Philipps-Universität Marburg
Fachbereich 12 Informatik
VL Datenbanken
Prof. Seeger
Sommersemester 2013
Zusammengefasst von: Dirk Winkel

Zusammenfassung Skript

Inhaltsverzeichnis

1 Einführung

- Datenbanksysteme (DBS) zur Verwaltung großer, persistent zu verwaltender Datenbestände
- DBS bestehen aus Datenbankverwaltungssystem (DBMS) und Datenbank (DB)
- Anwendungen: Zahlreich > Big Data, zentralisierung von Daten
- Probleme Anwendungsprogramme (AWP): Datenaustausch, Inkonsistenzen, Aufwändig, Mehrbenutzerbetrieb, Datenverluste, Datenverknüpfung, ...
- Anforderungen: Gemeinstame Datenbasis, Mehrbenutzerbetrieb, Kontrollierte Datenredundanz, Sicherstellung der Datenqualität, Schutz, Benutzerschnittstellen, Unterstützung bei AWP-Entwicklung, Leistungsfähigkeit
- Abstraktionsstufen: Externe (Sichten), Logische (Datenbeziehungen), Physische (Speicherstrukturen)
- Datenmodell: Strukturbeschreibung (Datendefinitionssprache, DDL), Datenmanipulationssprache (DML)
- Arten: physisch (speicherorientiert), logisch (benutzerorientiert)
- Logische: relationales Modell, objektorientiertes Modell, XML, JSON, HDF, ...

2 Das relationale Datenmodell

- Entwicklung 1970 durch Edgar Codd
- Verwendung: Oracle, SQL Server, mySQL, PostgreSQL, Sybase, IBM DB2, ...
- Eigenschaften: Einfach (Relation>Tabelle, wenige Grundoperationen), Mengenorientierte Verarbeitung, formal fundiert

2.1 Relationen

- Relation (r) = Tabelle
- Relationenname = Tabellenname

- Relationenschema (RS) = "Spaltenbeschriftung" (Tabellenaufbau), REL(RS) = Menge aller Relationen über RS
- Tupel (t) = Tabellenzeile
- Formal: A: Attribut (Spaltenbeschriftung) mit dom(A) als Wertebereich, U: Menge aller Attribute
- t[X] = T eingeschränkt auf X (mit z.B. X={A})
- Integritätsbedingung: Einschränkung für Werte
- Schlüssel: Eindeutig (, Minimal); in jedem Relationenschema mindestens einer vorhanden
- Primärschlüssel: Maximal einer pro RS, Stellvertreter für Datensatz, im Schema unterstrichen, kann z.B. auch aus mehreren Fremdschlüsseln bestehen
- Konsistenz (von r): Alle lokalen Integritätsbedingungen sind erfüllt
- Datenbankschema (DS): Menge von Relationenschemata
- Datenbank (d): Menge von Relationen

2.2 Die relationale Algebra

- Ziel: Definition einer DML (> SQL)
- Anforderungen: Ausdrucksstärke (Berechenbarkeit), Effizienz (Komplexitätsklassen O), Einfachkeit (minimale Anzahl d. Operatoren)
- Algebra: Menge von Ankern N (z.B. Zahlen oder Boolean), Menge von Operatoren
- Relationale Algebra: Anker = Relationen, 6 Operatoren: $\sigma, \pi, \cup, \times, -, \rho$
- Projektion π: Filtern von Spalten einer Relation, Ausgabe ist temporäre Relation: $\pi_X(r) = s = \{t[X] | t \in r\}$
- Selektion σ: Filtern von Zeilen aus einer Relation mit boolscher Funktion F: $\sigma_F(r) = \{t | F(t) \wedge t \in r\}$
- Umbenennung ρ: Erzeugt neue Relation mit anderem RS: $\rho_{X \leftarrow A}(r) = s$ mit A als altem und X als neuem Bezeichner
- Vereinigung $\cup$: vereinigt zwei Relationen mit gleichem Relationnschemata: $r_1 \cup r_2 = s = \{t | t \in r_1 \vee t \in r_2\}$
- Differenz $-$: Mengendifferenz: $r_1 - r_2 = \{t | t \in r_1 \wedge t \notin r_2\}$
- Kartesisches Produkt $\times$: "Kreuzprodukt" (ergibt n*m Tupel): $r_1 \times r_2 = \{t | t[RS_1] \in r_1 \wedge t[RS_2] \in r_2\}$
- Höherwertige Operatoren: lassen sich aus Grundoperatoren erzeugen: $\bowtie, \cap, \div, \bowtie_\Theta, \ltimes$

- Einfacher Verbund $\bowtie$ (natural join): Vereinigung bei gleichen Inhalten gleicher Attribute:
  $r_1 \bowtie r_2 = \{t | t[RS_1] \in r_1 \wedge t[RS_2] \in r_2\}$
- Schnitt $\cap$: Nur gemeinsame Elemente:
  $r_1 \cap r_2 = r_1 - (r_1 - r_2)$
- Division $\div$: Ermittlung der Tupel mit Beziehung zu Tupeln anderer Relation:
  $r_1 \div r_2 = \{t | \forall u \in r_2 \exists v \in r_1 : t = v[RS_1 - RS_2]\}$
- Theta Join $\bowtie_\Theta$: Verknüpfung bzgl. eines Prädikates Θ:
  $r_1 \bowtie_\Theta r_2 = \sigma_\Theta(r_1 \times r_2)$
- Semi-Join $\ltimes$: Joinbeteiligte von nur einer Seite:
  $r_1 \ltimes r_2 = \pi_{RS_1}(r_1 \bowtie r_2)$

2.3 Das Relationenkalkül

- Ziel: Deklarative Beschreibung der Ergebnisrelation
- Syntax: r(t) (r Relation, t Tupelvariable), r[A]=u[B] (A, B Attribute)
- Formeln können zusammengesetzt werden mit $\wedge, \vee, \neg, \forall t : f(t), \exists t : f(t)$
- Freie Variable: uneingeschränkt, gebundene: Durch Formel eingeschränkt
- Ausdruck des Tupelkalküls: {t | f(t)} mit t als einzig freie Variable
- Substitution in Formeln: Schrittweises Auflösen einer Formel mit "wahr" und "falsch"
- Wert eines Ausdruckes {t | f(t)} = alle Tupel, die sich zu "wahr" auflösen
- Kurzschreibweisen:
  $\exists t : r(f(t)) := \exists r(t) \wedge f(t)$
  $\forall t : r(f(t)) := \forall t : \neg r(t) \vee g(t)$
  $f \Rightarrow g := \neg f \vee g$
- Das Tüpelkalkül ist ausdrucksstärker als die RA (unendliche Ergebnismengen möglich)
- Sichere Ausdrücke: Eingeschränkt auf endliche > gleiche Mächtigkeit wie RA

2.4 Erweiterung der relationalen Algebra

- Probleme: Mengensemantik hat keine Ordnung, Verdichten nicht möglich, Duplikate werden nicht erhalten
- M-Relationen (Multimengen-R.), mit speicherung der Vielfachheit, MREL(RS) = Mende aller M-Relationen über RS
- berücksichtigung im kartesischen Produkt, Summenvereinigung, Projektion
- Aggregation: Berechnung von Kennzahlen wie Summe (sum), Durchschnitt (avg), Anzahl (count), min/max:
  z.B. $avg_A(r)$ liefert numerischen Wert
- Gruppierung γ_X: Relation mit gleichem Schema wie M-Relation
  z.B. $gamma_{X,A1 \leftarrow count}(r)$ mit Anzahl der Tupel mit X

3 SQL - Die Abfragesprache für relationale Datenbanken

3.1 Allgemeines

- SQL: structured query language, seit 1974
- Marktbeherrschend, mischung aus TK und RA, unterstützt (Multi-)Mengensemantik, Aggregation, Gruppieren und Sortieren
- Grade der Standardisierung: Entry, Intermediate, Full
- Rekursion, Prozeduren, XML, ...
- beinhaltet DDL und DML
- unterstützung aus Java, C++, C#, Scala, ...
- nicht case-sensitiv, Befehle enden mit Semikolon, Kommentare mit –
- DDL: **create view** <name>, **create table** <name>,
  create index <name>, **create** database <name>
  Beispiel: **create table** Name(nummer **int primary key**, mname **varchar**(19)); (primary key ist Integritätsbedingung)
  Fremdschlüssel: attribut **int** references Name(nummer)
  Weitere Integritätsbedingungen: **not null**, **default** 'Wert'
- Primärschlüssel aus mehreren Attributen: **constraint** primkey **primary key** (a1, a2)
- Löschen: **drop table** xname>, Ändern: **alter table** <name> **add** <komponente>
- DML-Beispiel: **insert into** <name> [(<a1>, <a2>)] **values** (<k1>, <k2>), (<l1>, <l2>)

3.2 Anfragen in SQL

- Grundschema:
  select <Liste von AttrNamen> **from** <Liste von RelNamen> [**where** <Bedingung>]
- input: mehrere Relationen, output: eine temporäre Relation

3.2.1 Operatoren der relationalen Algebra in SQL

- Ausgabe einer Relation: **select** $*$ **from** <name>;
- Projektion: **select distinct** <a> **from** <r>; wobei disdinct Duplikate verhindert
- Selektion: **select** $*$ **from** <name> **where** <Bedingung>;
  Bedingungen: <, >, =, **in**, **not in**, **exists**, **any**, **some**, **all**, **and**, **or**, **not**, ...)
- Kartesisches Produkt: **select** $*$ **from** <name1>, <name2>;
- Vereinigung: **select** <a1> **from** <name1> **union select** <a2> **from** <name2>;, mit Duplikaten: **union all**
- Differenz: **select** <a1> **from** <name1> **except select** <a2> **from** <name2>;, mit Duplikaten: **except all**

- Umbenennung: **select** <a1> **as** <neu1>, <a2> als <neu2> **from** <name>;
- SWF-Anfrage: bestehend aus **select**, **from**, **where**,
  Umsetzungsreihenfolge: **from**, **where**, **select**

3.2.2 Joins

- Durch kartesisches Produkt und Selektion ausdrücken
- Equi-Join: **select** $*$ **from** r, s **where** r.A = s.B
- Semi-Join: **select** r.$*$ **from** r, s **where** r.A = s.B
- Das Datenbanksystem vermeidet dabei die Bildung des kartesischen Produktes
- Kartesisches Produkt: **from** r **cross join** s
- Innerer Theta-Join: **from inner join** s **on** r.A > s.B wobei **inner** jeweis optional ist
- Innerer Equi-Join: **from** r **inner join** s **using** (A)
- Natural Join: **from** r **natural inner join** s
- Outer Join: Ergebnisse des inneren Joins mit denen ohne Join-Partner, fehlende Werte werden **null**
  Beispiel: **from** r **left outer join on** r.A = s.B
- NULL-Werte: Kein Wert bekannt, eigentlich nicht Bestandteil des relationalen Modells
- automatische Eintragung verhindern: **not null** (Integritätsbedingung)
- Prüfen: is **null**
- erfordert dreiwertige Logik: true, false, unknown

3.2.3 Where-Klausel

- Ehere-Klausel entspricht im Wesentlichen der Formel des TK: "A <operator> B"
- between-Operator: A **between** B **and** C
- like-Operator: A **like** B Vergleich mit Wildcards: %: bel. viele Zeichen, _: genau ein Zeichen
  Beispiel: **select** A **from** r **where** B **like** 'M\%g_t'
- in-Operator: A **in** (b, c, ..) (A: Ausdruck; b, c, ..: Konstanten)

3.2.4 select-Klausel

- Duplikate beseitigen: **select distinct** $*$ **from** r, s **where** ...
- Umbenennung: **select** A **as** X, B$*$C **as** Y, abs(D) **as** Z **from** r; (vgl. Map-OP der RA)

3.2.5 Aggregate

- Numerische: **count**, **sum**, **avg**, **min**, **max**
- statistische: variance, corr, stddev, regr slope
- logische: **exists**, every, **any**, **some**
- Group-by-Klausel: ggf. alternative zu Aggregat, γ-Operator der RA
  Beispiel: **select** <nummer>, cout(∗) **from** r **group by** <nummer>
- Having-Klausel: Filtert Gruppen mit einem Boolean-Ausdruck
  Beispiel: **select** <nummer>, **avg**(zahl) **from** r **group by** <nummer> **having count**(∗)>2;
  (Nur Relationen mit mehr als 2 Tupeln)

3.2.6 Sortierte Ausgabe

- **order by** <a1>, <a2> **asc** nulls **last limit** 3 sortiert aufsteigend erst nach a1, dann nach a2, Ausgabe auf 3 beschränkt

3.3 Geschachtelte Anfragen

- Anstelle von persistenten Relationen können auch temporäre verwendet werden (Abfragen)
- Anstelle von Werten können Attribute temporärer Relationen verwendet werden
- Tupelvariablen: Werden in **from**-Klausel deklariert und binden an eine (temporäre) Relation an Beispiel: **select** a1.nummer, a2.nummer **from** r a1

3.3.1 from-Klausel

- Beispiel: **select** v.nr1 **from** (**select** nr1, nr2 **from where** nr3<100)v **where** v.nr2 < 3;

3.3.2 select-Klausel

- Bsb: **select** B.nr1, (**select count**(∗) **from** r A **where** a.nr1 = B.nr1 **and** nr2 < 5) **from** r B;
- Es lässt sich damit eine group-by-klausel nachbauen (aber ausdrucksstärker)

3.3.3 where-Klausel

- Beispiel: **select** nr1 **from** r **where** nr2 = 67 **and** nr3 < (**select avg**(nr3) **from** r);
- Unteranfragen mit **exists** sind möglich
- Tupelvariablen sind in Unteranfragen gültig, werden aber darin bei neudeklaration überdeckt
- korrelierte Unteranfragen lassen sich oft in gewöhnliche Anfragen transformieren
- rekursive Anfragen: umsetzung nur rudimentär, Anfragen über with-klausel

- with-Klausel: temporäre Relation kann in darauffolgenden SQL-Befehlen genutzt werden Beispiel: with r(nr1, nr2) **as** (**select** nr1, **avg**(nr3) **from** s **group by** nr1) **select** nr1 **from** r **where** ...
- with recursive: with **recursive** gauss(n) **as** (**values** (1) **union select** n+1 **from** gauss **where** n<100)
  select sum(n) **from** gauss;
- auswertung rek. Anfragen: nicht-rekursiver Teil: temporäre Arbeitsrelation bei auswertung des nicht-rekursiven Teils, dann der rekursive Term in eine temporäre Relation
- ggf. limit oder union all zwecks Abbruck einfügen
- konstante Relation: **values** (<k1>, <inhalt>, ..),(..) erzeugt eine konstante Relation

3.4 Einfügen, Löschen, Ändern

- Einfügen: **insert into** r [<Attribute>] **values** (<Konstanten>) (ggf. statt **values** eine SQL-Anfrage)
- Löschen: **delete from** r [**where** <bedingung>]
- Ändern: **update** r **set** a = a−1 [**where** ...]
- Loading: Einfügen beispielsweise aus csv-Datei, Spezialfunktionalität des DBMS

3.5 Anlegen von Datenstrukturen

- Index verbessert Antwortzeiten O(log n) statt O(n); Anlegen mit:
  creat [**unique**] indes <IndexName> **on** <Relation>(<Attribut>[<Ordnung>][,..])
  mehrere Indexe auf eine Relation möglich (unabhängig von Speicherorganisation)
- Sichten: Benutzerspezifische temporäre Ausschnitte der DB, logisch unabhängig
  create view <Sichtname> [(Attribut>[, ..]] **as** <Subquery> [with **check** option]
  with **check** option: Nur Datensätze, die auch wiedergefunden werden (zwecks änderung/einfügung > an Relation weiterreichen!)
- Namensraum: zur Vermeidung von Namenskonflikten
  create schema <Name> [authorization <Benutzer>] [schema_element]∗
- Wertebereich: Einschränkung durch Integritätsbedingungen:
  create domain <Name> [**as**] <Datentyp> [..]
- Datentypen: (Beispiele)
  bigint , **bit**, **bit varying**, **char**, **character varying**, **date**, **time**, xml, ...

3.6 Integritätsbedingungen

- statisch: beschreibt Datenbankzustand, dynamisch: DB-Änderungen
- Primärschlüssel: Ein einziger, eindeutiger Schlüssel in einer Relation: **primary key**
- weitere eindeutige Attribute: **unique** (null ist erlaubt, auch mehrfach!) Beide erstellen automatisch ein Index

- Fremdschlüssel: Schlüssel in einer anderen Relation (oder null)
  create table <Name>(<> **int** references R(r), ..)
  Referentielle Integrität wird eingehalten, löschen bei Fremdbezug nur durch Kaskadierendes Löschen: **on delete cascade on update set null** (alle abhängigen Werte werden auf null gesetzt)

- Integritätsbedingungen können z.B. zu bestimmten Zeitpunkten überprüft werden. **not** deferrabe: sofortige überprüfung, **deferrable initially** deferred: am Ende der Transaktion, .. **immediate**: vor Änderung. Ändern mit **set** ..

- Bedingung für Relation: **check**(note $>$ 0 **and** note $<$ 7), hinzufügen mit:
  alter table r **add constraint** v **check**(..)

- Assertion: Eigenständiges Objekt für Bedingung:
  create assertion <name> **check** <Bedingung>

4 Anwendungsprogrammierung

- Aufteilung: Datenbank, problemspezifische Operationen, GUI; wobei nur die Datenbank in SQL ist, der Rest z.B. Java

- SQL: Deskriptive Anfragesprache, aber geringer Funktionalitätsumfang für alltägliche Programmierung

- Impedance Mismatch: Mengenorientiert vs. Objektorientiert
  Kopplung über: Call-Schnittstelle wie JDBC, Vorübersetzer SQLJ, Frameworks Hibernate oder JPA, Erweiterungen PL/SQL ...

4.1 CALL-Schnittstelle

- in Java: JDBC (Bestandteil von Java: Paket java.sql)

- SQL-Anfragen werden durch den Linker als Strings im Compilercode eingebunden

- Rückgabe als Cursor

- Nachteile: komplizierte Programmierung, fehleranfällig; Vorteil: sehr Flexibel

- Wichtige Klassen: Connection, DriverManager, CallableStatement, PreparedStatement, Statement, ResultSet

- Client-Server-Kopplung: Nach Anmeldung der Treiberklasse durch
  Class.forName(“oracle.jdbc.driver.OracleDriver”);
  Connection con=DriverManager.getConnection(“<URL>”, “<user>”, “<passwort>”);
  Connection-Objekt sendet Anweisungen an Server, empfängt Resultate etc.

- Ausführen von Anfragen durch z.B. ResultSet executeQuery (String sql)

- PreparedStatement: Muss bei Objekterzeugung bis auf Parameter (? im SQL-Befehl) fertig sein:
  PreparedStatemend stmt = con.prepareStatement(..);
  Ausführen z.B. mit stmt(setInt (1,20);

- Abfrageergebnisse liegen als ResultSet, einem Cursor, vor. Methoden wie next(), findColumn(), getObject. Neuere JDBC-Versionen unterstützen Änderungsoperationen, auch Batch-Updates sind möglich
- Metadaten: Schnittstellen: ResultSetMetaData (Anzahl Spalten, Datentyp und Namen der Spalten) und DatabaseMetaData (Schema, also Namen der Relationen, und Eigenschaften des DBMS
- Ausnahmen: SQLException, unkritisch: SQLWarning
- SQL-Typen: Typen in unterschiedlichen DBMS ist unterschiedlich, in java.sql.Types finden sich einige generische Bezeichner
  bestimmten Typ auslesen mit getXXX, z.B. getInt (ebenso: setXXX)

4.2 Einführung in Hibernate

- Ziel: Objektrelationales Mapping (ORM): (Daten-)Objekte werden in der Datenbank persistent verwaltet.
  Vorteile: Indizierung, Konsistenzsicherung, hohe Verfügbarkeit, Mehrbenutzerbetrieb, integration voehandener Datenbestände, Performanz
- ORM-Architektur: OR-Mapping zwischen OO Anwendung und DB
- Problem: Impedance Mismatch: Objekte vs. Mengen
- Himernate: ORM für Java & .Net, unterstützt gängige DBMS
  Vorgehen: Top-Down (DB-Schema aus Klassenmodell generiert), Bottom-Up: Klassen aus rel. DB-Schema generiert, Meet-In-The-Middle
  Keine spezifischen Erweiterungen im Java nötig
- Hibernate Query Language (HQL): Eigene Abfragesprache von Hibernate
- Konfiguration: Enthält Verbindungsdaten zum DBMS, Einstellungen etc. und wird über XML-Datei hibernate.cfg.xml geladen.
- XML-Mapping: Zuordnung (Abbildung) Java-Klassen <-> Relationen über XML-Datei, Problem: Schlüssel vs. equals()
- POJO: Java-Klassen müssen "Plain Old Java Object" sein (Default-Konstruktor, Getter- und Setter-Methoden für alle Felder)
- Klassen von Hibernate: SessionFactory; Session: Laden, Speichern, ...; Querry: Anfragen in HQL
- Beziehungen: Fremdschlüssel vs. Assoziationen (Verweise)
- Unterschiedliche Implementierung Uni- und bidirektional
- n:m mit <many-to-many>-Tag möglich
- Is-A-Beziehung: keine direkte Unterstützung in SQL
  Möglichkeiten: eine Tabelle je (konkrete) Klasse oder eine Tabelle je Klassenhierarchie
- Identität: Objekt erhält Primärschlüssel Eigenschaft: objekt.getId(), Hibernate garantiert Persistenz nur innerhalb einer session!

- Anfragesprachen: HQL (Abfragen in SQL als vordefinierte Strings); Criteria API (Anfragen in Java), Problem: dynamische Erzeugung von Anfragen schwierig; Natives SQL per JDBC (kein ResultSet-Handling von Hibernate)
- Konkrete Umsetzung der Abbildungsdatei hier nicht dokumentiert!

4.3 Prozedurale Erweiterung von SQL

- SQL/PSM (Persistent Stored Modules) Teil des SQL-Standart
- Zur Ausführung von Funktionen auf dem Server: unabhängig vom Client, Performant (keine Kommunikation), Zentral
- Vorgehen: Implementierung,
  Installation auf Server: loadjava −user <> hello.class
  Registrierung: **create** procedure <> return ... bzw. **create** function
  Aufruf vom Client: CallableStatement <> = con−preparaCall(“{call ? HelloWorld()}”); ..

5 Datenbankentwurf

- Schritte: Anforderungsanalyse (Datenverarbeitung/Informationen); Konzeption; logischer Entwurf (DBMS berücksichtigen); Physischer Entwurf
- Anforderungsanalyse: Informations(struktur)anforderungen (Objekte/Attribute, Beziehungen, wie viele?); Datenverarbeitungsanforderungen (typischen Prozesse, Reihenfolge, Priorität der Operationen)
  Dazu Pflichtenheft mit Benutzern erstellen
- Konzeptioneller Entwurf: Datenbeschreibung in einer formalen Sprache (UML/ER-Modell)
- Logischer Entwurf: Abbildung der Konzeption in Datenstrukturen des logischen Datenmodells (z.B. Transformation in relationales Modell), redundanz vermeiden! unabhängig vom physischer repräsentation/DBMS!
- Physischer Entwurf: physische Umsetzung des logischen Entwurfes, Anlegen von Hilfsstrukturen wie Indexe etc., Hardwarestrukturen, ...

5.1 Entity-Relationship Datenmodell

- ER-Modell: Modellierung durch Abstraktion
- Beschreibung von Entitäten (Entities) mit Eigenschaften (Attributen) und Beziehungen (Relationships)
- Entitätstypen: (Entitätsmengen) Sturukturgleiche Entitäten
- Schlüssel: minimale Menge von Arrtibuten, die eine Entität eindeutig bestimmt
- Beziehungstyp: Menge strukturgleicher Beziehungen von n>1 Entitätstypen (n = Grad des Beziehungstyps); Kann weitere Attribute haben (Beispiel: kannMaschineBedienen, Note:1-6)

- Funktionalität von Beziehungstypen: 1:1; 1:M (eine aus E1 mit vielen aus E2, aber einer aus E2 nur mit einem aus E1); M:N (many-to-many)
- Graphische Repräsentation: Entitätstyp: Rechteck; Attribut: Ellipse, Schlüssel wird unterstrichen; Beziehungstyp: Raute; Funktionalität an Verbindungslinien schreiben
- min-max-Notation: min/maxanzahl der Beziehungen an Beziehungen schreiben (jeweils an die Entitätenseite!), * bedeutet beliebig viele, z.B.: (1,*)
- Id-Beziehungen: Eine Entität (schwache E.) hängt von der anderen (starken) ab (schwache und Beziehung in Doppelrechteck/-raute)
- Is-A-Beziehung: Eigenschaften werden vererbt (u.A. auch der Schlüssel)

5.2 Abbildung des ER-Modells in ein relationales Datenmodell

- Einfache Umsetzung, dann Konsolidierung
- (fast) 1:1 Entitätstypen in Relationen umsetzen, Schlüssel wird Primärschlüssel
- Jeder Beziehungstyp wird zu Relation, daber bilden Primärschlüssel die Schlüssel der Relationen (Achtung: Einduetige Attributnamen!), Attribute der Relation aufnehmen
- Konsolidierung: Vereinfachung des Datenbankschemas: Verschmelzen von Relationen mit 1:1, 1:N, N:1 (gleiche Primärschlüssel > outer-join)

5.3 Entwurfstheorie relationaler Datenbanken

- Ziel: Beurteilung der Güte eines DB-Schemas Vermeidung von Redundanzen und Anomalien, Informationsverlust, Effizienz
- Funktionale Abhängigkeiten (FD): statische Integritätsbedingungen, bei Anforderungsanalyse ermitteln
  Beispiel: {ShopName, Ware} -> {Preis} ShopName und Ware bestimmen eindeutig den Preis
- FD: $a \to B$ *trivial* wenn $B \subseteq A$; *voll* wenn keine echte Teilmenge C, sonst *partiell*; *transitiv* wenn nur über ein anderes Element.
- Ziel: Minimierung der FDs: Menge F von FDs zu einer (minimalen) äquivalenten Menge F_c reduzieren
- F^+: Menge aller gültigen FDs (Hülle), (3 Amstrong-)Axiome & Regeln:
  - *reflexiv*: $B \subseteq A \Rightarrow A \to B$
  - *Verstärkung*: $A \to B \Rightarrow A \cup C \to B \cup C$
  - *Transitivität*: $A \to B \wedge B \to C \Rightarrow A \to C$
  - *Vereinigung*: $A \to B \wedge A \to C \Rightarrow A \to B \cup C$
  - *Dekomposition*: $A \to B \cup C \Rightarrow A \to B \wedge A \to C$
  - *Pseudotransitivität*: $A \to B \wedge C \cup B \to D \Rightarrow A \cup C \to D$

- *Membership-Problem*: Test ob eine Abhängigkeit $A \to B \subset F$ ist: Ermittlung der Hülle A^+ der Attributmenge bzgl. der Menge F! Dann: $B \subseteq A^+ \Rightarrow A \to B \in F^+$
- F_c Kanonische Überdeckung von F falls $F_c^+ = F^+$ und $\forall A \to B$ gibt es keine "überflüssigen" As & Bs $((F_c \setminus A)^+ \notin F^+)$
- Algorithmus: Linksreduktion durchführen; Rechtsreduktion durchführen; alle $A \to \emptyset$ entfernen; alle $A \to B_1, ..$ ersetzen durch $A \to B_1 \cup ..$
- Zerlegung einer Relation zur Beseitigung von Anomalien; kein Informationsverlust, FDs erhalten!
  Hüllentreue zerlegung: $F^+ = (F_1 \cup ..)^+$
- Normalformen:
  - *1. NF*: Alle Attribute nur atomare Werte
  - *2. NF*: jedes Attribut prim (Teil v. Schlüsselkandidat) oder voll FD (nur verletzt wenn Schlüsselkandidaten zusammengesetzt)
    (kein Nichtschlüsselattribut ist abhängig von einem Teilschlüssel)
  - *3. NF*: Alle Abhängigkeiten haben Schlüsselkandidaten oder abhängige Elemente sind prim
    (kein Nichtschlüsselattribut hängt transitiv von einem Schlüsselkanditaten ab)
- Synthesealgorithmus: *Top-Down*: (Dekomposition: Universalrelation zerlegen bis alle in 3.NF; *Bottom-Up*: 3.NFs verschmelzen solange sie in 3.NF bleiben.
  4 Schritte: 1. Überdeckung der FDs ermitteln; 2. RS erzeugen aus FDs; 3. Falls kein Kandidatenschlüssel: RS mit K & F ($\emptyset$) erzeugen; 4. anderweitig enthaltene RS entfernen
- Boyce-Codd Normalform (BCNF): Alle "echte" (nicht-zirkuläre) Abhängigkeiten haben einen Schlüsselkandidaten (immer in 3.NF); Existiert zu jeder Menge von Attributen und FDs ohne Informationsverlust (aber nicht zwingend hüllentreu)
- Mehrwertige Abhängigkeiten (MVD): $A \to\to B$ z.B. bei ISBN und Autor möglich; bei mehreren MVDs: Dekomposition! (verlustfrei!)
- *4. NF*: (Verstärkung der BCNF) Jede nicht-triviale MVD enthält einen Schlüsselkandidaten ($A \to\to B$ *trivial* $\Leftrightarrow B \subseteq A \vee B = RS - A$)

5.4 Grenzen der Normalformen

- ggf. unpassend bei nur lesendem DB-Zugriff
- Beispiel: Star-Schema (z.B. "Data-Warehouse") liegt nicht in 3.NF vor (Star-Join)

6 Transaktionen

- Transaktionen (TA) müssen zahlreiche Bedingungen erfüllen (Konsistenz, Gleichzeitigkeit, Mehrbenutzer, Zuverlässigkeit auch bei Hardwareproblemen, ...)
- Elementaroperationen: Mit Auswirkung auf DB: Lesen, Wertzuweisung

- Ablaufsteuerung: Anfang einer Transaktion T: bot T.bot(); Ende: commit T.c(); Abbruch: abort T.a(); Ansonsten: T.r(), T.w() (lesen/schreiben)
- ACID-Bedingungen: TA ist Atomate Ausführungseinheit; Consistenter Zustand; Isoliert gegen andere TA; Dauerhaft; (Aufgaben des Transaktionsmanagers)
- (Elementar-)Operationen in sequentieller Ordnung im Ausführungsplan (& Historie)
- Probleme:
  - *lost update*: Gleichzeitiger Zugriff läßt ersten Zugriff ungeschehen machen (DB bleibt konsistent)
  - *inkonsistente Sicht*: Durch verzahlnte Operationen sieht die Datenbank nur inkonsistent aus
  - *inkonsistente DB*: Verzahnte Zugriffe sorgen für Inkonsistenzen
  - *Phantom-Problem*: Ein zwischen Lesen und Schreiben aller Tupel eingefügter Datensatz erscheint beim Schreiben als Phantom

6.1 Serialisierung von TAs

- Bei zwei Transaktionen auf dem gleichen Datensatz entsteht ein *Konflikt* wenn die Teiloperationen nicht vertauschbar sind. Wenn Ausführungpläne bei Konflikten identisch sind heißen sie *äquivalent*.
- Sequentiell: Transaktionen werden komplett nacheinander abgearbeitet
- Serialisierbar: $\exists$ ein äquivalenter sequentieller Ausführungsplan (genau dann, wenn ein Gerichteter Serialisierbarkeitsgraph (1 Knopen pro T, Kanten = Konflikte) zyklenfrei ist)

6.2 Syncronisationsverfahren

- Zwei Verfahren:
  - *verifizierend*: Ausführungsplan beobachten, bei nicht gegebener Serialisierbarkeit zurück
  - *präventiv:* nicht-serialisierbare Pläne verhindern (Sperrverfahren; Zeitstempel, Mehrversionen)
- Sperrvarfahren: Aktueller DB-Bereich wird gesperrt (Probleme mit Verklemmungen bei mehreren Sperren, Lösungen durch gemeinsames Sperren, alternativ: gemeinsames Entsperren)
  Auch nur Schreibsperren als Modus möglich
- Mehrbenutzersyncronisation: Isolationslevel: Erhöhung der Parallelität, aber Gefahr der Inkonsistenzen
  read uncommited, read committed, repeatable reads, seializable

6.3 Fehlerbehandlung

- Schutz vor Beeinträchtigung (User/system) durch Fehler oder Systemabsturz innerhalb einer TA, Lösung durch recovery-Komponente
- Fehlerklassen: Transaktionsfehler (rücksetzen von TAs), Systemfehler (Alle TAs rücksetzen), Speicherfehler (HD) (Rekonstruktion der DB)
- Lese/Schreiboperation: Lese Seite (Block), Fixiere Seite im Puffer, Sperre Datensatz, (TA), Zürückgeben der Sperre, Entsperren (Unfix)
- Bei Schreibpoerationen: DB kurzzeitig inkonsistent
- Varianten: (no-) steal: Freigabe vor (nach) dem commit; (no-)force: (nicht) schreiben bei commit
  Bei Inkonsistenz: REDO oder UNDO mit Protokoll
- Logging: Physisches (vorher- oder nachher-Zustand); Logisches (redo/undo); Physiologisches (beide)
- WAL (Write Ahead Logging): vor commit in Log-Datei schreiben!
- Nach Systemfehler: Analyse, Widerholung der Historie (Winner: schon committed), rückläufiges undo der Looser (uncommitted)
- redo ist idempotent (nach abbruch widerholbar), undo braucht Kompensationseinträge
- Sicherungspunkte: Um Logs (und damit Recoverykosten) klein zu halten:
  - *Transaktionskonsistente*: Puffer schreiben, neue Protokolldatei bei keinen (gesprerrten) TAs
  - *Aktionsbasierte*: Min_LSN aufnehmen (unfertige Änderungen zum Sicherungspunkt) ggf. schrittweises Weitersetzen des Sicherungspunktes
- Verlust des externen Speichers: Archivkopie mit Log-Datei!

7 Konzepte der Implementierung von DBMS

- Speichersystem > Zugriffssystem > Tatensystem > SQL-Shell

7.1 Speichersystem

- Zuordnung der Datenobjekte zu physischem Speicher
- Charakteristif von Festplatten: langsamer Direktzugriff, schneller Datentransfer > größere physische Einheiten (Seiten = Blöcke) (SQL: 8KB)
- Datei ist folge von Seiten, in PostgresSQL als Tablespace anzulegen; Speichersystem erlaubt Zugriff auf einzelne Seiten in Dateien
- Seitenzuordnung: Alles möglich zwischen statischer Datei-Zuordnung und dynamischer Block-Zuordnung (Adressierung über Tabelle)

- Beschleunigung durch Systempuffer

7.2 Zugriffssystem

- Tuple-Identifyer (TID, RowID oder RID): Seitenadresse und relative Adresse innerhalb der Seite; bei Migration in eine andere Seite: Stellvertreter-TID
- Recordmanager: Verwalter von Datensätzen (Suche/Spiechern/Clustern)
- Zugriff auf Tupel: Entweder Relationen-Scan oder Index-Scan (vordefiniert)

7.3 B^+-Bäume

- Gegensatz binärer Suchbaum: Für Externspeicher > viele Einträge in einem Knoten, Daten in Blattknoten, direkte Anpassung!
- Gegensatz: ISAM: statische Indexstruktur, periodische Reorganisation
- Binärer Suchbaum: O(log n), Speicher O(n); bei vielen Daten nicht für Externspeicher geeignet
- B^+-Baum Typ (b, c): Weg zum Blatt immer gleich; Knoten min 2, max 2b-1 Söhne, Blatt min c, max 2c-1 Einträge; Kosten O(h) (h=Höhe), h=O($log_b N$)
  - *Zwischenknoten*: Abwechselnd Zeiger auf Kindseite/Trennschlüssel
  - *Blattknoten*: Zeiger linkes Blatt | k | TID | .. | frei | Zeiger rechtes Blatt
- Die Trennschlüssel geben den Maximalwert an, der unter dem Zeiger davor zu finden sein könnte
- Kosten: Durch feste Höhe: O(h) mit h = Höhe des Baumes = konstant, Bereichssuche O(h+r/c) (r = Anz. der Antworten)
- Einfügen: Blatt suchen, in Blatt einfügen; falls voll: Aufspalten in je die Hälfte & Zeiger neu setzen (Elternknoten/Nachbarn)
- Löschen: Löschen im Blatt; falls zu wenig im Blatt: rechts o. links vereinigen (Zeiger!); Ständiges Aufspalten/Verschmelzen unterbinden durch Verschmelzen bei erst 33%
- Varianten mit diversten Optimierungen (Präfix-Bäume, Cache-sensitiv, ...)

7.4 Datensystem

- Schritte bei Anfrage-Bearbeitung: Formulieren; Übersetzen in Operatorbaum (logisch) > RA-Ausdruck; Auswahl physischer Operatoren; Ausführung
- Logische Operatoren: *Algebra*: Selektieren, Projektion, Vereinigung, Differenz, Kartesisches Produkt und Joins, Umbenennung;
  Gruppieren, Aggregate, Allgemeine Map-Funktionen, Sortieren
  erw. rel. Algebra: Multimengensemantig, Unterstützung von Duplikaten
- *Logischer Operatorbaum*: Interne Repräsentation der Anfrage durch Operatorbaum (RA), Optimierung durch algebraische Umformungen (günstige zuerst)

- Physische Operatoren: Implementierung der logischen, i.d.R. Vielzahl von Implementierungen (>100(0))
- Kopplungen der Daten von einem zum nächsten Operator:
  - *Nativ*: vollst. Verarbeitung, speichern in temp-Relation
  - *Datengetrieben*: partielle Verarbeitung der Ausgabe > schneller, Datenstrom
  - *Anforderungsgetrieben*: partielle Verarbeitung der Eingabe, auf Anforderung der 2. Operation (Konsument) > Schnell, Ergebnisse schon bei unvollständiger Verarbeitung, Abbruch günstig, kein Zwischenspeichern (auf Externspeicher)
- ONC-Schnittstelle: Umsetzung der angetriebenen Kopplung; Iteratoren Open (Iterator öffnen), Next (nächstes Element), Close (Iterator)
- Implementierung von Operatoren: Probleme: große Datenmengen, native zu teuer; Beispiele:
  - *Nested-Loops*: Duplikatbeseitigung: partielle Ergebnisse in Relation S Speichern, neues Tupel mit S vergleichen...
  - *Hashverfahren*: Für jeden Datensatz hash ermitteln und in Tabelle eintragen sofern noch nicht vorhanden (danach passen hoffentlich die Daten in den Hauptspeicher, sonst: z.B. rekursives Partitionieren)
    Kosten durch Partitionieren: O(n*N/B) (N: Datensätze, B: Seitenkapazität, n:Partitionen (die in Hauptspeicher passen))
  - *Sortieren*: Erst externes Sortierverfahren, dann lineares durchlaufen