

Grundlagen

Aufbau Applikation -> DBMS(Datenbank-Management-System) -> Datenbanken

Persistenz: nicht-flüchtig gespeichert nach Transaktionsende

Konsistenz: Widerspruchsfrei, durch Integritätsregeln

Redundanz: Mehrfaches vorkommen von Daten => Lösen durch Normalisierung

Isolation: Keine Beeinflussung von Nutzern untereinander

Semantische Integrität: korrekt aus Fachsicht

Operationale Integrität: konsistenz / Integrität während Systembetrieb erhalten

ACID: Atomicity, Consistency, Isolation, Durability; Gegenpart bei Verteilten: BASE

Datenmodell: definiert Operatoren und Objekte (relational, netzwerk, xml,...)

Logisches Schema: Struktur der Daten

Physisches Schema: Indizes etc,

Entity Integrity: Eindeutigkeit der PKs

Referentielle Integrität: Datensätze erst löschen, wenn nicht mehr referenziert

Domänen Integrität: Werte atomar, liegen in definierten Wertebereichen (CHECK)

Surrogat-Schlüssel: künstlicher Primärschlüssel (id)

CAP-Theorem: Konsistenz, Verfügbarkeit, Toleranz gegen Netzausfall: nur 2 von 3 möglich

3Schichten-Modell:	Externe Ebene(Views)
	Logische Ebene(Schema)
	Interne Ebene (Indizes)

Physische Datenunabhängigkeit: Anwendung unabhängig von interner Ebene

Logische Datenunabhängigkeit: Anwendung fast unabhängig von logischer Ebene

Umsetzung von Vererbung

Single Table alles in einer Tabelle, die Kind-Attribute sind ggf. NULL

Vorteile: Redundanzfrei, keine JOINS

Nachteile: Viel NULL, Große Tabelle, nicht über mehrere Ebenen

Table-per-Class (Joined Subclass) : jede Klasse eigene Tabelle

Vorteile: Redundanzfrei, Einfacher Zugriff auf typenabhängige Attribute der Oberklasse, keine NULL-Werte

Nachteile: JOINS nötig => schlechtere Performance, Referentielle Integrität muss geprüft werden

Table-per-Concrete-Class (Leaf Model) : nur Tabellen für Klassen mit Instanzen

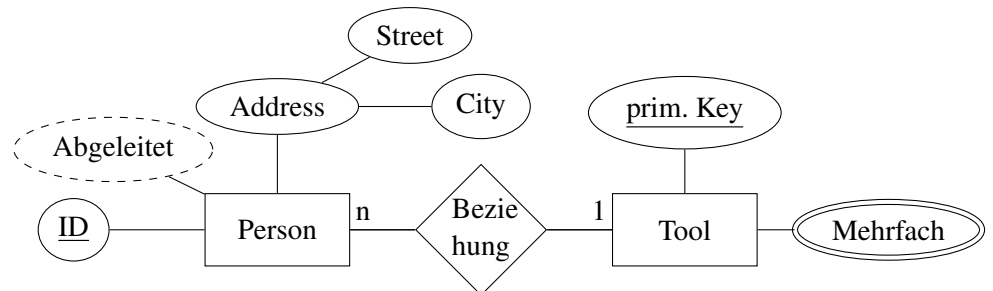
Vorteile: Typen leicht unterscheidbar, Objektinformation in einer Tabelle

Nachteile: ggf. Redundante Informationen, Anfragen über alle Objekte: UNION

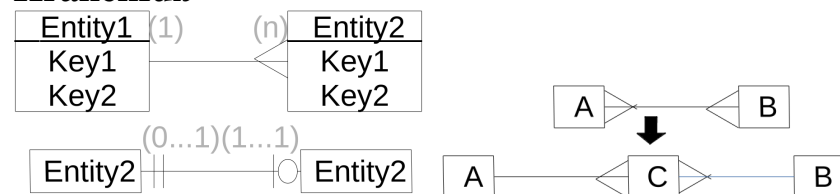
Kardinalitäten: 1:1, 1:n, n:m, + optionalität

Identifizierende Beziehungen: schwache Entität setzt andere Entität voraus (z.B. Bestellpositionen Bestellung)

Chen-Notation



Krähenfuß



Transaktionen

Transaktionsabbrüche durch Integritätsverletzungen, Konsistenzbedingungen, Speicher voll, Verbindungsabbruch, Systemausfall ...

Anomalien vs isolation level

1. Lost-Update: Überschriebene Änderungen bei parallelen Transaktionen
isolation level: read uncommitted
2. Dirty-Read: Lesen eines nicht committeten Wertes
isolation level: read committed
3. Unrepeatable Read: identische abfragen mit verschiedene Ergebnisse durch commit zweiter Transaktion
isolation level: repeatable read
4. Phantom Read: Einfügen von Datensätzen durch andere Transaktion
isolation level: serializable
5. —
isolation level: Snapshots (Multi-Version-Concurrency-Control) keine Lesesperren, änderungen erzeugen kopie

Serialisierbar: Gleiches Ergebnis wie bei hintereinanderausführung der Transaktion

z.B. $R_1(a)$, $W_2(a)$, $R_3(a)$, $W_1(a)$

T1 → T2
T2 → T3
T3 → T1 ⇒ Zyklus ⇒ nicht serialisierbar

Zwei-Phasen-Protokoll:

Sperren werden erst am Transaktionsende (vor commit) wieder freigegeben

Sperren: Shared-Locks (s) beim Lesen, Exklusiv-Locks (X) beim Schreiben

$R_1(a)$, $R_2(a)$, $W_1(a)$, $W_2(a)$

T1	T2
Slock(a), read(a);	Slock(a), read(a);
Wait(Xlock(a))	Wait(Xlock(a))
	DEADLOCK ⇒ Rollback;
write(a)	
unlock(a);	
commit;	

Anfrageverarbeitung

SQL -> Query Execution Plan:

Parsen der Anfrage -> Operatorengraph; // hierbei Standardisieren und vereinfachen

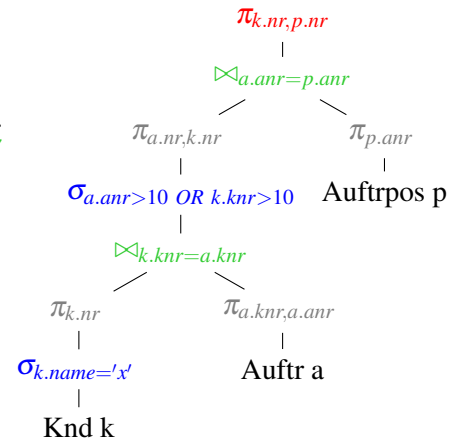
KNF : (a and b) or (c and d);

$$\text{deMorgan: } \overline{a \text{ AND } b} = \overline{a} \text{ OR } \overline{b}$$

$$\overline{a \text{ OR } b} = \overline{a} \text{ AND } \overline{b}$$

HINWEIS: von unten nach oben konstruieren.

SELECT $\underbrace{k.name, p.nr}_{\pi_{k.nr, p.nr}}$
FROM Knd k JOIN Auftr a $\underbrace{ON k.knr = a.knr}_{\Join_{k.knr=a.knr}}$
JOIN Auftrpos p $\underbrace{ON a.anr = p.anr}_{\Join_{a.anr=p.anr}}$
WHERE $\underbrace{k.name = 'x'}_{\sigma_{k.name='x'}}$ AND
 $\underbrace{(a.anr > 10 \text{ OR } k.knr > 10)}_{\sigma_{a.anr>10 \text{ OR } k.knr>10}}$



Kostenschätzung

Anhand von Statistiken, Histogrammen, etc.; evtl. Hints für Optimizer

Kardinalität: |A|: Anzahl der gelieferten Datensätze

Selektivität: $\text{Sel}(a) = \frac{\text{zurückgegebeneDS}}{\text{gesamtDS}}$

NumBlocks: $\frac{|R| \cdot \text{Grösse}_{\text{Datensatz}}}{\text{Blocksize}}$

Levels(I(R,A)): Höhe des Index auf A

Attribut = 'sth' ⇒ $\text{Sel}(A) = \frac{1}{|A|}$

Attribut IN $\{c_1, c_2, \dots, c_n\}$ ⇒ $\text{Sel}(A) = \frac{n}{|A|}$

$A > c$ ⇒ $\text{Sel}(A) = \frac{A_{\max} - c}{A_{\max} - A_{\min}}$

$A < c$ ⇒ $\text{Sel}(A) = \frac{c - A_{\min}}{A_{\max} - A_{\min}}$

$P_1 \text{ AND } P_2$ ⇒ $\text{Sel}(P_1) \cdot \text{Sel}(P_2)$

$P_1 \text{ OR } P_2$ ⇒ $\text{Sel}(P_1) + \text{Sel}(P_2) + \text{Sel}(P_1 \text{ AND } P_2)$

Fehlerbehandlung

Lokaler Fehler (in Transaktion): Rollback der Transaktion

Verlust des internen Speichers: Abarbeiten des Logs.

1. Redo-lauf (durchführen der geloggtten Änderungen);
2. Undo-lauf: rollback nicht committeter Änderungen;

Verlust des externen Speichers: Backup einspielen, inkrementell log-einspielen

Write-Ahead-Logging: logging vor commit, bei rollback wiederherstellen aus log;

Log beinhaltet Undo- und Redo-Informationen; (vor und nach Zustand)

logisches Logging: protokollierung der ausgeführten Befehle

physisches Logging: Kopien der Datensätze (vor/nach)

Steal-noforce: Steal: gepufferte, uncommittete Seiten können von anderer Transaktion eingelagert werden, zusammen mit zugehörigem Undo-logeintrag

NoForce: committete Seiten werden sofort im Redo-log vermerkt, irgendwann auf Platte geschrieben

Recovery time objective: max. Ausfallzeit; **recovery point objective:** max. Datenverlust

Dateiorganisation

Heap Speichern von Datensätzen in Einfügereihenfolge => unsortiert

Einfügen am Ende, löschen suchen und setzen eines Löschrbits;

Sequentielle: sortierte Speicherung;

einfügen: suchen, einfügen und Seite sortieren; löschen: suchen und Löschrbit setzen;

Index: B-Baum-Struktur;

clustered Index: Segmente selbst sind sortiert (vgl. Sequentiell)

Plan-Operatoren

- **Full-Table-Scan:** durchsuche gesamte Tabelle: $Cost = NumBlocks(R)$
- **Index-Scan:** Suche anhand Index: $Cost = Levels(Index) + Sel(P) \cdot |R|$
- **Nested-Loop-Join:** für jeden Block: durchlaufe die andere Tabelle
Ohne Index: $Cost = NumBlocks(R) * NumBlocks(S)$;
mit Index $Cost = NumBlocks(R) * Cost(IndexScan)$
- **Merge-Join:** Sortiere die Relationen nach Join-Attribut;
paralleles Durchlaufen der Paare in den sortierten Relationen;
 $Cost: Cost(Sort(R)) + Cost(Sort(S)) + NumBlocks(S) + NumBlocks(R)$
- **Hash-Join:** Teile kleinere Relation K in h Abschnitte, (Abschnitt < RAM);
durchlaufe die Abschnitte: Erstelle Hashtabelle, prüfe für jeden Datensatz der 2. Relation JOIN-Bedingung mit den zugehörigen Werten;
 $Cost: NumBlocks(R) + x * NumBlocks(S)$

BigData: 3+1Vs :

- Volume: große Datenmengen
- Velocity: hohe Erzeugungsgeschwindigkeit
- Variety: strukturierte, semistrukturierte, unstrukturierte Daten
- Veracity: geringe Qualität/Glaubwürdigkeit

NoSQL

Vorteile: kompakt, Änderungen betreffen nur eine Tabelle

Nachteile: Konsistenzsicherung, eingeschränkte Abfragemöglichkeiten

key-value stores

reine Zuordnung von key-value-paaren => einfaches Datenmodell, keine Integritätsbedingungen/Joins, schnell, gut aufteilbar, Abfragen ohne Key schwierig

Document stores

abspeichern von JSON/XML zu keys; => flexibles Datenmodell, ähnlich Key-Value

Column Family

Feste Column Families (z.B. meta, posts, ...)

Row Key -> Column Families -> Liste von Key-Values

gute Kompression, flexibles Schema, schnelles Schreiben

Graph DB

Knoten und Kanten, beide mit Eigenschaften

Vorteil: Graphalgorithmen (Tiefen/Weitensuche) anwendbar

Verteilte Architekturen

- shared Memory: Multicore-System
- shared Disk: mehrere Server nutzen zentrale Platten => Synchronisation aufwendig
- shared nothing: kein gemeinsamer Speicher

Sharding: Aufteilung der Datensätze;

Typisch: Kombination aus Replikation/Sharding

z.B. Tabelle(A,B,C,D) -> Server1(A,B), Server2(B,C), Server3(C,D), Server4(D,A)

Replikation: synchron (warten auf Bestätigung) vs asynchron (zwischenzeitlich inkonsistent)

SQL

Typen

Ganzzahlen: INT, SMALLINT, BIGINT

Gleitkomma: FLOAT, DOUBLE //nicht exakt

Festkomma: DECIMAL, NUMERIC // exakt

Zeichenketten: VARCHAR(n), TEXT

Zeit: DATE, TIME, DATETIME

Data Dictionary

Metadaten über den Aufbau der Datenbank;

z.B. SELECT * FROM information_ schema.tables;

Tabellen

CREATE TABLE <tabelle>

(<spalte> <typ> PRIMARY KEY NOT NULL | UNIQUE],

...

<tabellenconstraints> // z.B. PRIMARY KEY (Name, Ort)

)

Bsp: CREATE TABLE student

(id INT PRIMARY KEY,

name VARCHAR(10) NOT NULL,

lehrer INT,

CONSTRAINT fk_klassleiter FOREIGN KEY (lehrer) REFERENCES angestellte(pid) on delete cascade

)

ALTER TABLE <tabelle> <änderung>

- ALTER COLUMN <spalte> <typ>
- ADD COLUMN <spalte> <typ>
- ADD CONSTRAINT <constraint> // z.B UNIQUE, FOREIGN KEY ...

DROP TABLE <tabelle>;

Fremdschlüssel

CONSTRAINT <name> FOREIGN KEY (<spalte>) REFERENCES <tabelle>(<spalte>)

ON UPDATE/DELETE CASCADE/SET NULL

Anfrage

SELECT [DISTINCT] <spalte/Aggregation>

FROM <tabelle> JOIN <tabelle> ON <bedingung> Varianten NATURAL, LEFT, RIGHT

WHERE <bedingung>

GROUP BY <spalten> Alle spalten im SELECT auch im GROUP BY

HAVING <bedingung>

ORDER BY <spalten> ASC/DESC

Bsp: SELECT sg.Name, sg.Fak, SUM(s.Bafoeg), COUNT(DISTINCT s.Ort)

FROM Studenten s JOIN Studiengang sg ON s.SgNr = sg.SgNr

GROUP BY sg.Name, sg.Fak

HAVING COUNT(*) >= 2

Funktionen

COALESCE(<spalte>, <ersetzung>): Null-behandlung z.B. COALESCE(preis, 0)

Aggregationen: SUM, AVG, COUNT, MIN, MAX ...

Bedingungen

WHERE <spalte> BETWEEN <a> AND //Intervall [a,b]

WHERE <spalte> IN (x_1, x_2, \dots, x_n)

WHERE <spalte> LIKE <pattern> // _ beliebiges zeichen % beliebige kette

WHERE <spalte> IS NULL;

Subquery

FROM <subquery> ...

WHERE <spalte> [NOT] IN (<subquery>)

WHERE EXISTS (<subquery>)

z.B. SELECT * FROM Studiengang sg

WHERE EXISTS (

SELECT * FROM Studenten s

WHERE s.SgNr = sg.SgNr

)

Einfügen

INSERT INTO <tabelle> (<spalte1>, <spalte2> ...) VALUES (<wert1>, <wert2>, ...)

Es können auch mehrere Datensätze eingefügt werden

Verändern

UPDATE <tabelle> SET Ort = 'Berlin', Bafoeg = 0

WHERE <bedingung>

Löschen

DELETE FROM <tabelle>
WHERE <bedingung>

Mengenoperationen

Spalten müssen gleiche Typen haben
Vereinigung: <abfrage1> UNION <abfrage2>
Schnitt: <abfrage1> INTERSECT <abfrage2>
Differenz: <abfrage1> EXCEPT <abfrage2>

Transaktionen

BEGIN, COMMIT, ROLLBACK;
SQL-Isolationssteuerung:
SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }

Index anlegen

CREATE INDEX <name> ON <tabelle>(<spalte(n)>); DROP INDEX <name>

Handout Datenbanken

Normalisierung: Vorgehensweise

Ausgangsrelation

Verkäufe(Datum, KundenID, Name, Vorname, Wohnort, ProdID, Produkt, Marke, PgID, Produktgruppe)

Datum	KundenID	Name	Vorname	Wohnort	ProdID	Produkt	Marke	Menge	PgID	Produktgruppe
17.06.12	K1	Nuhr	Dieter	Düsseldorf	100	S 4	Samsung	2	1	Smartphone
17.06.12	K2	Pelzig	Erwin	Würzburg	101	iPhone 5	Apple	1	1	Smartphone
31.08.12	K3	Gruber	Monika	Erding	702	iPad	Apple	2	2	Tablet
12.10.12	K2	Pelzig	Erwin	Würzburg	702	iPad	Apple	3	2	Tablet
18.12.12	K3	Gruber	Monika	Erding	115	Galaxy 10	Samsung	2	2	Tablet
23.12.12	K3	Gruber	Monika	Erding	366	MacBook	Apple	1	3	Notebook
23.12.12	K2	Pelzig	Erwin	Würzburg	587	Vaio	Sony	1	3	Notebook
27.12.12	K1	Nuhr	Dieter	Düsseldorf	100	S 4	Samsung	1	1	Smartphone

Überführung in 1 NF

Sofern notwendig: Auflösung mehrwertiger Attribute in Extra-Relationen

Überführung in 2NF: Eliminiere partielle Abhängigkeiten von Primärschlüssel-Teilen

Schritt 1: Volle funktionale Abhängigkeiten identifizieren (sofern nicht vorgegeben)

Voraussetzung: Identifikation des Primärschlüssels der Relation (hier: KundenID, ProdID, Datum).

Gibt es Attribute, die von unterschiedlichen Teilen des Primärschlüssels abhängen?

- KundenID, ProdID, Datum → Menge
- KundenID → Name, Vorname, Wohnort
- ProdID → Produkt, Marke, PgID, Produktgruppe

Schritt 2: Zerlegung

- Verkäufe_Neu(Datum, KundenID, ProdID, Menge)
- Kunden(KundenID, Name, Vorname, Wohnort)
- Produkte(ProdID, Produkt, Marke, PgID, Produktgruppe)

Schritt 3: Integritätsbedingungen angeben

- Verkäufe.KundenID referenziert Kunde.KundenID
- Verkäufe.ProdID referenziert Produkte.ProdID

Überführung in 3NF: Eliminierte funktionale Abhängigkeiten zwischen Nicht-Schlüssel-Attr.

Schritt 1: Transitive funktionale Abhängigkeiten identifizieren

- ProdID → PgID und PgID → Produktgruppe
- Betrifft nur Relation Produkte

Schritt 2: Zerlegung von Produkten

- Produkte_Neu(ProdID, Produkt, Marke, PgID)
- Produktgruppen(PgID, Produktgruppe)

Schritt 3: Integritätsbedingungen angeben

- Produkte_Neu.PgID referenziert Produktgruppen(PgID)