Datenbanksysteme

- Relationale Algebra
- SQL - DQL
- SQL DDL und DML
- Funktionale Abhängigkeiten
- Normalisierung
- Transaktionen
- Recovery
- Lineare Anfrageoptimierung
- NoSQL
- Graph-Datenbanken

Relationale Algebra

Operatoren

- pi: Projektion
- sigma: Selektion
- kartesisches Produkt
  - Alle Tupel aus A jeweils verknüpft mit allen Tupeln aus B
- natural join
  - Teilmenge des kartesischen Produktes, bei der gleichnamige Attribute in A und B den selben Wert haben und als ein Attribut behandelt werden
- theta join
  - Teilmenge des Kartesischen Produktes, bei der jedes Tupel die Bedingung erfüllt
- semi join
  - gibt alle Tupel aus A aus, zu denen es eine Entsprechung in B gibt, so dass die Bedingung erfüllt ist
- beta
  - _[neu1<-alt1,neu2-<alt2]
- A geteilt B
  - alle Attribute von A, die nicht in B sind. Und dann alle Werte, für die in A jede Wertkombination aus B vorliegt
- Minus, Vereinigung, Schnitt
  - normale Mengenoperationen

Sonst Wichtig

- Bag-Semantik iirc
- Maximum finden: Relation minus alle Tupel, zu denen es einen Wert gibt der größer ist

SQL - DQL

Demo

```
SELECT a, b AS c, avg(d), 1000
    FROM f NATURAL JOIN g JOIN h ON g.a = g.h
    WHERE Name LIKE '%u%' AND NOT (Name LIKE 't%' OR Name LIKE '%t')
    GROUP BY a, c
    HAVING sum(e) > 1234
UNION (
    SELECT DISTINCT i
        FROM j JOIN k USING l LEFT JOIN m
    )
LIMIT 5678;
```

```
SELECT (SELECT COUNT(*)*1.0 FROM Training) / (SELECT COUNT (*) FROM Benutzer) AS FOO;
```

Bemerkenswert

- bei `GROUP BY` müssen alle selektierten Attribute, die nicht durch eine Aggregatfunktion gejagt werden, in der `GROUP BY`-Klausel stehen
- `LIKE` vergleicht den ganzen string, aber `%` ist eine Wildcart, also basically `.*`
- `HAVING` erlaubt quasi `WHEN` mit Aggregatfunktionen über `GROUP BY`-Mengen
- Sub`SELECT`s nach `UNION` oder `EXCEPT` kommen in Klammern
- `RIGHT` bzw. `LEFT JOIN` geben alle Tupel der spezifizierten Seite aus und ergänzen diese mit Daten der anderen Seite falls vorhanden
- nach `COUNT` kommt idR `(*)
- cast to float zum Teilen mit `*1.0`

SQL DDL und DML

DEMO

```
CREATE TABLE T (
    TID INTEGER PRIMARY KEY AUTOINCREMENT,
    W   TEXT DEFAULT 'blabla',
    R   REAL,
    B   BLOB,
    UID INTEGER UNIQUE,
    D   INTEGER CHECK ( D >= 19000101),
    DIM INTEGER CHECK ( DIM >= 0),
    K   INTEGER CHECK ( K >= 0 AND K <= 100 * DIM ),
    FOREIGN KEY ( W ) REFERENCES B ( BID ) ON DELETE CASCADE ,
    FOREIGN KEY ( UID ) REFERENCES U ( UID ) ON DELETE RESTRICT
);
```

```
ALTER TABLE T
    ADD TR INTEGER DEFAULT NULL REFERENCES B ( BID ) ON DELETE SET NULL;
```

```
ALTER TABLE Tabelle
    DROP/ALTER COLUMN Spaltenname
```

```
INSERT INTO Training (Wer, UebungsID, Datum, VALUES
    ("3", "103" ,"20240326", "60", "100"),
    ("1", "102" ,"20240326", "60", "350");
```

ON DELETE

- NO ACTION
- SET DEFAULT
- SET NULL
- CASCADE
- RESTRICT

Funktionale Abhängigkeiten

Minimale Menge funktionaler Abhängigkeiten finden

- FDs (Functional Dependencies) rechte Seite aufteilen
  - `A -> BC` wird zu `A -> B, A -> C`
- Linksreduktion
  - Attribute auf der linken Seite, die von anderen Attributen auf der linken Seite abhängen werden entfernt
  - `AB -> C, A -> B` wird zu `A -> C, A -> B`
- Ableitbare FDs entfernen
  - Redundanz durch transitive Abhängigkeiten loswerden
  - `A -> B, B -> C, A -> C` wird zu `A -> B, B -> C`
- FDs rechte Seite zusammenführen
  - `A -> B, A -> C` wird zu `A -> BC`

Schlüsselmengen

- XL
  - alle Attribute, die auf keiner rechten Seite vorkommen
  - muss Teil jedes Schlüssels sein
- XR
  - alle Attribute, die auf keiner linken Seite vorkommen
  - ist nie Teil eines Schlüssels
- XB
  - alle Attribute, die auf beiden Seiten vorkommen
  - kann Teil eines Schlüssels sein
- Schlüsselkandidaten
  - Alle Vereinigungsmengen von XL und allen Teilmengen von XB
- Ist ein Schlüsselkandidat tatsächlich ein Schlüssel?
  - alle Attribute müssen teil der Attributhülle `X*` vom Schlüssel sein
    - Prüfen mit dem RAP-Algorithmus
      - **R**eflexivität
        - Initialisierung
        - alle Attribute des Schlüssels in `X*`

        - **A**kkumulation
            - Wähle eine FD, bei der alle Attribute der linken Seite bereits in `X*` sind
            - die Attribute der Rechten Seite sind von `X*` abhängig und somit jetzt auch in X*
            - wiederhole bis `X*` stabil
        - **P**rojektion
            - Umfasst `X*` alle Attribute?
    - der Schlüssel muss minimal sein
        - deshalb mit den kleinsten Schlüsselkandidaten anfangen, die keine Teilmengen anderer noch nicht geprüfter Schlüsselkandidaten sind

Normalisierung

Normalformen

- **1. Normalform**
  - nur atomare Attribute
- **2. Normalform**
  - 1NF und
  - Jedes nicht zum Primärschlüssel gehörende Attribut ist vom Primärschlüssel voll funktional abhängig
  - nicht in 2NF ist also:
    - Schlüssel: `AB`
    - FD: `A -> C`
    - weil `C` nur von `A`, also nicht dem gesamten Schlüssel abhängt
- **3. Normalform**
  - 2NF und
  - Keine transitiven Abhängigkeiten zwischen einem Schlüssel und Nicht-Schlüsselattributen
  - nicht in 3NF ist also:
    - Schlüssel: `A`
    - FDs: `A -> B` und `B -> C`
    - weil `C` kein Schlüsselattribut ist und transitiv von `A` abhängt
- **Boyce-Codd-Normalform**
  - 3NF und
  - auch keine transitiven Abhängigkeiten über Schlüsselattribute
  - `R1 = {C, D, E},  F1 = {DE → C, C → D},  K1 = {CE, DE}` erfüllt die 3NF, aber nicht die BCNF
  - `R2 = {D, A, C, F }, F2 = {D → ACF, AC → D}, K2 = {{D}, {A, C}}` erfüllt die BCNF, aber nicht die 4NF
  - **Was ist der Unterschied zwischen diesen beiden? Warum erfüllt die eine die BCNF und die andere nicht???**
    - In `R1` ist `C` kein Superkey, in `R2` sind dies `D` und `AC` jedoch schon. Für die BCNF muss die linke Seite jedes FDs ein Superkey sein.

Syntheseverfahren

- dient zur Erreichung der 3NF

- garantiert Verbundtreue und Abhängigkeitstreue
  - Verbundtreue heißt, dass wenn man alle Relationen JOINt, man wieder die Ausgangsrelation erhält
- Vorgehen
  - Sortiere die FDs
    - absteigend nach Anzahl der beteiligten Attribute
    - absteigend nach der länge der linken Seite
  - führe für jede FD folgende Schritte durch
    - gibt es eine Relation, die bereits alle Attribute der FD enthält?
      - ja: füge die FD zu dieser Relation hinzu
      - nein: erstelle eine neue Relation mit der FD und all seinen Attributen
  - gibt es eine Relation, die den gesamten Schlüssel enthält?
    - wenn nein, erstelle eine solche

Dekompositionsverfahren

- dient zur Erreichung der BCNF
- garantiert Verbundtreue, nicht aber Abhängigkeitstreue
- Vorgehen
  - Wähle eine FD, die die BCNF verletzt (weil die linke Seite kein Superkey ist)
  - Teile die Relation auf in
    - alle Attribute außer der, der rechten Seite der FD
    - alle Attribute der FD
  - füge die betrachtete FD der zweiten Relation und alle anderen FDs **der transitiven Hülle**, deren Attribute in der ersten Relation enthalten sind, der ersten Relation hinzu
  - wiederhole dies solange, bis alle Relationen die BCNF erfüllen
- abhängig von der Reihenfolge der betrachteten FDs ergeben sich unterschiedliche Ergebnisse

Transaktionen

Anforderungen

- **A**tomic: Transaktion passiert ganz oder gar nicht
- **C**onsistent: Daten wiedersprechen sich nie
- **I**solated: Transaktion hängt nicht von anderer T ab
- **D**urable: Was commitet ist, ist dauerhaft gespeichert und überlebt einen Crash

Konzepte

- Historie: Abfolge von Operationen ???
- serialisierbar: Transaktionen lassen sich in eine eindeutige Reihenfolge bringen
- Konflikt: Operationen auf dem selben Objekt und mindestens eine Operation ist ein write
- Präzedenzgraf
  - 1 -> 2 -> 3 ist serialisierbar und legal
  - 1 -> 2 -> 3 -> 2 enthält einen Kreisschluss und ist damit nicht serialisierbar

Fehlerklassen

- dirty read
- lost update

2-Phasen-Sperrprotokoll

Legalität:

> Vor dem Lesen oder Schreiben eines Objektes muss immer eine Lese- oder Schreibsperre geholt werden. Eine Schreibsperre impliziert immer auch eine Lesesperre. Alle Sperren werden nach den Operationen wieder freigegeben.

Sperr-Regel:

> Es darf nur entweder eine Schreibsperre oder beliebig viele Lesesperren auf ein Objekt gleichzeitig geben.

Zweiphasigkeit:

> Die Anzahl der Sperren muss in der ersten Phase monoton zunehmen und in der zweiten Phase monoton abnehmen. Keine Sperre darf freigegeben werden, bevor die letzte angefordert wurde.

Um das Phantomproblem zu umgehen, sind Sperren logisch und beziehen sich nicht auf konkrete Tupel.

Parallele Validierung

Eine Transaktion hat drei Phasen:

1. Lesephase
2. Validierungsphase
3. Schreibphase

Transaktionen werden anhand des Endes ihrer Lesephase sortiert und müssen gegen alle älteren Transaktionen validieren, damit die zunächst auf einer lokalen Kopie vorgenommenen Änderungen in die Datenbank geschrieben werden. Schlägt die Validierung fehl, werden sie neu gestartet.

Validierungsphase von Tneu

Tneu unterteilt alle älteren Transaktionen in 3 Gruppen:

1. Talt1: Transaktionen, die beendet werden, bevor Tneu in der Lesephase ist
2. Talt2: Transaktionen, die beendet werden, während Tneu in der Lesephase ist
3. Talt3: Transaktionen, die beendet werden, nachdem Tneu in der Lesephase war

Validierung liefert genau dann true, wenn es keine Transaktion aus Talt2 gibt, deren writeSet eine Schnittmenge mit dem readSet(Tneu) hat und wenn es außerdem keine Transaktion aus Talt3 gibt, deren writeSet eine Schnittmenge mit der Vereinigung vom readSet(Tneu) und dem writeSet(Tneu) hat.

Recovery

Um Durability zu gewährleisten, müssen Datenbanken so arbeiten, dass committete Änderungen auch nach einem Crash noch zuverlässig Bestand haben, andere jedoch nicht. Dies ließe sich mit der Policy

- **non steal** (uncommitete Änderungen dürfen nicht auf die Festplatte geschrieben werden) und
- **force** (commitete Änderungen müssen sofort auf die Festplatte geschrieben werden)

erreichen. Dies würde allerdings die Parallelisierbarkeit viel zu stark einschränken, so dass reale Datenbanken fast ausschließlich steal und non force verwenden. Stattdessen verwenden sie ein sogenanntes Write-Ahead-Log (WAL).

- non force macht redo notwendig
- steal macht undo notwendig

Write Ahead Log

- Speichert jede Änderung
- muss bei jedem Commit auf die Festplatte geschrieben werden
- muss bei jeder Verdrängung von Seiten auf die Festplatte geschrieben werden
- In den Seiten auf der Festplatte wird vermerkt, von welcher LSN ihr Stand ist
- Attribute
  - LSN: Fortlaufende Nummerierung der Logeinträge
  - TransaktionsID
  - PageID
  - Redo
  - Undo
  - Prev-LSN: die vorherige LSN der gleichen Transaktion

Beispiel:

LSN	TA-ID	Page-ID	Redo	Undo	Prev-LSN
#1	T_1	BOT	-	-	-
#2	T_1	P_{CD}	$D-=10$	$D+=10$	#1
#3	T_2	BOT	-	-	-
#4	T_2	P_B	$B+=2$	$B-=2$	#3
#5	T_1	commit	-	-	#2
#6	T_3	BOT	-	-	-
#7	T_2	P_{CD}	$C+=17$	$C-=17$	#4
#8	T_3	P_A	$A-=15$	$A+=15$	#6
#9	T_3	commit	-	-	#8
#7	T_2	P_{CD}	$D+=3$	$D-=3$	#7

Recovery-Vorgang

- Redo **alle** Logbucheinträge, die noch nicht in der Datenbank stehen
- Undo alle Logbucheinträge von Loser(uncommitteten)-Transaktionen
  - Schreibe hierbei für jeden Eintrag einen CLR (Compensation Log Record) ins Logbuch
    - Ohne Undo-Eintrag
    - Mit Undo-Next-LSN
    - Mit <LSN>

Sollte es währenddessen zu einem erneuten Absturz kommen, sind diese CLR schon Teil der Redo-Phase und die Undo-Phase kann für jede Transaktion beim `Undo-Next-LSN` des letzten CLR fortgesetzt werden.

Checkpoints

brauchen wir, damit Redo und Undo nicht beliebig weit in die Vergangenheit zurück gemacht werden müssen

Arten:

- Transaktionskonsistente Sicherungspunkte
  - Alle Transaktionen zu Ende laufen lassen
  - keine neuen beginnen
  - Vorteil: Logbuch von davor kann weg
  - Nachteil: Datenbank lange komplett geblockt
- Aktionskonsistente Sicherungspunkte

    - wartet, bis laufende Operationen abgeschlossen sind
    - speichert aktive Transaktionen und minLSN
        - nur für diese muss im WAL zurückgegangen werden
- Unscharfe Sicherungspunkte
    - asynchron, mehr Aufwand
    - speichert dirty pages und deren LSN, aktive Transaktionen und minLSN
    - keine Unterbrechung des Datenbankbetriebs

Lineare Anfrageoptimierung

Datenbankanfragen haben das Potenzial, sehr ineffizient und langsam zu sein. Die Datenbank selbst optimiert zwar zu einem gewissen Grad, aber es macht definitiv Sinn, seine Queries auch von Hand auf Performance zu optimieren. Hierbei wird versucht, die Zahl der zu berechnenden Zwischenergebnise möglichst gering zu haltenn.

LAO in vier Schritten

1. Teile die Selektionsbedingungen so auf, dass sich jede bedingung auf genau eine Tabelle bezieht
2. Führe alle Selektionen, die sich nur auf eine Tabelle beziehen direkt auf dieser aus
3. Ersetze Kombinationen aus Selektion und kartesischem Produkt durch einen Join
4. Bestimme die Ausführungsreihenfolge aller Joins, die zu den kleinsten Zwischenergebnissen führt

Regeln

- Vereinigung, Schnitt, kartesisches Produkt und Join sind kommutativ and assoziativ
- Wenn die Selektionsbedingung eine Konjunktion ist, können Selektionen aufgesplittet und ihre Reihenfolge vertauscht werden
- Die Reihenfolge von Projektion und Selektion ist vertauschbar, wenn die Projektion alle für die Selektion benötigten Attribute liefert

Außerdem

- Projektion möglichst früh durchführen
- Reihenfolge innerhalb eines Joins spielt eine Rolle
- gemeinsame Teilanfragen wiederverwenden

NoSQL

Relationale Datenbanken sind unflexibel und skalieren schlecht horizontal. Um hier Abhilfe zu schaffen, gibt es NoSQL-Datenbanken mit weniger starrer Struktur und optimierung für horizontale Skalierung.

Arten von NoSQL-Datenbanken

- Key-Value: unter einem Schlüssel liegt ein Wert mit unbekannter Struktur
- Dokumentenorientiert: unter einem Schlüssel liegt ein Dokument mit abfragbarer Struktur
- Column Stores: Key-Value in geschachtelt bzw. mehrdimensional
- Graph: Knoten-Objekte und deren Beziehungen

Map-Reduce

- Idee: Programm kommt zu den Daten und kann auf vielen Maschinen parallel ausgeführt werden
- Es müssen nur zwei Funktionen implementiert werden
  - die Map-Funktion destilliert aus Key-Value-Paaren die gewünschte Sichtweise oder Aufteilung in Form von Key-Value Paaren
  - die Reduce-Funktion fasst alle Werte eines Keys zu einem Ergebnis zusammen. Gegebenenfalls lässt sie sich mehrfach hintereinanderreihen, um sukzessive die Ergebnisse zu sammenzuführen

Beispiel zur Berechnung eines Durchschnittswertes:

```
def map(self, data):
      yield ('count', (len(data[1]), 1))


  def reduce(self, key, values, rereduce):
      return (sum(value[0] * value[1] for value in values) /
          sum(value[1] for value in values),
          sum(value[1] for value in values))
```

Graph-Datenbanken

- basiert auf Beziehungen
- Schemalos und flexibel
- Joins müssen nicht teuer berechnet werden
- Sharding aufwändig

Knoten

- repräsentieren Objekte
- eindeutiger Identifier
- können mit einer Menge von Typen und Attributen verknüpft werden

Kanten

- Beziehungen zwischen Knoten
- eindeutiger Identifier
- genau einen Typ
- können mit einer Menge von Attributen verknüpft werden
- haben Start-, Endknoten und eine Richtung

Attribut

- Key-Value-Paare
- Key ist ein String
- Value ist ein atomarer Wert eines primitiven Datentyps

Beispiele

```
MATCH (tom:Person {name:"Tom Hanks"})-[r:ACTED_IN]->(m)
WHERE m.released >= 1900 AND m.released < 2000
RETURN tom,m;
```

```
CREATE (TheMatrix:Movie {title:'The Matrix', released:1999, Tagline:'Welcome to the Real World’})
```

```
CREATE (Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrix)
```

Allgemeine Syntax

- Knoten: „(“ variable? (“:“ label)* properties? „)“
- label: alle müssen erfüllt werden
- properties: {k:v, k:v, ...}, alle müssen erfüllt werden
- Kanten: („-“ | „<-“) („[„ kantenDetails „]“)? („-“ | „->“)
- Kantendetails: - Variablenname - :label – Liste von Typen (getrennt durch „|“; nur ein Typ muss erfüllt werden) - Pfadlänge – Beispiele: *, *4, *2..4, *..6 - properties – siehe Knoten