Grundlagen

In einer Datenbank werden daten «bewusst» in einer strukturierter Form gehalten.

Informationen: vom Empfänger interpretierte Daten

Daten: maschinell verarbeitbar, weder wahr/falsch, wichtig/unwichtig. Sind an Datenträger gebunden. Daten müssen Konsistenz sein. Formatiert und unformatiert.

Datenbasis: strukturierte Sammlung von Daten

Datenbanksystem (DBS): besteht aus einem Datenbankmanagementsystem (DBMS) und einer (oder mehreren) Datenbasen (DB) [DBS = DBMS plus x Mal eine DB]

Anforderungen an DBMS:

- Redundanzfreiheit: Jedes Element nur einmal
- Datenintegrität:
  - Datenkonsistenz = logische Widerspruchsfreiheit
  - *Datensicherheit* = Sicherung von physischem Verlust
  - *Datenschutz* = Schutz Daten vor unberechtigtem Zugriff

Datenbankmodell (Paradigma)

- Legt Datenstrukturierungs-Konzepte des DBMS fest (Beziehungen)
- Legt die Datenstrukturen fest (Datentypen, Speicherung)
- Auch Paradigma genannt (in UML 2 Meta-Modell)

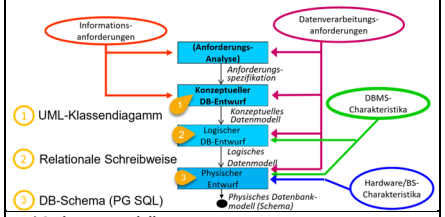
Modelle:

- Hierarchisches Datenmodell (legacy)
  - Struktur:
    - speichernden Informationen in einer einzigen Hierarchie
    - Beziehungen werden zusammen mit den Daten gespeichert
    - Ermöglicht einfache, sequentielle Verarbeitung
  - Nachteile:
    - Die 'Welt' lässt sich nicht immer als Hierarchie modellieren
    - Keine Trennung zwischen Anwendung (externes) und Daten (internes Schema)
    - Aufwendige Anpassungsarbeiten bei Änderungen
- Netzwerkmodell
  - Struktur:
    - speichernden Informationen in einer vernetzten Hierarchie
    - Beziehungen werden zusammen mit den Daten gespeichert
    - Mehrfachbeziehungen zwischen Datenelementen sind möglich -> Ermöglicht eine wirklichkeitsgetreuere Modellierung
  - Nachteile:
    - Effiziente Anwendungen nur möglich mit Kenntnis der komplexen Netzwerk-Datenstruktur, dadurch starke Abhängigkeit zwischen Anwendung (externem) und Daten (internem Schema)
    - Praktische keine kommerzielle Bedeutung
- Relationen Modell:
  - Struktur:
    - Informationen werden als unstrukturierte Daten (1. Normalform) in Form von Tabellen (=Relationen) gespeichert
    - sehr flexibel, praktisch alle Abfragen auf Daten möglich
    - ermöglicht eine klare und genaue Trennung zwischen Daten und Anwendungen
  - Nachteil:
    - Impedance Mismatch zwischen Typensystem der Programmiersprachen und den Tabellen Strukturen führen zu aufwändigen Konversionen beim Zugriff
    - Komplexe Abfragen können zu Performance-Problemen führen
  - Marktanteil: Alle gängigen Datenbanksysteme
- Objektrelationales Modell (ORDBMS)
  - Eigenschaften:
    - Erweiterung des rel. Datenmodelles: benutzerdef. Typen, TabellenVererbung
    - Verschachtelung von Tabellen (nicht unbedingt 1. Normalform)
    - neben Daten (=Objekt und Attribute) auch Methoden speichern
    - Evolutionärer Ansatz: Realisierung als Erweiterung eines rel. Datenbanksystems - SQL-Standard heute
- Weitere (NoSQL und NewSQL)
  - Eigenschaften:
    - Revolutionärer Ansatz: Neuimplementierung
    - Marktanteil gering/im Prozentbereich
    - Spezialanwendungen für bestimmte Anwendungszwecke
  - NoSQL (Besser «NotOnlySQL»):
    - Objektorientierte Modelle
    - Baum- und Graphen-orientierte Datenmodell
  - NewSQL:
    - Kolonnen-orientierte / In-memory Datenmodelle
    - Verteilte, cloudbasierte Datenbankmodelle

Datenbankmanagement (DBMS) Funktionen

- Speicherung /Persistenz (dauerhaft)
- Transaktionen («alles oder nichts» - Operationen)
- Mehrbenutzerbetrieb (Sync parallele Zugriffe)
- Sicherheit: Authentifizierung und Autorisierung
- Backup und Recovery
- Datentypen /Datenstrukturen mit Datenkatalog
- Abfragesprachen (z.B SQL) mit Optimierer
- Programmierschnittstellen (z.B JDBC)

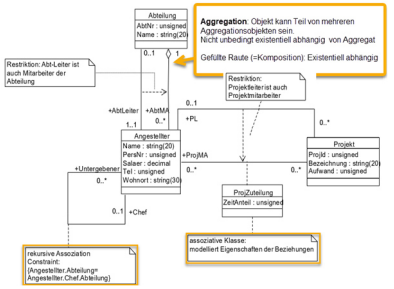
DB-Entwurfsprozess



Ansi-3-Ebenen-Modell

- Logische Ebene
  - Logische Struktur der Daten
  - Definition durch log. Schema «Trägermodell»
- Interne Ebene
  - Speicherstrukturen
  - Definition durch internes Schema
- Externe Ebene
  - Sicht einer Benutzerklasse auf Teilmenge der DB
  - Definition durch externes Schema
- Mapping
  - Zwischen den Ebenen ist eine mehr oder weniger komplexe Abbildung notwendig

UML-Klassendiagramm (UML = Unified Modelling Language)



Vererbung: «is-a»

- {complete} / {incomplete} sind zusätzliche Subklassen erlaubt?
- {disjunkt} Objekt ist Instanz von genau einer Unterklasse
- {overlapping}
  - Objekt kann Instanz von x überlappenden Unterklassen sein
  - Unterklassenzugehörigkeit kann wechseln
- *Krähenfuss-Notation*

Relationales Modell

Begriffe

- Entität: Inviduelles Element
- Relation: Beschreibt Entitätsmenge
  - Z.B: Ang (PersNr, Name, Chef, Salaer)
- Tupel: repräsentiert eine Entität
  - Z.B: (1001, «Marxer», 1000, NULL, 10580)
- Tabelle
  - Zeile=Tupel, Spalte=Attribut, Reihenfolge bedeutungslos
- Schlüssel
  - Attribute oder Kombi, die ein Tupel eind. identifizieren
  - Kombi muss minimal sein (Am wenigsten Attribute)
  - Primärschlüssel
    - Eindeutig (innerhalb Tabelle), Laufend zuteilbar, Möglichst kurz und nicht zusammengesetzt
  - Fremdschlüssel (Für Beziehungsmoddelierung)
    - Verweist auf Primärschlüssel anderer Tabelle

Regeln

- PK unterstreichen (default: *UNIQUE*, *NOT NULL*)
- FK Kursiv und mit *REFERENCES*
- Schlüsselkanditaten-Attribute mit *UNIQUE*
- Attribute (ausser PK!) default : *NOT UNIQUE*, *NULL*
- Attribut-Datentypen :
  - NUMBER (Allg.), DECIMAL, INTEGER
  - DATE, TIME, DATETIME, INTERVAL
  - VARCHAR(x) [maximal x Zeichen]
  - CHAR(x) [genau x Zeichen]
  - BINARY, ENUM
- Abbildung von Klassen (Regel 1)
  - Ein Attribut oder Kombi, muss PK sein
  - Nur Unstrukturierte Werte (1. Normalform)
  - Jede Klasse wird auf eine Tabelle abgebildet
- Abbilung von 1:n (Regel 2.1)
  - 1..* zu 1
  - Primärschlüssel der 1-Tabelle muss in der *-Tabelle als FK vorkommen
- Abbildung von optionalen Assoziationen (Reg. 2.2)
  - 0..1 zu 0..*
  - Variante a (mit optionaler Beziehungsattributen)
    - Bei häufigen Beziehungen
    - NULL-Werte für nicht vorhandenen Beziehungen
  - Variante b (Mit separater Beziehungstabelle)
    - Bei seltenen Beziehungen, Vorteil: Keine NULL-Werte
- Abbildung von Aggregationen (Regel 2.3)
  - Tupel in der abhäniger Tabelle ist vom referenzierten Tupel abhängig
  - Variante a: Zusammenges. PK (FK und weit. Schlüssel)
  - Variante b: Unabhängige Tabelle -> Eigener PK
- Abbildung von *:* Assoziationen (Regel 2.4)
  - Variante a: Beziehungsklasse
  - Variante b (Hauptsächlich): Assoziative Klasse
- Abbildung von Vererbung (Regel 3)
  - Regel 3.a / Variante 1
    - Je eine Tabelle pro Sub- und Superklasse
    - Zusätz. Diskriminierendes Attribut, das Typ definiert
    - PK der Subklassen = PK der Superklasse
    - PK der Subklassen = FK zu Superklasse
    - Vorteil: Flexibel, Redundanzfrei
    - Nachtteil: Viele Tabellen, komplexe Zugriffe
  - Regel 3.b / Variante 2
    - Je eine Tabelle pro Subklasse
    - Keine Abbildung der Superklasse
    - Subklassen erhalten Merkmale von Superkl.
    - Vorteile: Einfach Zugriff
    - Nachteil: Semantikverlust (Welche attribute gemeinsam?) Schlüsseleindeudigkeit über mehrere Tabellen verteilt, Assoziationen verteilt auf mehrere Tabellen, Überlappende Vererbung nicht abbildbar
  - Regel 3.c / Variante 3
    - Einzige Tabelle für Supperklasse
    - Super speichert alle Attributte (Auch Sub)
    - Je nach Sub sind Attribute, von anderem Sub NULL
    - Vorteil: Einfache Zugriffe
    - Nachteil: Viele NULL-Werte (3. Normalform verletzt)

Anomalien

- Einfügeanomalie
  - Tubel kann erst eingefügt werden, wenn andere Tupel (z.B in anderen Tabellen) existieren
- Löschanomalie:
  - Wenn ein Tupel gelöscht wird, gehen zusätzl. Infos verloren (z.B Mitarbeier Löschen, löscht auch Abteilung)
- Änderungsanomalie
  - Bei Änderungen muss an x Orten nachgetragen werden

Normalisierung

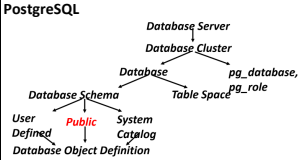
Stellt Redundanzfreiheit sicher, Verhindert Anomalien

- 1. Normalform
  - Bedingung: Werte atomar (z.B. «Hans», «Muster» statt «Hans Muster»)
  - Zusätzliche Zeile oder Spalte
- 2. Normalform
  - Bedingung: Jedes Nichtschlüsselattribut voll funktional abhängig vom gesamten Schlpssel
  - Bei nur einem Schlüssel => Automatisch
  - Zusätzliche Tabellen
- 3. Normalform
  - Bedingung: Zwischen zwei Attribute, die nicht zum Schlüssel gehören, darf keine Abhängikeit bestehen. (Doppelt gemobbelt, Änderungsanomalie tritt auf)

Relationale Algebra

$R \cup S$	**union**	
$R \cap S$	**intersection**	
$R \setminus S$	**set difference**	
$R \times S$	**Cartesian product**	select * from R1,R2
$\pi_{A1, A2, ..., An}(R)$	**projection**	select A1, A4 from Tabelle
$\sigma_F(R)$	**selection**	select * from Tabelle **where** A1 > 30
$R \bowtie S$	**natural join**	select * from R **natural join** S
$R \bowtie_\theta S$	**theta-join**	select * from R **join** S **on** R.a < S.b
$R \div S$	**division**	
ρ [A1 B1,.., An Bn]	**rename**	select A2 **as renamed** from Tabelle

PostgreSQL



- \c <db>: Connect to db
- \l: Alle Datenbanken anzeigen
- \d: Alle Tabellen anzeigen oder Tabellenschema
- \q: psql-Shell verlassen

DDL – Data Definiton Language (Schema)

Funktionen für SCHEMA, INDEX, TABLE, VIEW:

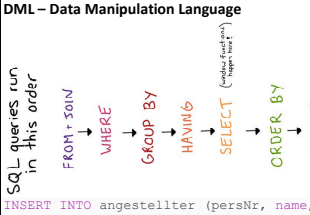
Erzeugen	Ändern	Löschen	Löschen Inhalt
CREATE...	ALTER...	DROP...	TRUNCATE ...

```
CREATE TABLE projekte (
  projNr INTEGER PRIMARY KEY,
  projLeiter REFERENCES angestellter ON DELETE SET NULL
);
CREATE TABLE angestellter (
  persNr INTEGER NOT NULL,
  chef INTEGER NULL,
  email VARCHAR(50) UNIQUE,
  salaer INTEGER,
  PRIMARY KEY (persNr, name),
  FOREIGN KEY (chef) REFERENCES angestellter
);
ALTER TABLE angestellter
  ADD CONSTRAINT constraintName Check(salaer between 1000 and 2000),
  ADD CONSTRAINT constraintName2 CHECK(alter <66);
```

Löschen eines Tupels einstellen (default: RESTRICT):

```
ON DELETE CASCADE --Alle Sub-Tupel werden auch gelöscht
ON DELETE RESTRICT -- Super-Tupel kann nicht gelöscht werden
ON DELETE SET NULL
ON DELETE SET DEFAULT
```

DML – Data Manipulation Language

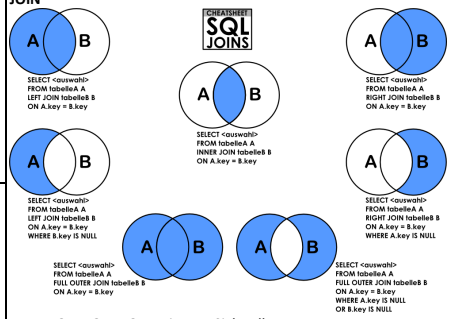



```
INSERT INTO angestellter (persNr, name, chef, salaer)
VALUES (101, 'Spring', NULL, 5000);
UPDATE angestellter SET salaer=6000 WHERE persNr=101;
DELETE FROM angestellter WHERE persNr = 1001;
```

```
(WITH..) SELECT… FROM… (JOIN …) WHERE … ORDER BY… (LIMIT…) -SFW
```

- WHERE-Prädikate
  - BETWEEN … AND .../ … AND …
  - IN (…,…,…) – Alternativ auch SubQUERY
  - LIKE '%B%', LIKE '_B_' - % beliebige.., _ ein… …Zeichen
  - AND/OR (Veknüpfung)
  - <> Ungleich
- DISTINCT: Unterdrückt Duplikate
- Aggregatsfunktionen (GROUP BY)
  - MAX (…), MIN(..)
  - AVG(…), SUM(…)
  - COUNT(…)
  - HAVING <-> WHERE
- NULL (WHERE blabla **IS** NULL !!)

JOIN



- *EQUI-JOIN:* JOIN mit «=», Sich selber
- *CROSS JOIN:* Kreuzprodukt (SELECT * from a,b)
- *LATERAL JOIN:* Zweite Tabelle als Subquery

```
select abtnr, name, maxsalaer from abteilung as abt
join lateral ( select max(salaer) as maxsalaer from
angestellter where abtnr=abt.abtnr) as ang on true;
```

Unterabfragen

Korreliert: Unterabfrage nicht selbständig (Abhängig)

Unkorreliert: Unterabfrage selbständig (Unabhängig)

- *IN*: Mehrere Tupel, Gelieferte Liste enthält
- *EXISTS*: Mehrere Tupel, Gelieferte Tabelle nicht 0
- ANY: Mehrere Tupel, Mindestens ein Wert aus liste
- *ALL*: Mehrere Tupel, Alle Werte aus der Liste

Mengenoperationen

- *UNION / UNION ALL*: Fügt zwei Tabellen zusammen, UNION ALL entfernt keine Duplikate
- *MINUS (EXCEPT)*: Differenz zwei Tabellen (JOIN)
- *INTERSECT*: Durchschnitt zwei Tabellen (EXIST)

Window Functions

```
SELECT abtnr, persnr, salaer, RANK() OVER (Partition BY abtnr ORDER BY salaer DESC) FROM ang ORDER BY 1;
```

Gibt eine Liste mit den Mitarbeitern aus, zusätzlich eine Spalte mit einem Rang des Salaers. Die Window Function schaut auf einen Teil der Tabelle und füllt je nach Funktion das Feld.

```
MIN(),MAX(),AVG(),SUM(),
LAG()--Differenz zum vorhergehenden
```

CTE (Common Table expression)

Hilfs-Query in einer WITH-Klausel (Temporäre Tabellen während des Statements)

Können…

- …SELECT, INSERT, UPDATE, DELETE enthalten
- …sich auf vorergehende Hilfs-Query beziehen
- …anstelle Subqueries verwendet werden
- … dem DB-Optimierer helfen
- … rekursiv sein

Syntax (*Normal*):

```
WITH nameTable1 AS ( SELECT * FROM myTable ),
nameTable2 AS ( SELECT * FROM nameTable2 )
SELECT * FROM nameTable2;
```

Rekursiv: zuerst Initialisierung, dann Rekursiver Teil

```
WITH RECURSIVE unter (persnr, name, chef) AS (
  SELECT A.persnr, A.name, A.chef FROM angestellter A
  WHERE A.chef = 1010
  UNION ALL
```

```
  SELECT A.persnr, A.name, A.chef FROM angestellter A
  INNER JOIN unter B ON B.persnr = A.chef
)
SELECT * FROM unter ORDER BY chef, persnr;
```

Views

Sicherheit: Irrelevante Daten für bestimmte Nutzer entfernen.
View ist eine virtuelle Tabelle basierend auf andere Tabellen oder Views. Daten werden zur Ausführzeit aus Tabellendaten hergeleitet. Man kann auch Queries damit vereinfachen.

```
CREATE VIEW AngPublic (Persnr, Name, Tel, Wohnort) AS
SELECT Persnr,Name,Tel, Wohnort FROM Angestellter;
```

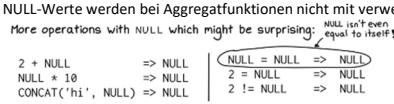
- Updatable Views:
  - Wenn nicht enthalten:
    - Join , Set-Operationen
    - Gruppen-Funktionen (min, max)
    - GROUP BY, CONNECT BY, START WITH
    - DISTINCT
- Weitere Views
  - Materialized Views:
    - Gespeicherte Views
    - nicht automatisch aktualisiert

```
CREATE MATERIALIZED VIEW name… ;
REFRESH MATERIALIZED VIEW name…;
```

  - Row-Level Security (RLS)
    - Eine Art "System-Views"
    - Nur User mit entsprechenedem Lese- und Schreibrecht («Policy»)
- Temporäre Tabellen (CREATE TEMPORARY TABLE):
  - Werden gelöscht (dropped) am Ende einer Session oder Transaction
  - Andere «permanente» Tabellen mit gleichem Namen sind nicht sichtbar

3-Wertige Logik und Null Werte

Logik: TRUE, FLASE, UNKNOWN
NULL-Werte werden bei Aggregatfunktionen nicht mit verwendet



Datenbank-Sicherheit (DCL – Data Control Language)

- System-Sicherheit (Allgemeiner Datenschutz)
  - Authentisierung: Benutzer muss sich identifizieren mit Passwort und Benutzername
  - Benutzerprivilegien (Autorisierung): Darf Nutzer Das
  - Kontrolle von System-Ressourcen, Auditing, Transportsicherheit
- Daten-Sicherheit (Datenschutz DBMS)
  - Zugriffskontrolle zur Verhinderung von nicht autorisiertem Zugriff auf Datenobjekte
  - Auditing von Zugriffsoperationen
  - DCL ist nicht standartisiert
- Massnahmen gegen Exploids, SQL-INJECTION:
- User-Input eingrenzen (Typ prüfen (z.B numeric), Escapen von nicht numerischen Input
- Datenzugriff abstrahieren (Stored Procedures)
- Spezieller Nutzer mit limiterten Rechte (~~Superuser~~)
- Benutzerverwaltung:
  - Jedem User sind Rechte zugeordnet für die Datenbankoperationen (Login/Connect)
  - Jedem User sind Rechte zugeordnet für die Verwaltung von Datenbankobjekte
- Schema:
  - Schema fassen Datenbank-Objekte zusammen in einer bestimmten Datenbank
  - Datenbank kann n-Schemas haben
  - Default: public
- Rolle (ROLE):
  - Oberbegriff für User oder Gruppen
  - Rollen gelten für den ganzen Cluster (über alle Datenbanken)
  - Rolle kann n-Schemas besitzen
- Benutzer: Role mit LOGIN
  - CREATE USER … ist identisch mit CREATE ROLE … WITH LOGIN
- Gruppe: ROLE ohne LOGIN
- Privilegien:
  - Systemprivilegien
    - Erlauben den zugriff auf die Datenbank-Operationen
    - CREATEDB, CREATEROLE

```
CREATE ROLE user CREATEDB NOCREATEROLE;
ALTER ROLE user WITH CREATEROLE;
```

  - Datenprivilegien
    - Erlauben den Zugriff auf die Datenobjekte
    - Grantor gewährt Privilegien
    - Grantee erhält Privilegien

```
GRANT object_privileg ON object TO {user|group|PUBLIC};
```

    - object_priv: - SELECT, INSERT, UPDATE, DELETE, REFERENCES, TRIGGER, ALL[
    - REVOKE – Gegenteil von GRANT (Entfernt die vergebenen Privilegien)
    - Gruppe - User zu Gruppe zuordnen: GRANT gruppe TO user;
      - Gruppe sind globale Objekte, nicht in einem Schema!
      - Nur Objektprivilegien

Transaktionen

Pro Session maximum 1 Transaktion
NESTED Transaktion nicht unterstützt.
Nutzen von Transaktionen:
Fault Tolerance: Bei Serverchrash kann operation wiederholt werden oder wird ganz gecancelt
Concurrency: Isolation der Transaktionen, Parallelität wird ermöglicht
ACID

- ***A**tomicity*: Vollständig oder gar nicht
- ***C**onsistency*: Konsistenter Zustand bleibt erhalten
- ***I**solation*: Transaktion soll von anderen Isoliert sein
- ***D**urability*: Alle änderungen sind persistent

```
BEGIN TRANSACTION; -- Kurznotation -> BEGINN;
COMMIT TRANSACTION; -- Kurznotation -> COMMIT;
ROLLBACK TRANSACTION; -- Kurznotation -> ROLLBACK;
SAVEPOINT blabla;
ROLLBACK TO blabla;
RELEASE blabla;
COMMIT;
```

- Commit Resultate
  - Success
    - Änderungen atomar und durable gespeichert
  - Failure
    - Alle temporären Änderungen werden abgebrochen (abort)
- Gründe für Abort
  - Explizit durch ROLLBACK oder ABORT
  - Unzulässige Verzahnung mit anderen nebenläufigen Transaktionen, Deadlock
  - Applikationsabbruch, Systemabsturz, Fehler

Serialisierbarkeit

wenn, nebenläufige Ausführung, gleich wie serielle Ausführung
Muss azyklisch sein -> Serialisierbarkeit keine Schlaufen
Beispiel:

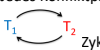
- T1 = r1(x)w1(x)r1(y)c1 und T2 = r2(x)w2(x)r2(y)w2(y)c2
- S = r1(x)r2(x)w1(x)w2(x)r2(y)r1(y)w1(y)w2(y)c2c1

Konfliktpaare :

- r1(x)→w2(x), r2(x)→w1(x), w1(x)→w2(x)
- Von w(x) zu allen ..(x) zeichnen, Pfeile links nach rechts
- Immer mit einem Write vorhanden

Serialisierungsgraphen

- Jedes Konfliktpaar eine Verbindung (Ausnahme T1 zu T1)



Zyklus ➔ Nicht serialisierbar

- Topologische Sortierung (Halbordnung) bestimmt Commit-Reihenfolge

Implementation der Isolation

- Pessimistische Verfahren
  - Sperrprotokolle
  - Besser bei hoher Konflikt-Wahrscheinlichkeit
- Optimistische Verfahren
  - Konfliktbehebung im Nachhinein
  - Besser bei kleiner Konfliktwahrscheinlichkeit

Locking

Locks auf verschiedene Granulitäten (Table, Table Range, Row, Item). Desto kleiner, desto grössere Parallelität. Psql meist ROW.

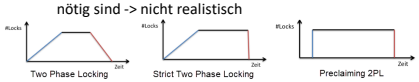
- Exclusive Lock (X)
  - Für Schreibe- oder Lesezugriffe
  - Nur eine Transaktion *xlock*
- Shared Lock (S)
  - Nur für Lese-Zugriffe
  - Mehrere Transaktionen *slock*

	S	X
S	✓	✗
X	✗	✗

Lock-Verträglichkeit

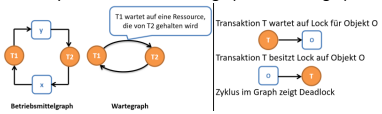
Locking garantiert keine Serialisierbarkeit ➔ zu frühes Unlock.

- Two Phase Locking (2PL)
  - Phase 1(Growing Phase): Objekt sperre vor Zugriff
  - Phase 2(Shrinking Phase): nach einem unock, kein lock
- Strict Two Phase Locking
  - Alle Sperren nach Ende der T freigeben
  - Vorteile zum 2PL
    - kein Cascading Rollback
    - kein unklarer Beginn der Shrinking Phase
  - Nachteile
    - Deadlocks sind möglich
    - schränkt Parallelität unnötig ein
- Preclaiming Two-Phase Locking
  - alle Sperren auf einmal (atomar) am Anfang
  - Alle Sperren nach Ende der T freigegeben
  - Vorteil: Deadlock-Freiheit
  - Nachteil: T muss im Vorherein wissen, welche Sperren nötig sind -> nicht realistisch



Deadlock Szenario

Gegenseitige Locks. Wird entweder durch Timeout (poor man solution) oder durch die Erkennung (Abbruch einzelner Transaktionen abgebrochen.
Für Analyse dient Betriebsmittelgraph und Wartegraph:



- Deadlock-Auflösung
  - Scheduler erkennt Zyklus im Wartegraph
  - Opfersuche bei Zyklus
    - Wähle eine Transaktion im Zyklus und breche diese ab
    - Minimierung des Rücksetzaufwands, T mit Wenigsten sperren
    - Maximierung frei gewordene Ressourcen: T mit meisten Sperren
    - Vermeidung der Starvation: Nicht immer dieselbe T rollbacken
    - Mehrfache Zyklen zerstören
  - Cascading Rollbacks: Implizite Rollbacks von weiteren T

Isolation

SERIALISED am besten, doch Parallelität limitiert, deshalb wird Effizienz mit schwächeren Levels gesteigert, auf kosten der Korrektheit. Fehler schwer nachvollziehbar.
4 Level nach ANSI SQL-92 Standard

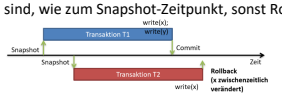
- READ UNCOMMITTED
  - Lesezugriffe nicht synchronisiert (keine read-lock)
  - Read ignoriert jegliche Sperren
- READ COMMITED
  - Lesezugriffe nur kurz/temporär synchronisiert (default)
  - setzt für gesamte T Write-Lock, Read-Lock nur kurzfristig
- REPEATABLE READ
  - einzelne Zugriffe ROWS sind synchronisiert
  - Read und Write Lock für die gesamte T
- SERIALIZABLE
  - Vollständig (korrekte) Isolation ACID

	Read Uncommited	Read Commited	Repeatable Read	Serializable
Dirty Write	möglich	möglich	möglich	unmöglich
Dirty Read	möglich	unmöglich	unmöglich	unmöglich
Lost Update	möglich	möglich	unmöglich	unmöglich
Fuzzy Read	möglich	möglich	unmöglich	unmöglich
Phantom	möglich	möglich	möglich	unmöglich
Read Skew	möglich	möglich	unmöglich	unmöglich
Write Skew	möglich	möglich	möglich	möglich
Deadlock			möglich	unmöglich
Cascading Rollback				unmöglich

- Dirty Read – Daten Lesen von anderer nicht commited T
- Fuzzy Read – Lese gleiche Daten mehrmals, sehe aber andere Werte, gelesene Daten ändern sich durch andere T
- Phantom – gleiche Selects entdecken neue/gelöschte Rows
- Serializable: Können Leser, Schreiber blockieren
- Read Commited: Können Schreiber, Schreiber blockieren
- Es gilt immer höchste Isolation
  - SET TRANSACTION ISOLATION LEVEL …;

Alternative: Optimistische verfahren ohne Locks.

- Jede Transaktion sieht Snapshot zu Start-Zeitpunkt
- Bei Änderung im Commit prüfen, dass Objekte unverändert sind, wie zum Snapshot-Zeitpunkt, sonst Rollback



- Multi-Version Concurrency Control (MVCC)
  - Mehrere Versionen pro DB-Objekt
  - Jede T hat Timestamp des Startzeitpunkts
  - write(x): neue Version x_t, t = Start-TimeStamp
  - read(x): letzte Version x_t, mit grösst. t ≤ Start-TimeStamp
- PostgeSQL unterstützt MVCC
  - Verfahren:
    - Update -> Tupels X-Lock -> Deadlocks möglich
    - Lesen -> keine Locks, nicht überprüft
    - Jedes Update führt zu neuer Version des Tupels
  - Verhalten:
    - write blockiert keine read, read blockiert keine write
    - write blockieren write, welche dasselbe Tupel modifizieren wollen
    - READ COMMITED: read nur mit derselben Verison
    - REPEATABLE READ od. SERIALIZABLE: ganze T nur mit derselben Version
    - Serialisierbar nur mit Level SERIALIZABLE

	Garantiert Serialisier-bar	Keine Deadlocks	Keine Cascading Rollbacks	Keine Konflikt-Rollbacks	Hohe Parallelität	Realistisch (ohne Voranalyse)
Two-Phase Locking	✓	✗	✗	✓	✗	✗
Strict 2PL	✓	✗	✓	✓	✗	✓
Preclaiming 2PL	✓	✓	✓	✓	✗	✗
Validation-Based	✓	✓	✗	✗	✓	✓
Timestamp-Based	✓	✓	✗	✗	✓	✓
Snapshot Isolation	✗	*) ✗	✓	✗	✓	✓
SSI	✓	*) ✓	✓	✗	✓	✓

*) Deadlock in PS

Log Files

- Write-Ahead Log (WAL)
  - Änderung der T in Log Schreiben (Flush)
  - Ebenso Commit in Log schreiben atomar (Flush)
  - Schlussendlich In-Place Updates in DB
  - Aufbau: [LSN, TaID, PageID, Redo, UNDO, PrevLSN]
    - LSN (Log Sequence Number): Eindeutig monoton ansteigend
    - TransaktionsID: T-Kennung, die die Änderung durchgeführt hat
    - Page: Page-Kennung, auf der die Änderung vollzogen wurde
    - Redo: Absoluter Wert nach der Änderung
    - Undo: Absoluter Wert vor Änderung
    - PrevLSN: vorhergehende Log-Eintrag der jeweilige T (effizienz)
- Nach Absturz Recovery: Undo nach Log für alle COMMIT T
  - Winners: Commited T müssen wieder REDO werden
  - Looses: Nicht abgeschlossene T werden UNDO werde

Backup

- Planung
  - Maximale Ausfallzeit, Zeit für Recovery
  - Was soll gesichert werden? (Read-Only via Import=
  - Wann soll gesichert werden?
    - Voller Backup (Log- und Datenfiles)
    - Inkremenelles Backup (Logfiles)
    - Export (Daten)
    - Online/Offline Backup
  - Speicherstrukturen
    - Spiegelung des Log-Files, durch DBMS, Durch OS/Hardware
- Backup-Arten
  - Logischer Backup – SQL DUMP (pg_dump)
    - Blockiert keine schreibende oder lesende T
    - Für Mittelgrosse Datenmengen
    - Interkompatibel mit neuen PG-Versionen, andere Maschienen
  - Physisches Backup – File System
    - Datenbank muss gestoppt werden
    - Schneller als Logische Backup
    - Passt nur zu deselben MAJOR Version
  - Andere: Cloud, ContinuosArchiving, Snapshot, Agent

Indexe

- Data Pages – Heap
- Index Pages – Suchbaum (B-Baum)
  - Wurzel-Knoten(Root-Node), Knoten(Node), Blätter(Leaf)
- Primär-Index immer unique

Arten:

- ISAM (Index-Sequential Access Method) [Historisch zuerst]
  - Einfügen und Suchen: einfach und schnell
  - Aktualisieren: schlecht
  - Daten werden über die Indexspalte aufsteigend sortiert
- B-Bäume
  - B für Balancierter Baum -> Überall gleich viele Stufen
  - Geclusterter Index: Nachbar Leafs referenzieren auch auf Nachbardaten
  - Grad k -> Leaf & Node[k ; 2k] Einträge, Root [1 ; 2k]
  - Garantierte Mindestauslastung beträgt 50% (Real:49.99% ->Root evt 1 Eintrag)
  - Einfügen:
    - 1. Suche nach dem Schlüssel -> Suche endet in den Blättern
    - 2. Füge den Schlüssel dort ein
    - 3. Ist der Knoten(Blatt) überfüllt?
    - 3.1 Neuer Knoten mit dem mittleren und der rechts liegenden Einträge
    - 3.2 Kleinste Eintrag im neuen Knoten in den Vaterknoten
    - 4. Ist der Vaterknoten (jetzt) überfüllt?
      -> Analog zu den Normalen Knoten
      -> Root überfüllt? -> Baum wächst
- B+-Bäume
  - Ein B-Baum: Daten oder die Referenz nur in den Blättern (Unterste Stufe)
  - Daten in den Blättern sehr selten
- Hash
  - Ordnet Key zu Einträgen
  - Problem: Overflow
- Spezielle Speicherstrukturen bei PostgreSQL

Indexe in PostgreSQL:

- B-Tree (Default bei PostgeSQL)
  - Universell: Bereichsabfragen, Vergleiche, Mustersuche
- Hash (Ab PG Version 9.3 nicht nutzen)
  - B-Tree in fast allen Aspekten besser
- GiST (Generalized Search Tree)
  - Baumartige Struktur
  - Range/Containment Search, K-Nearest-Neighbour Search
  - Für geometrische Datentypen, Volltextsuche
- GIN (Generalized Inverted Index)
  - Geeignet für Arrays
  - speichert effizient Duplikate
  - VS GiST: erstellen, update langsamer, Zugriff schneller
- BRIN (Block Range Index)
  - Speichert min/max-Werte als «Blöcke»
  - Gut für «Range Search»
  - natürlich benachbarte oder sortierte Daten
  - Nutzt natürliche Nähe, z.B. Einfügedatum, PLZ
  - kleiner Disk-Verbrauch
- Index-Variationen
  - Zusammengesetzter Index
    - Index über mehrere Attribute/Kolonnen

```
CREATE INDEX table_idx ON table(col1,col2);
```

  - Index mit INCLUDE

```
CREATE INDEX table_idx ON table(col1) INCLUDE(col2);
```

  - Partieller Index: Queries beziehen sich auf eine Selektion

```
CREATE INDEX t_idx ON table(c1) WHERE c2 IS NOT NULL;
```

  - Funktionaler Index
    - Index mit Funktion / Ausdruck

```
CREATE INDEX table_idx ON table(function(col));
```

Optimierungen (=> EXPLAIN ANALYZE)

Ist Query i.O.? / ist Schema i.O?. / DB-Konfiguration HW angepasst? / DB-Statistik nachgeführt? / richtige Indexe vorhanden? / Index geclustert? / DB-Architektur adäquat?
Logisch: Abfrage umformulieren, *Physisch*: Algorithmus ändern