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 => Normalisierung

Isolation: Keine Beieinflussung von Nutzern untereinander

ACID: Atomicity, Consistency,Isolation,Durability

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

Schema:

Logisches Schema: Struktur der Daten

Physisches Schema: Indizes etc,

Entity Integrity: eindeutigkeit der PKs

Referentielle Integrität: Datensätze erst löschen, wenn nicht mehr referenziert

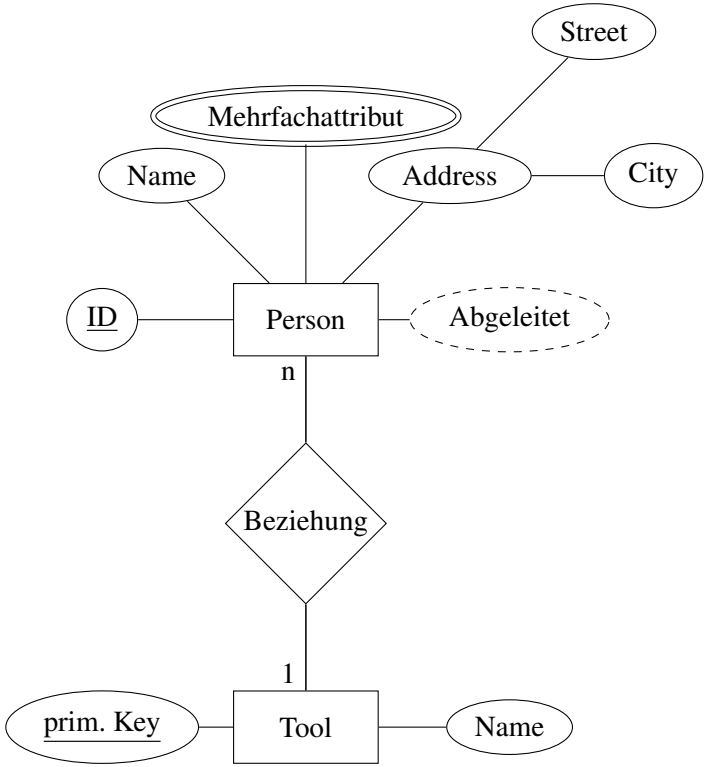
Domänen Integrität: Werte liegen in Definierten Wertebereichen (falls CHECK) und sind atomar

Surrogat-Schlüssel: künstlicher Primärschlüssel (id)

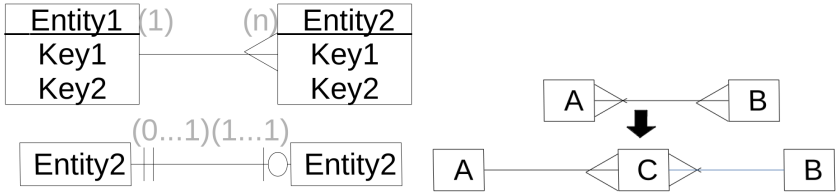
Modellierung

3Schichten-Modell: Externe Ebene(Views) - Logische Ebene(Schema) - Interne Ebene (Indizes)
=> Physische Datenunabhänigkeit: Anwendung von Änderungen auf Interner Ebene unberührt
=> Logische Datenunabhängigkeit: äuf 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



Krähenfuß



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 gehören, Einfacher Zugriff auf typunabhängige Attribute der Oberklasse, keine NULL-Werte
Nachteile: JOINs nötig => schlechtere Performance, Referentielle Integrität muss geprüft werden

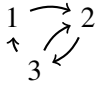
- Table-per-Concrete-Class: nur Tabellen für Klassen mit Instanzen
  Vorteile: Typen leicht unterscheidbar, Objektinformation in einer Tabelle
  Nachteile: ggf. Redundante Informationen, Anfragen über alle Objeke durch UNION

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 $\Leftrightarrow$ 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

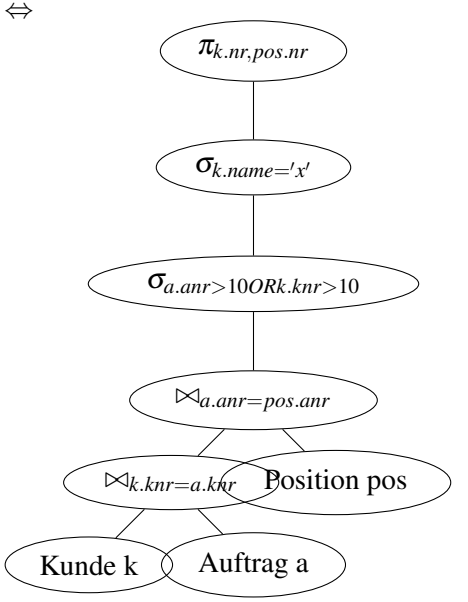
Fehlerbehandlung

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 geloggten änderungen); Undo-lauf: rollback nicht committeter änderungen
Recovery time objective: maximale ausfallzeit; recovery point objective: maximaler Datenverlust

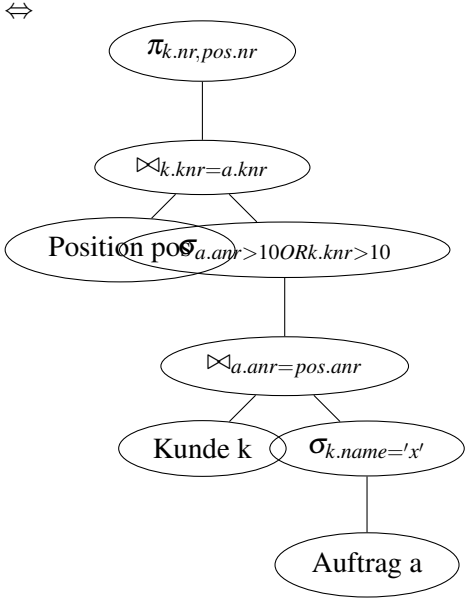
Anfrageverarbeitung

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 -> Operatorengraph; // hierbei Standardisieren und vereinfachen (KNF = (a and b) or (c and d); deMorgan: $\overline{aANDb} = \overline{a}OR\overline{b}$
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) // $\sigma_{a.anr>10ORk.knr>10}$
$\Leftrightarrow$



$\Leftrightarrow$



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: $\frac{|R| \cdot Laenge_{Datensatz}}{Blocksize}$ Levels(I(R,A)): Höhe des Index auf A
Sel(P) = $\frac{|\sigma_P(R)|}{|R|}$
Attribut = 'sth' => Sel(A) = 1/|A| Attribut IN $\{c_1, c_2, \ldots, c_n\}$ => Sel(A) = n / |A| A >c
=> Sel(A) = $\frac{A_{max} - c}{A_{max} - A_{min}}$ $P_1 ANDP_2$ => $Sel(P_1) * Sel(P_2)$ $P_1 ORP_2$ => $Sel(P_1) + Sel(P_2) +$
$Sel(P_1 ANDP_2$

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 Bedinung beim entsprechenden Hasheintrag; Cost: NumBlocks(R) + x * NumBlocks(S)