

Grundlagen

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

Persistenz: nicht-flüchtig gespeichert nach Transaktionsende

Semantische Integrität: korrekt aus Fachsicht

Konsistenz: Widerspruchsfrei, durch Integritätsregeln

Operationale Integrität: konsistenz / Integrität während Systembetrieb erhalten

Redundanz: Mehrfaches vorkommen von Daten => Lösen durch Normalisierung

Isolation: Keine Beeinflussung von Nutzern untereinander

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

Modellierung

3Schichten-Modell:

Externe Ebene(Views)

Logische Ebene(Schema)

Interne Ebene (Indizes)

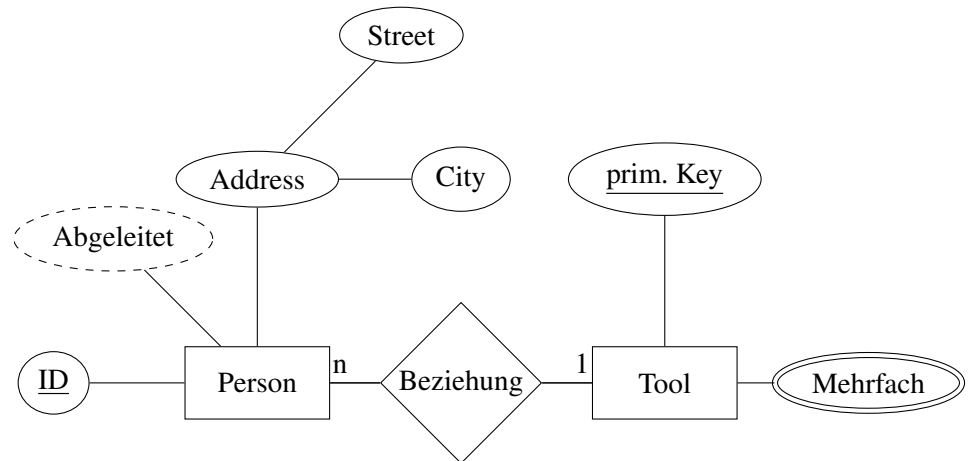
Physische Datenunabhängigkeit: Anwendung von Änderungen auf Interner Ebene unberührt

Logische Datenunabhängigkeit: auf Logische Ebene "kaum"berührt

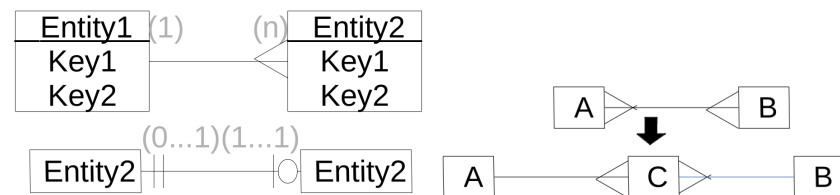
Kardinalitäten: 1:1, 1:n, n:m, + optionalität

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

Chen-Notation



Krähfuß



Umsetzung von Vererbung:

- Single Table (eine Tabelle, die Kind-Attribute sind ggf. NULL):
Vorteile: Redundanzfrei, keine JOINS, Auch Elemente speicherbar, die nur zur Eltern-Klasse gehören;
Nachteile: Viel NULL, Große Tabelle, nicht über mehrere Ebenen
- Table-per-Class: jede Klasse eigene Tabelle
Vorteile: Redundanzfrei, Auch Elemente speicherbar, die nur zur Eltern-Klasse ge-

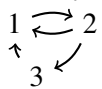
- hören, Einfacher Zugriff auf typunabhängige Attribute der Oberklasse, keine NULL-Werte
- Nachteile: JOINS nötig => schlechtere Performance, Referentielle Integrität muss geprüft werden
- Table-per-Concrete-Class: nur Tabellen für Klassen mit Instanzen
- Vorteile: Typen leicht unterscheidbar, Objektinformation in einer Tabelle
- Nachteile: ggf. Redundante Informationen, Anfragen über alle Objekte durch UNION

Transaktionen

SQL: BEGIN, COMMIT, ROLLBACK;
 Transaktionsabbrüche durch integritätsverletzungen, Konsistenzbedinungen, Speicher voll, Verbindungsabbruch, Systemausfall
 Nebenläufigkeit:

1. Lost-Update: Überschriebene Änderungen
2. Dirty-Read: Lesen eines nicht commiteten Wertes
3. Unrepeatable Read: durch commit anderer Transaktion liefert die selbe anfrage nicht das gleiche Ergebnis
4. Phantom Read: Einfügen von Datensätzen durch andere Transaktion

Serialisierbar: Gleiches Ergebnis wie bei hintereinanderausführung der Transaktion ⇔ kein Zyklus im Abhängigkeitsgraph z.B. $R_1(a), W_2(a), R_3(a), W_1(a)$



Sperren: Shared-Locks (s) beim Lesen, Exklusiv-Locks (X) beim Schreiben
 Zwei-Phasen-Protokoll: Sperren werden erst am Transaktionsende (vor commit) wieder freigegeben

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

SQL-Isolationssteuerung: SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }
 Multi-Version-Concurrency-Control: Snapshots: keine Lesesperren, änderungen erzeugen kopie

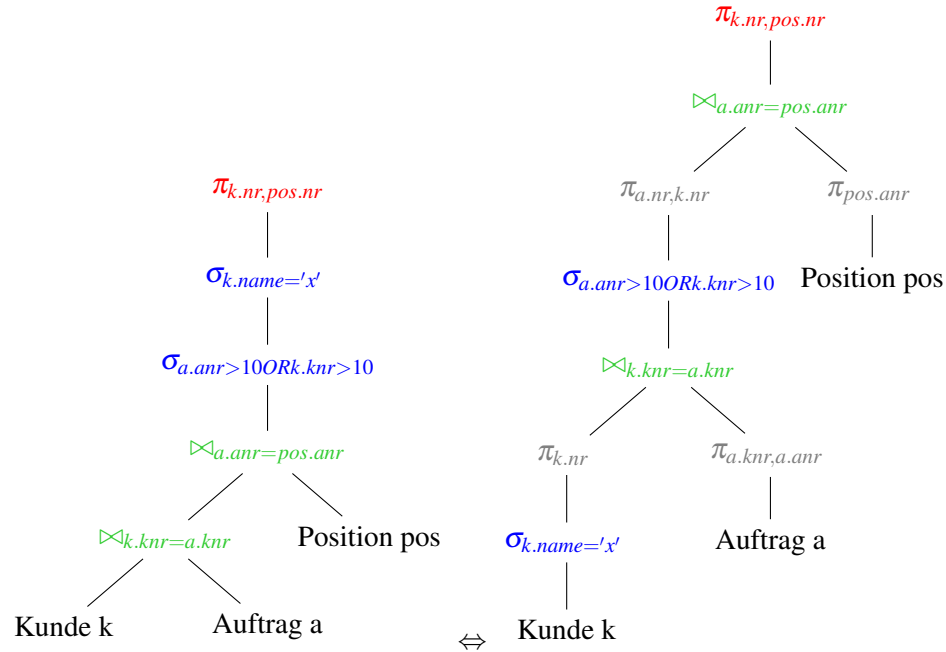
Fehlerbehandlung

Lokaler Fehler in Transaktion: Rollback der Transaktion
 Verlust des Hauptspeichers: Durch Logging der Aktionen, nach Crash Undo nicht abgeschlossener Aktionen, Redo abgeschlossener.
 Verlust des externen Speichers: Backup einspielen, inkrementell log-einspielen
 Write-Ahead-Logging: festhalten aller änderungen vor commit, bei rollback wiederherstellung aus transaktionslog;
 Log beinhaltet Undo- und Redo-Informationen; (vor und nach zustand)
 logisches Logging: protokollierung der Befehle
 physisches Logging: Zustandkopien
 Steal-noforce: Steal: gepufferte seiten können von anderer Transaktion eingelagert werden, zusammen mit zugehörigem Undo-logeintrag NoForce: committete Seiten müssen nicht sofort auf platte geschrieben, aber im Redo-log vermerkt werden.
 Ablauf: Redo-lauf (durchführen der geloggtten änderungen); Undo-lauf: rollback nicht committeter änderungen
 Recovery time objective: maximale ausfallzeit; recovery point objective: maximaler Datenverlust

Anfrageverarbeitung

Heap-Dateiorganisation: Speichern von Datensätzen in Einfügereihenfolge => unsortiert Einfügen am ende, löschen suchen und setzen eines Löschbits, suchen: sequenziell oder index;
 sequentielle-Dateiorganisation: sortierte speicherung; einfügen: suchen, anhängen und dann sortieren der Seite; löschen: suchen und löschbit setzen; suchen über index
 Index: B-Baum-Struktur; clustered Index: Segmente selbst sind sortiert; CREATE INDEX <name> ON <tabelle>(<spalte(n)>); DROP INDEX <name>
 SQL -> Query Execution Plan: Parsen der Anfrage -> Operatorenengraph; // hierbei Standardisieren und vereinfachen (KNF = (a and b) or (c and d); deMorgan: $\overline{a \wedge b} = \overline{a} \vee \overline{b}$
 SELECT k.name, pos.nr // $\pi_{k.nr, pos.nr}$
 FROM Kunde k JOIN Auftrag a ON k.knr = a.knr // $\bowtie_{k.knr=a.knr}$

JOIN Position pos ON a.anr = pos.anr // $\bowtie_{a.anr=pos.anr}$
 WHERE k.name = 'x' AND // $\sigma_{k.name='x'}$
 (a.anr > 10 OR k.knr > 10) // $\sigma_{a.anr>10 \vee k.knr>10}$
 \Leftrightarrow



Kostenschätzung

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

Kardinalität: |a|: Anzahl der gelieferten Datensätze

Selektivität: Sel(a): % der Datensätze im Vergleich zu Gesamtzahl;

NumBlocks: $\frac{|R| \cdot \text{Laenge}_{\text{Datensatz}}}{\text{Blocksize}}$ Levels(I(R,A)): Höhe des Index auf A

$$\text{Sel}(P) = \frac{|\sigma_P(R)|}{|R|}$$

Attribut = 'sth' $\Rightarrow \text{Sel}(A) = 1/|A|$ Attribut IN $\{c_1, c_2, \dots, c_n\} \Rightarrow \text{Sel}(A) = n / |A|$ A > c

$$\Rightarrow \text{Sel}(A) = \frac{A_{\max} - c}{A_{\max} - A_{\min}} \quad P_1 \text{ AND } P_2 \Rightarrow \text{Sel}(P_1) * \text{Sel}(P_2) \quad P_1 \text{ OR } P_2 \Rightarrow \text{Sel}(P_1) + \text{Sel}(P_2) + \text{Sel}(P_1 \text{ AND } P_2)$$

Plan-Operatoren

- Full-Table-Scan: durchsuche gesamte Tabelle: Cost = NumBlocks(R)
- Index-Scan: Suche anhand Index: Cost = Levels(Index) + Sel(P) * |R|
- Nested-Loop-Join: für jeden Block in einer Tabelle: durchlaufe die gesamte andere Tabelle ohne Index: Cost = NumBlocks(R) * NumBlocks(S); mit Index Cost = NumBlocks(R) * Cost(IndexScan)
- Merge-Join: Sortiere die Relationen nach Join-Attribut; wechselndes Ablaufen der sortierten Relationen; Cost: Cost(Sort(R)) + Cost(Sort(S)) + NumBlocks(S) + NumBlocks(R)
- Hash-Join: Teile kleinere Relation R in x Abschnitte, die im RAM gehalten werden können; für jeden Abschnitt: Hashen der Datensätze in R; prüfe für jeden Datensatz der größeren Relation die Join Bedingung beim entsprechenden Hascheintrag; 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 \Rightarrow einfaches Datenmodell, keine Integritätsbedingungen/Joins, schnell, gut aufteilbar, Abfragen ohne Key schwierig

Document stores

abspeichern von JSON/XML zu keys; \Rightarrow flexibles Datenmodell, ähnlich Key-Value

Column Family

Feste Column Familys (z.B. meta, posts,...) Row Key -> Column Familys -> liste von Key-Values

gute Kompression, flexibles Schema, schnelles Schreiben

Graph DB

Knoten und Kanten, beide mit Eigenschaften

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 vs asynchron.