attributen-1>`, ..., `GROUP BY <attribute1>`, `GROUP BY -`
- Der `GROUP BY CUBE (<attribute1>, <attribute2 ...>)`-Operator vereinigt ähnlich wie `ROLLUP` alle möglichen Kombinationen der Attribute als Gruppierung.
- Beim Aggregieren kann man temporäre Ergebnisse wieder verwenden.

- Zwei unterschiedliche Architekturen für DWH-Systeme:

  ROLAP Relationales-OLAP System, d.h. DWH wird auf Basis des relationalen Datenmodells realisiert

  MOLAP multi-dimensionales OLAP-System, bei dem die Daten in speziellen mehrdimensionalen Datenstrukturen (z.B. Arrays) abgespeichert werden

11 Data Mining

- Ziel des Data Mining ist es, Datenmengen nach bisher unbekannten Zusammenhängen und Mustern zu durchsuchen, so dass man Vorhersagen über das zukünftige Verhalten anhand der bekannten Daten treffen kann.

- Der Data Mining Prozess teilt sich in folgende Schritte auf:

  1. Datenerhebung
  2. Datenreinigung, d.h. Außenseiter entfernen, Korrektheit überprüfen, Redundanzen beseitigen ...
  3. Modell bauen (z.B. Entscheidungsbaum, Cluster ...)
  4. Modell auf neue Daten anwenden

- Ein *Entscheidungsbaum* ist ein Baum, der es erlaubt Vorhersagen für ein Datum aufgrund von vorgegebenen Attributen zu treffen. Beim Konstruieren eines Entscheidungsbaumes betrachtet man die Entropie, die Aufschluss über den Informationsgehalt eines Attributs gibt. Je höher für ein Attribut der Informationsgehalt ist (d.h. niedrigere Entropie), um so höher wird dieses als Knoten im Baum abgelegt.

- Mit *Assoziationsregeln* versucht man Zusammenhänge im Verhalten bestimmter Objekte durch Implikationsregeln auszudrücken. Zwei Kenngrößen geben die Qualität der Regeln an:

  Confidence Legt fest, bei welchem Prozentsatz der Datenmenge, der die Voraussetzung erfüllt, auch die Regel erfüllt ist.

  Support Legt fest, wie viele Datensätze die Voraussetzung und die Regel erfüllen müssen.

- Der *À Priori-Algorithmus* findet solche Assoziationsregeln auf Basis einer Warenkorb-Analyse. Dabei findet der Algorithmus *frequent itemsets*, also Mengen von Produkten, die häufig im selben Einkauf erstanden wurden.

- Hat man nun alle frequent itemsets gefunden, findet man die Assoziationsregeln so, in dem man alle möglichen disjunkten Zerlegungen des itemsets F finden. Dabei gilt, dass $confidence(L \Rightarrow R) = \frac{support(F)}{support(L)}$. Dabei kann man die Confidence erhöhen, in dem man Werte, die rechts von der Implikation stehen, auf die linke Seite verschiebt.

- Beim *Clustering* geht es darum logisch verwandte Objekte in Gruppen („Cluster“) einzuteilen. Dazu beschreibt man jedes Datenobjekt mehrdimensional so, dass man sich ein Datum als Punkt im Raum vorstellen kann. Dann werden die Objekte anhand ihres Abstands gruppiert.

12 Information Retrieval

- Beim *Information Retrieval* will man aus einer Menge von Text-Dokumenten, diejenigen Dokumente finden, die einer Anfrage nach einem bestimmten Wort genügen.
- Bei den ganzen unterschiedlichen Dokumentendarstellungen muss man sich eine *kanonische* Darstellung überlegen.
  - Man könnte einen riesigen Vektor K aller in der Datenbank vorkommenden Wörter anlegen (natürlich noch entsprechend normalisiert und aufbereitet) und für jedes Dokument dann einen eben solangen Vektor anlegen, der an der Stelle des Worts in K eine 0 oder 1 zu stehen hat, je nach dem ob das Wort in dem Dokument vorhanden ist.
  - Eine weiter Möglichkeit wäre eine *Posting List*, in der jedes Wort, dass in der Datenbank auftaucht, mit einer verketteten Liste von den Dokumenten verknüpft wird, in denen das Wort auftaucht.
- Bei einer Anfrage muss man irgendein Ähnlichkeitsmaß definieren, so dass man die gefundenen Dokumente irgendwie in eine Ordnung bringen kann. Dazu benutzt man meistens irgendeine Art von Metrik.
- Beim *Boolean Retrieval Model* werden die Wörter der Suchanfrage mit den booleschen Operatoren $\vee, \wedge, \neg$ verknüpft. Dabei ist eine Anfrage erfolgreich, wenn die entsprechenden Wörter in der Anfrage vorkommen (oder auch nicht, halt entsprechend der booleschen Operation)
- Beim *Vector Space Model* sind die Dokumente durch Punkte im $|K|$-dimensionalen Raum repräsentiert. Eine Anfrage entspricht dabei dann auch einem solchen Dokument, das als Punkt repräsentiert wird. Als Maß kann man hier das Kosinusmaß nehmen, wo auch noch Gewichte mit eingeflochten werden. Dabei sollten die Gewichte so gesetzt werden, dass häufig auftauchende Begriffe geringer gewichtet werden, da sie nicht so informativ sind (Stichwort „Document frequency“). Zum anderen charakterisiert ein Term, der öfters in einem Dokument vorkommt als ein andere, das Dokument besser als der andere (Stichwort „Term frequency“).
- Bei *Top-k* will man nur die besten mit der Anfrage matchenden Dokumente ranken und nicht alle. Dazu kann man einen Heap aufbauen und nur die wichtigstens k Ergebnisse zurückliefern.
- Mit dem *Pagerank* wird im Internet die Wichtigkeit einer Webseite angegeben, auf Basis von Verlinkungsstrukturen.

- Man kann Datenbanksysteme für IR erweitern, z.B. mittels *Text extenders*.

13 Transaktionen

- Eine *Transaktion* ist eine Bündelung mehrerer Datenbankoperationen zu einer Arbeitseinheit. An eine Transaktion gibt es zwei grundlegende Anforderungen

  Recovery Behandlung von Fehlern

  Synchronisation von mehreren (parallel) ablaufenden Transaktionen

- Eine Transaktion überführt eine Datenbasis von einem konsistenten Zustand in einen (eventuell) neue konsistenten Zustand. Dabei muss innerhalb der Transaktion allerdings nicht unbedingt die Konsistenz gewahrt sein.

- Für eine Transaktion sind nur folgende Operationen notwendig:

  read Lesen eines Datensatzes

  write Schreiben eines Datensatzes (kann natürlich auch löschen oder ändern sein)

  BOT Anfang der Transaktion wird markiert

  commit Erfolgreiche Beendigung der Transaktion

  abort Selbstabbruch der Transaktion

  rollback work Zurücksetzen (*Rollback*) der Transaktion in den Zustand vor dem Transaktionsbeginn

  Eine Transaktion ist dann eine Abfolge dieser Operationen.

- Das *ACID-Paradigma* gibt an, welche Eigenschaften eine Transaktion erfüllen muss.

  Atomicity Eine Transaktion soll atomar behandelt werden, d.h. sie wird entweder vollkommen ausgeführt, oder überhaupt gar nicht. („Alles oder nichts“)

  Consistency Eine Transaktion hinterlässt stets einen konsistenten Zustand.

  Isolation Für eine Transaktion sieht es stets so aus, als würde sie alleine auf der Datenbank arbeiten. Parallelität ist versteckt und nebenläufige Transaktionen dürfen sich nicht beeinflussen.

  Durability Das Ergebnis einer erfolgreichen Transaktion bleibt dauerhaft in der Datenbasis erhalten, auch nach einem Systemfehler.

  Die *Mehrbenutzersynchronisation* muss die Isolation von parallel ablaufenden Transaktionen gewährleisten, wohingegen die *Recovery* die Atomarität und Dauerhaftigkeit von Transaktionen sicherstellen muss.

- Schlimmste Fall bei der Verletzung der Isolation ist das *Lost update*, bei dem zwei unabhängige Transaktionen auf das selbe Objekt schreiben, und ein falscher Wert resultiert (da eine Schreiboperation von der anderen einfach überschrieben wurde).

- Es gibt mehrere Isolationslevel, die man einstellen kann, die unterschiedliche Konfliktarten zulassen. Es gilt, je mehr Isolation man will, um so weniger Parallelität bekommt man.

  READ UNCOMMITED Updates werden für lesende Transaktionen sofort sichtbar. Allerdings lässt man hier dirty reads zu und erhält Zugriff auf eventuell inkonsistente Daten

  READ COMMITED Es werden nur Daten gelesen, die in der Datenbasis festgeschrieben wurden. Allerdings kann es vorkommen, dass man bei zwei Leseoperationen auf dem gleichen Objekt unterschiedliche Werte liest, d.h. der Lesezugriff ist nicht *repeatable*.

  REPEATABLE READ Das non-repeatable-read Problem wird ausgeschlossen, allerdings kann noch *Phantomproblem* auftreten. Dabei kann eine Transaktion einen neuen Datensatz hinzufügen, der von einer anderen Transaktion eigentlich mit berücksichtigt werden müsste.

  SERIALIZABLE *Serialisierbarkeit* von Transaktionen ist gefordert.

- Mit dem Einfügen von *Savepoints* in einer (langen) Transaktion können Sicherungspunkte gesetzt werden, die bei einem Abbruch dazu führen, dass nicht die ganze Transaktion zurückgesetzt werden muss, sondern nur alles bis zum Savepoint.

- Eine *Historie* ist der Ablauf eine verzahnte Ausführung einer Menge von Transaktionen. Dabei kommt jede atomare Operation einer Transaktion nur einmal in der Historie vor, und die Reihenfolge, in der die Operationen einer Transaktion in der Historie auftauchen, ist invariant.

- Eine Historie ist *serialisierbar*, wenn die Ausführung der Historie das selbe Ergebnis liefert, wie eine serielle Ausführung.

- Zwei Operationen zweier Transaktionen stehen in *Konflikt*, wenn sie auf das gleiche Objekt zugreifen und mind. eine der Transaktionen eine Schreiboperation durchführen will.

- Um zu einer serialisierbaren Historie zu kommen kann man Nicht-Konfliktoperationen entsprechend vertauschen. Die Reihenfolge von Konfliktoperationen muss aber beibehalten werden!

- Der *Konfliktgraph* gibt an, wie die Konfliktabhängigkeiten unter den Transaktionen sind. Dabei stellt jede Transaktion einen Knoten dar, und es gibt gerichtete Kanten zwischen zwei Knoten T_i und T_j, wenn es ein Konfliktpaar $(op_i(x), op_j(x)$ gibt, d.h. $op_i(x)$ wird in der Historie vor $op_j(x)$ ausgeführt.

- Das *Serialisierbarkeitstheorem* sagt aus, dass eine Historie genau dann serialisierbar ist, wenn der zugehörige Konfliktgraph azyklisch ist.

14 Concurrency Control und Serialisierbarkeit

Als *Concurrency Control* bezeichnet man die Verfahren, die es ermöglichen nebenläufige Transaktionen (bzw. deren Operationen) so zu schedulen, dass die resultierende Historie serialisierbar ist. Man unterscheidet zwei grundlegende Arten von Concurrency Control:

Pessimistische Verfahren gehen davon aus, dass Konflikte zu einer nicht seriallisierbaren Historie führen. Die Benutzung von Sperren (Locks) ist die häufigste Methode.

Optimistische Verfahren gehen davon, dass schon alles gut gehen wird und die Operationen ausführt, und man erst im Nachhinein nachprüft, ob ein Konflikt aufgetreten ist. Bei optimistischen Verfahren arbeitet man auf Datenkopien.

Multiversion Verfahren arbeiten alle Transaktionen auf mehreren Versionen eines Datums.

14.1 Pessimistische Verfahren

- Beim *2PL* (2-Phasen-Sperrprotokol) wird folgendes verlangt:
  1. Jedes Objekt, dass in einer Transaktion benutzt werden soll, muss vorher gesperrt werden.
  2. Eine Sperre, die von der Transaktion schon gehalten wird, wird nicht erneut angefordert.
  3. Sperren von anderen Transaktionen müssen beachtet werden.
  4. Transaktion durchläuft eine Wachstumsphase, in der die Sperren erworben werden und eine Schrumpfungsphase, in der die Sperren wieder freigegeben werden. Ganz wichtig: *NO LOCK AFTER UNLOCK!*
  5. Sperren werden spätestens beim Transaktionsende freigegeben.
- Das 2PL-Protokoll gewährleistet die Serialisierbarkeit des Schedules.
- Beim 2PL-Protokoll kann es zu *kaskadierendem Zurücksetzen* kommen, da eine Transaktion T_1 in ihrer Schrumpfphase Sperren freigeben kann, die von einer Transaktion T_2 dann geholt werden. Wenn jetzt T_1 zurückgesetzt würde, dann müsste auch T_2 zurückgesetzt werden, weil diese Transaktion auf Daten gearbeitet hat, die nicht akzeptiert wurden.
- Das *strikte 2PL* verhindert kaskadierendes Rollback, in dem alle Sperren erst beim Commit freigegeben werden, es gibt also keine Schrumpfungsphase mehr.
- Beim *hierarchischen Sperren* wird die Sperrgranularität erhöht, so dass man auf verschiedenen Ebene (Datensatz, Seite, Segment, Datenbasis) sperren kann. Das erhöht dann natürlich die Parallelität. Dazu benötigt man zusätzliche Sperrmodi, nämlich IS (oder auch IR) (intention share/ read) und IX (intention exclusive),

die angeben, dass in einer unteren Ebene der Hierarchie ein R oder X Lock angefordert wurde. Die Sperrung erfolgt so, dass erst geeignete Sperren in allen übergeordnete Knoten erfolgt (top-down). Die Freigabe erfolgt dann bottom-up.

- Bei Sperrbasierten Synchronisationsmechanismen gibt es das unvermeidbare Problem der *Deadlocks*. Ein Deadlock tritt auf, wenn zwei Transaktionen gegenseitig auf Sperren vom jeweils anderen warten.
- Deadlocks kann man mit Hilfe eines *Wartegraphen* erkennen. Ein Wartegraph hat wie der Konfliktgraph die Transaktionen als Knoten und immer dann eine gerichtete Kante zwischen T_1 und T_2, wenn T_1 auf eine Sperre von T_2 wartet. Eine Verklemmung liegt genau dann vor, wenn der Wartegraph einen Zyklus enthält.
- Deadlocks werden durch das Zurücksetzen eines der verklemmten Transaktionen gelöst. Man kann nach verschiedenen Heuristiken vorgehen (jüngsten, den mit den wenigsten Sperren, mit den meisten Sperren, die Transaktion, die am seltesten zurückgesetzt wurde, die Transaktion, die in den meisten Zyklen auftritt ...)
- Man kann auch potentielle Deadlocks anhand der Aktivität einer Transaktion ausmachen. Sollte z.B. eine Transaktion länger als eine bestimmte Zeit gewartet haben, bricht man sie ab. Kann natürlich auch sein, dass sie gar nicht verklemmt war ...
- Mit *Preclaiming*, d.h. dass alle Sperren zu Beginn der Transaktion geholt werden (keine Wachstumsphase), werden Deadlocks verhindert.
- Mit Zeitstempeln kann man auch Verklemmungen vermeiden, da Transaktionen hier nicht mehr bedingungslos auf die Freigabe von Sperren durch andere Transaktionen warten. Es gibt zwei Strategien:

  wound-wait Ist T die ältere Transaktion und fordert eine Sperre an, die von einer jüngeren Transaktion gehalten wird, dann wird die jüngere Transaktion zurückgesetzt. Ansonsten wartet T auf die Freigabe.

  wait-die Ist T die ältere Transaktion und fordert die Sperre einer jüngeren Transaktion an, dann wartet T auf die Freigabe. Ansonsten wird T zurückgesetzt.
- Bei der *Zeitstempel-basierten Synchronisation* wird die Synchronisation anhand der Zeitstempel, die jeder Transaktion zu Beginn zugewiesen wird, durchgeführt (es werden also keine Sperren benötigt). Dabei erhalten ältere Transaktionen echt kleinere Zeitstempel als jüngere. Der entstehende Schedule ist dann äquivalent zu einer seriellen Abarbeitung in Reihenfolge der Zeitstempel.
- Beim Zeitstempel-basietem Verfahren werden jedem Datum zwei Werte zugeordnet: Zum einen $maxW(x)$ den Zeitstempel der letzten (jüngsten) Transaktion, die x geschrieben hat und $maxR(x)$, den Zeitstempel der jüngsten Transaktion, die x gelesen hat. Es wird jetzt so vorgegangen, dass Transaktionen, die zu spät kommen, zurückgesetzt werden:

- Transaktion will lesen: $maxW(x) > ts$, Transaktion wird zurückgesetzt. Andernfalls wird gelesen und $maxR(x) = \max\{ts, maxR(x)\}$
- Transaktion will schreiben: $maxW(x) > ts$ oder $maxR(x) > ts$, dann wird die Transaktion zurückgesetzt. Sonst darf geschrieben werden und $maxW(x) = ts$. Nach der *Thomas-Write-Rule* darf in dem Fall, wo nur $maxW(x) > ts$ und nicht $maxR(x) > ts$ gilt, statt die Transaktion gänzlich abzubrechen, einfach der Schreibvorgang abgebrochen werden, da ja in der seriellen Ausführung sowieso das Ergebnis von der jüngeren Transaktion überschrieben wird und keine andere Transaktion das Datum lesen sollte.

14.2 Optimistische Verfahren

- Bei optimistischen Verfahren wird die Transaktion in drei Phasen eingeteilt:

  Lesephase Alle Operationen der Transaktion werden auf lokalen Kopien der Daten durchgeführt.

  Validierungsphase In der Phase wird geprüft, ob die Transaktion in irgendeinen Konflikt mit einer anderen Transaktion geraten ist.

  Schreibphase Wenn die Validierung positiv verlaufen ist, dann schreibt in dieser Phase die Transaktion die ihre veränderten Daten fest in die Datenbasis ein.

- Beim optimistischen Verfahren gibt es kein kaskadierendes Zurücksetzen, da bis zur Validierungsphase noch keine Änderungen an der Datenbasis vorgenommen wurden, und somit keine andere Transaktion beeinflusst sein kann.

- Man unterscheidet das $ReadSet(T)$, das ist die Menge der Daten, die Transaktion T gelesen hat, und das $WriteSet(T)$, die Menge der Daten, die T geschrieben hat.

- Bei der *BOCC* (backward oriented CC) ist die Validierungsphase für T erfolgreich, wenn für alle Transaktionen T', die nach dem BOT von T commiten gilt, dass $ReadSet(T) \cap WriteSet(T') = \emptyset$.

- Beim *FOCC* (forward oriented CC) ist die Validierungsphase für T erfolgreich, wenn für alle noch aktiven Transaktionen T' gilt, dass $WriteSet(T) \cap ReadSet(T') = \emptyset$. Beim FOCC wird die Validierung von nur lesenden Transaktionen garantiert.

- Bei beiden Varianten kann immer nur eine Transaktion in der Validierungsphase sein, daher sollte die Validierung schnell gehen.

14.3 Multiversion Verfahren

- Beim *MVCC* (multiversion CC) ist die Idee, dass lesende Operationen immer einen konsistenten Zustand sehen, der auch nicht unbedingt der aktuellste sein muss. Dabei hat man immer mind. zwei globale Versionen eines Objekts vorliegen.

- Bei Lesetransaktionen T wird auf eine Sperre verzichtet. Eine Lesetransaktion liest die Version, die beim Start von T die aktuellste war. Eine solche Transaktion ist transaktionskonsistent.

- Eine Transaktion T mit Startzeitstempel ist *Transaktionslevel-konsistent* genau dann wenn für alle gelesenen Objekte gilt, dass die Version die jüngste commitete ist.

- Beim *Consistent Read* kombiniert man Read-only-Transaktionen mit sperrbasierten CC-Methoden. Beim Schreiben einer Schreib/Lesetransaktion sperrt man die aktuellste Version der Objekts, so dass andere Schreiber warten müssen, und erstellt dann selbst eine neue Version. Beim Lesen in einer Schreib/Lesetransaktion kann man einfach die letzte commitete Version lesen (ohne Locks). Dadurch erreicht man das Isolationslevel Read Committed.

- Mit der *Snapshot Isolation* will man das Isolationslevel Repeatable Read erreichen. Dazu muss man Konflikte zwischen Schreiboperationen vermeiden, daher müssen die $WriteSet(T)$ disjunkt sein. Zudem lesen alle Transaktionen die Version eines Objekts, die zu Beginn der Transaktion die aktuellste war (keine Lesesperre nötig). Falls dies nicht der Fall ist, muss eine Transaktion abgebrochen werden. Es gibt zwei Implementationsmethoden:

  First commit wins Updates werden lokal wie im optimistischen Verfahren behandelt. Bei der Validierung wird geprüft, ob Objekte, die gelesen wurden inzwischen schon verändert wurde oder nicht und demnach commiten. Es werden also keine Locks benötigt!

  Sperrbasierte Implementierung Beim Schreiben: Wenn die Transaktion älter als die aktuelle Version ist, wird abgebrochen. Ansonsten wird das Objekt per 2PL gesperrt. Wenn es schon durch eine andere Transaktion gesperrt ist, dann wartet man bis die andere Transaktion commitet, dann bricht man ab oder bis die andere Transaktion abbricht, dann commitet man selber. Es werden wieder keine Lesesperren benötigt.

- Beim *2VMVCC* hat man von jedem Objekt zwei Version:
  - Konsistente Version mit Zeitstempel der letzten modifizierenden Transaktion
  - Schreiber kann eine neue zweite Version erstellen, die nicht sichtbar bis zum Commit ist.

  Daraus folgt, dass nie zwei Schreiber gleichzeitig auf einem Objekt arbeiten können. Und eine neue Version kann nur freigegeben werden, wenn keine Lesenoperation auf einer alten Version aktiv ist. Man führt noch ein zusätzliches Certify-Lock ein, wenn eine veränderte Version die aktuellste konsistente Version werden soll. Diese Sperre ist unverträglich mit Lese- und Schreibsperren, dafür kann nun gleichzeitig gelesen und geschrieben werden, da ja mehrere Versionen vorliegen. Es werden die Leseoperationen bevorzugt.

15 Logging, Recovery

- Ein *Fail-safe-System* ist ein System, dass auch nach einem Fehlerfall in einem sicheren Zustand endet. Man erreicht ein fehlertollerantes System, in dem man Daten redundant speichert um mit entsprechenden Wiederherstellungsmethoden aus den redundanten Daten wieder einen konsistenten Zustand herzustellen.
- Man kann Fehler grob in drei Klassen einteilen:

  lokaler Fehler

  Fehler mit Hauptspeicherverlust

  Fehler mit Hintergrundspeicherverlust

- Bei *lokalen Fehlern* scheitert eine Transaktion. Das heißt, die von der Transaktion ausgelösten Änderungen müssen rückgängig gemacht werden (Undo).
- Beim *Hauptspeicherverlust* werden die vom DBS gepufferten Daten vernichtet. Da es zu langsam ist alle Daten nach einem Commit in die Datenbasis einzubringen, werden sie lokal gepuffert. Dadurch wird ein *Redo* nötig, das diese abgeschlossenen Transaktionen nachvollzieht. Gleichzeitig kommt es vor, dass noch nicht abgeschlossene Transaktionen zu materialisierten Änderungen in der Datenbasis geführt haben, die nun per *Undo* rückgängig gemacht werden müssen. Die dazu notwendigen Zusatzinformationen stehen in sogenannten *Logdateien*.
- Es gibt mehrere Strategien, nach denen Pufferseiten im Hauptspeicher ersetzt und ausgeschrieben werden:

  Not Steal Die Ersetzung einer Seite, die von einer noch aktiven Transaktion modifiziert wurde, wird ausgeschlossen.

  Steal Jede nicht fixierte Seite, d.h. jede aktuell nicht benutze Seite, kann prinzipiell eingelagert werden.

  Force Alle von einer abgeschlossenen Transaktion verursachten Änderungen werden beim Commit in die materialisierte Datenbasis eingebracht.

  Not Force Hier wird die Einbringung beim Commit nicht erzwungen.

- Je nach dem, wie man Steal und Force kombiniert, sind Redo und Undo unterschiedlich notwendig:

	force	not force
not steal	kein Redo und kein Undo	Redo und kein Undo
steal	Undo und kein Redo	Redo und Undo

- Es gibt verschiedene Strategien, nach denen Änderungen in die Datenbasis eingebracht werden:

  update-in-place Jeder Seite im Puffer wird genau ein Speicherplatz im Hintergrundspeicher zugeordnet. Falls eine Seite verdrängt wird, wird sie direkt an diese Stelle kopiert und überschreibt den vorangehenden Wert.

indirekte Einbringstrategie Hier wird die geänderte Seite an einem separaten Platz gespeichert, das System initiiert dann das Ersetzen der alten durch die neuen Seiten. Durch ein globales Bit wird angegeben, welche Seite den aktuellen Zustand hält. Nachteil ist hier, dass sich der Speicherbedarf verdoppelt.

- Man unterscheidet zwei Logtypen:

  logisches Log Hier werden keine Daten sondern nur Operationen geloggt.

  physisches Log Man loggt jede geänderte Seite, und zwar mit zwei Zuständen: Dem *Before-Image* für die Undo-Log-Daten (alter Zustand) und dem *After-Image* für die Redo-Log-Daten (neuer Zustand).

- Für das Schreiben von Logeinträgen gibt es das sogenannte *WAL-Prinzip*. Das Write-Ahead-Log besagt, dass bevor veränderte Daten in die Datenbasis eingebracht werden, das zugehörige Logfile geschrieben werden muss. Dies garantiert die Undo-Möglichkeit im Wiederherstellungsfall. Als zweites müssten vor einem Commit, alle zur Transaktion gehörenden Logeinträge geschrieben werden, dies garantiert im Fehlerfall das Redo.

- Mit *Checkpoints* kann man den Aufwand beim Recovery herunterschrauben, in dem das System nur bis zu den Sicherungspunkten zurückgerollt werden muss. Alle vor dem Sicherungspunkt geschehenen Änderungen sind uninteressant. Man unterscheidet drei Arten von Sicherungspunkten

  Transaktionskonsistente Sicherungspunkte Beim Setzen werden alle noch laufenden Transaktionen zuende laufen gelassen und neue Transaktionen werden bis nach dem Sicherungspunkt verzögert. Die Datenbasis enthält dann beim Sicherungspunkt nur erfolgreich abgeschlossene Transaktionen.

  Aktionskonsistente Sicherungspunkte Hier wird nur verlangt, dass Änderungsoperationen vollständig abgeschlossen sind, es liegt also nicht unbedingt ein konsistenter Zustand vor. Man benötigt allerdings keine Redo-Informationen mehr, die älter als der Sicherungspunkt sind, höchstens noch Undo-Informationen.

  Fuzzy Sicherungspunkte Hier werden nur die Kennungen der modifizierten Seiten in einer Log-Datei notiert.

- Die Wiederherstellung nach einer Fehler erfolgt in den folgenden Schritten:
  1. Jüngsten Checkpoint finden
  2. Analyse der Situation, was nach dem Checkpoint alles passiert ist. Winner sind dabei die Transaktionen, die schon commitet hatten und nun nachvollzogen werden müssen. Loser sind hingegen die noch aktiv waren und nun rückgängig gemacht werden müssen.
  3. Redo aller Transaktionen
  4. Undo der Loser