Unified Modeling Language

UML gehört zur Phase des konzeptuellen Entwurfs.
UML-Klassendiagramme unterstützen objektorientierte Konzepte

- standardisiert und ausdruckstärker
- Spezifikation von Struktur und Verhalten (Methoden)
- genauere Kardinalitätsrestriktionen (Multiplizitäten)
- Unterstützung der Abstraktionskonzepte der
- Generalisierung/Spezialisierung, Aggregation/Komposition
- UML deckt den gesamten Softwareentwurf ab

Zweck Konzeptionelle Modellierungssprache für **objektorientierte Softwaresysteme** (GPL - General-purpose language)

Ausdrucksstärke Nicht auf DB-Schemata beschränkt (viele Diagrammarten - kann Struktur und Verhalten modellieren), kann damit alles abbilden, was auch mit EER abbildbar ist und noch viel darüber hinaus (z.B. Verhalten!)

Entwicklungsstatus **Standardisiert**; de facto Modellierungssprache im Softwareentwicklungsbereich

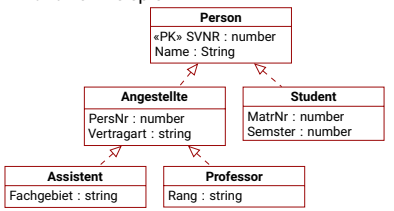
Verbreitung Große Werkzeugunterstützung

Generalisierung/Spezialisierung

Foldenge Kriterien sind wichtig um eine Entscheidung über Generalisierung/Spezialisierung zu treffen:

- Speicherverbrauch gering
- Einfacher Zugriff auf alle (abgeleiteten) Objekte einer Klasse (polymorphe Abfrage)
- Einfacher Zugriff auf alle Attribute einer Klasse
- Konsistenzsicherung
- Einfache Manipulation der Daten

Anhand von Beispiel



Basisrelationenmodell

Jede Klasse wird durch **eine Relation** abgebildet
Jede Klasse ist **genau einmal und vollständig** in ihrer Basisrelation gespeichert

Es wird eine **horizontale Partitionierung** der Instanzen erreicht

⊕ minimaler Speicherverbrauch
⊕ Zugriff auf alle Attribute einer Klasse erfordert nur eine Abfrage
😐 keine Änderungsanomalien falls Überdeckung disjunkt
⊖ Zugriff auf alle Objekte einer Klasse erfordert rekursives Suchen in Unterklassen (Nachfahren)

Ergebnis

Person

SVNR	Name
...	...

Angestellte

SVNR	Name	PersNr	Vertragsart
...	...	...	...

Student

SVNR	Name	MatrNr	Semster
...	...	...	...

Assistent

SVNR	Name	PersNr	Vertragsart	Fachgebiet
...	...	...	...	...

Professor

SVNR	Name	PersNr	Vertragsart	Rang
...	...	...	...	...

Partitionierungsmodell

Jede Klasse wird entsprechend der Attribute in der **Vererbungshierarchie zerlegt** und in Teilen in den zugehörigen Relationen gespeichert.
Es wird der Primärschlüssel dupliziert

Es wird eine **vertikale Partitionierung** der Instanzen erreicht

⊕ geringer Speicherverbrauch (PK/OID dupliziert)
⊕ inhärente Integritätssicherung
⊕ Zugriff auf alle Objekte einer Klasse erfordert eine Abfrage
⊖ hohe Such- und Aktualisierungskosten
⊖ Zugriff auf alle Attribute einer Klasse erfordert Verbundoperation (⋈) (alle Vorfahren)

Ergebnis

Person

SVNR	Name
...	...

Angestellte

SVNR	PersNr	Vertragsart
...	...	...

Student

SVNR	MatrNr	Semster
...	...	...

Assistent

SVNR	Fachgebiet
...	...

Professor

SVNR	Rang
...	...

Volle Redundanz

Jede Klasse wird durch **eine Relation** abgebildet, wobei die **geerbten Attribute dupliziert** werden (Flache Vererbungshierarchie), Primärschlüssel in Subklassen dupliziert
Instanzen werden dupliziert: besitzen nun Werte der **geerbten und neuen Attribute**.

⊕ nur eine Abfrage für Zugriff auf **alle Objekte** einer Klasse
⊕ nur eine Abfrage für Zugriff auf **alle Attribute** einer Klasse
⊖ sehr hoher Speicherverbrauch
⊖ erhöhte Gefahr von Änderungsanomalien

Ergebnis

Person

SVNR	Name
...	...

Angestellte

SVNR	Name	PersNr	Vertragsart
...	...	...	...

Student

SVNR	Name	MatrNr	Semster
...	...	...	...

Assistent

SVNR	Name	PersNr	Vertragsart	Fachgebiet
...	...	...	...	...

Professor

SVNR	Name	PersNr	Vertragsart	Rang
...	...	...	...	...

diskjunktes Einrelationenmodell

Gesamte Vererbungshierarchie wird durch eine **einzige Relation** abgebildet
Typinformation der Instanzen wird in **Typ-Attribut** gespeichert
ergibt viele NULL-Werte wenn Instanz keine Werte für Attribute der Subklassen hat

⊕ Nur eine Abfrage für Zugriff auf **alle Objekte** einer Klasse (Selektion mit Typ-Attribut)
⊕ Nur eine Abfrage für Zugriff auf **alle Attribute** einer Klasse (Selektion mit Typ-Attribut)
⊖ Einführen eines Typ-Attributes (Klassenkennzeichen)
⊖ viele Nullwerte falls Untertypen viele spezifische Attribute besitzen

vorteilhaft wenn

- Untertypen wenige spezifische Attribute besitzen ⊕ wenige Nullwerte
- Vererbungshierarchie flach ist ⊕ kleine Tabellen
- viele polymorphe Abfragen benötigt werden

Ergebnis

SVNR	Name	PersNr	Vertragsart	MatNr	Semester
...	...	...	...	...	...

Fachgebiet	Rang	PersonType
...	..	Pers., Angs., Stud.

überlappendes Einrelationenmodell

Mehrere boolesche Typ-Attribute bestimmen, zu welchen Untertypen eine Entität gehört
Vorteile: siehe disjunktes Einrelationenmodell

SVNR	Name	AngestellterTyp	PersNr	Vertragsart
...	...	true/false	...	...

Fachgebiet	Rang	Studenttyp	MatNr	Semster
...	...	true/false	...	...

Ergebnis

Abbildung ist Abhängig von Use-Case!

Sichten

Eine Sicht ist ein gespeichertes SELECT-Statement, welches lesend wie eine Tabelle genutzt werden kann.

Materialisierte Sichten

Was

Materialisierte Sicht = **Gespeicherte Tabelle**

materialisiert Eine einmal berechnete Sicht wird für die weitere Bearbeitung **permanent gespeichert**; ist eine **physische** Kopie ganzer Tabellen oder von Teilen einer Tabelle; kann wie eine **Basistabelle** verwendet werden.

Übersicht

⊕ können **Performance-Vorteile** bringen
⊕ ermöglichen **Query Rewriting** von komplexen Anfragen
⊕ sind **transparent** für den Nutzer
⊖ erfordern **mehr Speicherplatz**
⊖ verursachen Aktualisierungskosten
😐 können **inkrementell** oder mit **„Scheduler"** aktualisiert werden
😐 **Auswahl und Pflege** sind eine Herausforderung
⊖ sind **nicht standardisiert**

Wann

☑ Bei überwiegend **lesendem Zugriff**
☑ auf weitgehend **stabiler Datenbasis**
☑ Seltene Änderungen in der Datenbasis bedeuten **geringen Aufwand** bei der **Aktualisierung** der Sichten
☑ Materialisierung **reduziert Berechnungsaufwand** bei wiederkehrenden Anfrageteilen
  ✎ Datenbanksystem kann automatisch Anfrageteile erkennen, deren (Teil-)Ergebnisse durch materialisierte Sichten bereits zur Verfügung stehen (= Anfrageumschreibung - Query Rewrite)
👍 Verbesserung der Performanz (Reduktion der Antwortzeiten)
👍 Aktuelles Einsatzgebiet sind
  👆 Data Warehouse Anwendungen
  👆 Replikation

Motivation

- Viele ähnliche/gleiche Anfragen (Data Warehouse)
  - z.B. Bilanzen, Geschäftsberichte, Kennzahlenberechnung
  - **Sichten zur Vereinfachung schnellerer Zugriff auf komplexe JOINs und Aggregationen**
- Überwiegend **lesender** Zugriff auf weitgehend **stabiler** Datenbasis
  - Materialisierung der Sichten ggf. sinnvoll

Realisierung

Die materialisierte Sicht ist kein SQL Standard!

✖ Herstellerspezifische Erweiterungen von SQL (z.B. Oracle, DB2)
🔧 Durch Anlegen einer Tabelle und Einfügen der Tupel erzeugt →einfach realisierbar
🔑 **keine automatische Änderung** in der materialisierten Sicht bei Änderungen an den Basistabellen erfolgt (und umgekehrt)
  → dafür häufig ein erheblicher organisatorischer und systemtechnischer Aufwand notwendig
🕯 oft **spezielle Konstrukte zur Aktualisierung** (Snapshot oder Trigger)

Wartung

Änderungen im Basisdatenbestand erfordern:

- Neuberechnung der Sicht (Rematerialisierung) oder
- Propagierung der Änderungsoperationen zu den Sichten (inkrementelle Aktualisierung)

Zeitpunkt der Aktualisierung erfolgt

- **sofort**
  - Sichten immer aktuell
  - dafür werden die Modifikationstransaktionen behindert
- **verzögert**
  - Entkoppelung der Aktualisierung von Modifikationstransaktion; bei Zugriff auf Sicht wird diese aktualisiert
  - Sicht beim Lesen immer aktuell
  - Lesende Transaktion trägt die Aktualisierungskosten
  - Unter Umständen müssen viele Modifikationen nachgezogen werden, wenn auf die Sicht lange nicht zugegriffen wurde
- **durch Snapshot (asynchron)**
  - asynchron zur Modifikation und zum Lesezugriff nach anwendungsspezifischen Gesichtspunkten (bspw. Intervallen)

Datensicherheit

Schutz gegen **absichtliche Beschädigung** und **unbefugte Nutzung und Manipulation** durch

- Identifikation und Authentisierung (Benutzername, Passwort, ...)
- Autorisierung und Zugriffskontrolle (Rechte vergeben & prüfen)
- Auditing (Überwachung und Protokollierung von Aktionen)
- SQL-Injektion (Einschleusen von schadhaften SQL-Anweisungen)

Systemsicherheit Zugriff und Verwendung der Datenbank auf Systemebene (Benutzer/Passwort, Systemoperationen)

Datensicherheit Zugriff und Verwendung von Datenbankobjekten sowie darauf erlaubte Aktionen

Benutzer benötigen **Systemrechte**, für den Zugriff auf die Datenbank und **Objektrechte**, für den Zugriff auf Datenobjekte

Zugriffskontrolle

Besteht in SQL auf

- **Autorisierung** entweder **zentrale Vergabe** der Zugriffsrechte (Datenbankadministrator) oder **dezentrale Vergabe** der Zugriffsrechte (Eigentümer)
- **Objektgranulat** entweder **wertunabhängige** oder **wertabhängige Objektfestlegung (Sichtkonzept)**

Das Sicherheitskonzept stützt sich auf drei Annahmen:

- Fehlerfreie Benutzer-Identifikation/-Authentisierung
- Erfolgreiche Abwehr von Eindringlingen
- Schutzinformation (= Zugriffsmatrix) ist hochgradig geschützt

(no, na, ned ...)

Discretionary Access Control (DAC) **(= benutzerdefiniert)**
Eigentümer gibt Zugriffsrechte auf Objekte weiter, **Standard** bei Datenbank- und Betriebssystemen

Mandatory Access Control (MAC) **(= vorgeschrieben)**
Zugriffsrechte zentral vom System (Policy), Eigentümer kann Zugriffsrechte nicht verändern

Role-based Access Control (RBAC) **(= rollenbasiert)**
vereinfachte Administration, auch in Kombination mit DAC und MAC, Benutzer erhalten Zugriffsrechte über Rollenmitgliedschaft(en)

Systemrechte

Der Datenbankadministrator (DBA) besitzt Systemrechte auf höchster Ebene. Darf quasi alles. Kann Benutzern feingranular einzelne Rechte zugestehen.

Discretionary Access Control

Benutzer (Gruppen, Programme, Rechner, Geräte) kann nur zugreifen, wenn er die Berechtigung für die Operation (lesen, ändern, ausführen, erzeugen,...) auf das jeweilige Objekt (Relationen, Tupel, Attribute, Sichten) hat.
Die jeweiligen Zugriffsrechte werden als Matrix modelliert.

- ⊕ Weit verbeitet, in SQL realisiert
- ☺ Eigentümer (meist Ersteller) ist immer für Rechtevergabe verantwortlich
- ⊖ Schutzinformation (= Zugriffsmatrix) ist hochgradig geschützt

Role-based Access Control (seit SQL99)

Bei der rollenbasierten Zugriffskontrolle werden die Rechte nicht mehr direkt an Benutzer, sondern an Rollen geknüpft. **Rollenkonzept** vereinfacht Berechtigungsmanagement.

Auditing

Logging, alle Operationen werden Aufgezeichnet. Hilft Fehler und Sicherheitslücken zu finden. Klassisches Logging Problem: viel Müll, große Datenmengen, schwer auswertbar über die Zeit.

SQL Injection

Durch Eingabe von SQL Steuerzeichen die nicht hinreichend bereinigt wurden. Beispiel „SELECT * FROM PASWORDS WHERE Password = 'EINGABE'", Eingabe wird einfach durch String-Konkatination in das Statement eingefügt jetzt könnte der Benutzer „KA' OR '1' = '1" in das Passwortfeld eingeben und folgende Abfrage würde generiert werden SELECT * FROM PASSWORDS WHERE Password ='KA' OR '1' = '1' und es würden alle Passwörter gelistet werden.

Prozedurale Erweiterungen

Entsprechende Produkte:

- PostgreSQL: PL/pgSQL
- Informix: SPL
- IBM DB2: SQL/PSM (Persistent Stored Modules)
- MS SQL Server, Sybase: Transact-SQL
- MySQL: SQL/PSM (Persistent Stored Modules); ab Version 5
- Oracle: PL/SQL (Procedural Language/Structured Query Language)

nicht vollständig konform zum SQL-Standard

Vorteile von gespeicherten Prozeduren/Funktionen

- ⊕ Vorübersetzte Ausführungspläne (wiederverwendbar, optimiert)
- ⊕ Performance-Steigerung: Datenbank-Zugriffe der Anwendungen werden reduziert
- ⊕ Zentrale „Business Logic"
  - nutzbar von verschiedenen Anwendungsprogrammen
  - höherer Isolationsgrad der Anwendung von der Datenbank (Schichtenarchitektur)
  - Rechtevergabe für Prozeduren/Funktionen
- ⊕ Nachvollziehbarkeit von Datenmanipulationen im Fehlerfall
- ⊕ keine „externen" Transaktionen
- ⊕ Zur Integritätssicherung: Aktionsteil von Triggern

Nachteile von gespeicherten Prozeduren/Funktionen

- ⊖ DBMS wird stärker belastet
- ⊖ kompliziertere Fehlersuche (Debugging)
- ⊖ kaum Benutzerinteraktion möglich
- ⊖ zusätzliche „hybride" Sprache notwendig

PL/SQL

Besteht aus Blöcken bestehend aus:

- **Deklarationsteil** (optional)
  - Variablen/Konstanten
  - explizite Cursor
  - benutzerdefinierte Ausnahmen
- **Ausführungsteil** (erforderlich)
  - SQL-Anweisungen
  - PL/SQL-Anweisungen
- **Fehler- und Ausnahmebehandlungsteil** (optional)
  - Bearbeitung von Fehlersituationen durch „Exception Handler"

Blöcke lassen sich schachteln, bilden Gültigkeitsbereiche

Cursorkonzept

CURSOR = Iterator

Cursor/Iterator ist einer Anfrage zugeordnet und stellt der Anwendung die Zeilen der Ergebnismenge einzeln (one tuple at a time) bereit.

Cursor = Referenz auf privaten Arbeitsbereich

Impliziter Cursor

- für jede SQL-Anweisung, die nur eine einzige Zeile liefert, wird automatisch vom DB-Server erstellt und verwaltet

Expliziter Cursor

- für mengenorientierte Abfragen, die mehr als eine Zeile liefern können
- ermöglicht **zeilenweise Abarbeitung** des Abfrageergebnisses
- muss im Programm **explizit** deklariert werden

Expliziter Cursor: Vorgangsweise

1. Deklarieren eines Cursors
2. Öffnen des Cursors
   (a) veranlasst Ausführung der SQL-Anweisung im Server ohne jedoch Daten in den Programmbereich zu transferieren
3. Übertragen von Datensätzen aus der Datenbank in den Programmbereich
   (a) mit dem SQL-Befehl FETCH wird der jeweils nächste Datensatz in die spezifizierte Variable geladen
4. Schließen des Cursors
   (a) definiertes Beenden der Leseoperation und Freigeben aller Ressourcen

```
CURSOR <Cursor> [(<Param> {, <Param>})]
IS <SFW-Block> [FOR UPDATE [OF <
     Attribute>]
[NOWAIT | WAIT n]];
```

```
CURSOR c1 (name_var VARCHAR2) IS -- Datentyp
SELECT mit_name, mit_geb, mit_abtnr
FROM mitarbeiter
WHERE mit_name LIKE name_var
ORDER BY mit_name;
```

Zur Überprüfung der Cursor-Statusinformationen

%FOUND 'wahr', wenn zuletzt ausgeführte FETCH-Anweisung Datensatz gefunden hat

%NOTFOUND 'wahr', wenn zuletzt ausgeführte FETCH-Anweisung keinen Datensatz gefunden hat

%ROWCOUNT Anzahl bisher gelesener Datensätze

%ISOPEN 'wahr', wenn der Cursor geöffnet ist

```
IF NOT c1%ISOPEN THEN
OPEN c1('Huber');
END IF;
...
FETCH c1 INTO ...
IF c1%FOUND THEN ...
```

Cursor-FOR-Schleife

```
DECLARE
CURSOR emp_cursor(deptid NUMBER) IS
SELECT employee_id, last_name FROM employees
WHERE department_id = deptid;
BEGIN
FOR emp_record IN emp_cursor(90)
LOOP
DBMS_OUTPUT.PUT_LINE(
emp_record.employee_id ||
' ' ||
emp_record.last_name);
END LOOP;
END;
```

Prozeduren - Funktionen

Prozeduren und Funktionen

- Speicherung in kompilierter Form in der Datenbank
- bei Aufruf Ausführung im DB-Server **bessere Performanz**

Anonyme Blöcke

- Unbenannte PL/SQL-Blöcke

- Jedes Mal kompiliert
- Nicht in der Datenbank gespeichert
- Können nicht von anderen Anwendungen aufgerufen werden
- Geben keine Werte zurück
- Können keine Parameter annehmen

Gespeicherte Prozeduren/Funktionen

- Benannte PL/SQL-Blöcke
- Nur einmal kompiliert
- In der Datenbank gespeichert
- Werden benannt und können daher von anderen Anwendungen aufgerufen werden
- Als Funktionen bezeichnete Unterprogramme müssen Werte zurückgeben
- Können Parameter annehmen

Pakete

- kapseln Prozeduren, Funktionen, Typen, Variablen, Ausnahmen und Zugriffsprivilegien zu einem **Modul** bzw. zu einer **Bibliothek** oder Einheit zusammen
- bestehen aus zwei Teilen
  - Spezifikationsteil (öffentlich)
  - Rumpf (falls erforderlich) (privat)
- werden in der Datenbank gespeichert

Paketspezifikation

```
CREATE [OR REPLACE]
PACKAGE <Paketname> AS|IS
<glob. Deklarationen: Typen, VAR,>
<Prozedur- & Funktionsdeklarationen>
END [<Paketname>];
```

Paketrumpf

```
CREATE [OR REPLACE]
PACKAGE BODY <Paketname> AS|IS
<lokale Deklarationen: Typen, Var, >
<Prozedur- & Funktionsrümpfe>
[BEGIN
<Initialisierungsanweisungen>]
END [<Paketname>];
```

Überladen (Overloading)

- Prozeduren/Funktionen können gleichen Namen haben
- durch unterschiedliche Anzahl oder unterschiedliche Datentypen der Parameter unterscheidbar

```
CREATE PACKAGE xyz
IS
-- valid
PROCEDURE P (arg NUMBER);
-- valid (number of args)
PROCEDURE P (arg1 NUMBER, arg2 NUMBER);
-- valid (type of args)
PROCEDURE P (arg DATE);
 -- NOT valid
PROCEDURE P (arg2 NUMBER);
END;
/
```

Pakete entfernen/rekomplieren

Paket-Spezifikation und -Rumpf entfernen

```
DROP PACKAGE <PackageName>;
```

Nur Paket-Rumpf entfernen

```
DROP PACKAGE BODY <PackageName>;
```

Paket neu übersetzen

```
ALTER PACKAGE <Name> COMPILE SPECIFICATION;
ALTER PACKAGE <Name> COMPILE BODY;
ALTER PACKAGE <Name> COMPILE; -- both
```

Ausnahmebehandlung

Fehler, die zur Laufzeit auftreten, werden als Ausnahmen (Exceptions) bezeichnet

- Eine **Ausnahme** ist ein **Fehler** in PL/SQL, der bei der Ausführung ausgelöst wird.
- Eine Ausnahme kann wie folgt ausgelöst werden:
  - Implizit vom Oracle-Server (Oracle-Fehler tritt auf)
  - Explizit vom Programm
- Eine Ausnahme kann wie folgt behandelt werden:
  - Durch Abfangen mit einem "Exception Handler"
  - Durch Propagieren an die aufrufende Umgebung (z.B. SQL*Plus, PL/SQL-Programm, umschließender Block)

Dynamisches SQL

- SQL-Anweisung heißt **dynamisch**, wenn sie zur Laufzeit erstellt und dann auch ausgeführt wird.
  - Einzige Möglichkeit, DDL-Anweisungen in PL/SQL zu verarbeiten
- Früher: Built-in Packet DBMS_SQL
- Seit ORACLE 8 : Native Dynamic SQL (NDS)
  - Einfache Handhabung
  - Verwendung von :vars und USING um SQL-Injection zu verhindern (außer für Namen von Schemaobjekten)

```
EXECUTE IMMEDIATE
        'Zeichenkette' ;
EXECUTE IMMEDIATE
        'CREATE␣TABLE␣Test␣(..)';
EXECUTE IMMEDIATE
        'ALTER␣TABLE␣Test
␣␣␣␣␣␣␣␣DISABLE␣ALL␣CONSTRAINTS';
-- SELECT  Anweisungen
-- erst zur Laufzeit erstellen
```

Zusammenfassung Prozedurale SQL-Erweiterung

- vereint die Vorteile der deklarativen Abfragesprache SQL mit denen einer prozeduralen Sprache wie
  - Kontrollfluss
  - Blockstruktur
  - Prozedur, Funktion, Paket
  - Fehlerbehandlung
- ermöglicht das Speichern und Ausführen von Programmen auf dem Datenbank-Server
- ist eine proprietäre Sprache
- bildet auch die Grundlage für die Umsetzung von
  - prozeduralen Integritätsbedingungen: **Trigger**
  - objektrelationalen Konzepten

Integritätskontrolle & Trigger

Integritätsbedingungen

Integrität versus Konsistenz

Integrität korrekte, vollständige Abbildung der Miniwelt

- Integrität kann verletzt sein
- obwohl Konsistenz der DB gewahrt bleibt
- DBS kann nur die Konsistenz der Daten sichern!

Konsistenz (korrekter) DB interner Zusammenhang von

- Speicherungsstrukturen
- Zugriffspfaden und
- sonstigen Verwaltungsinformationen

Constraints Sprachkonzepte für die Sicherstellung der Konsistenz

- Wertebereiche
- Primär- & Fremdschlüssel
- Check-Klauseln

Trotzdem spricht man bei Datenbanken von **Integritätssicherung**

- also Referentielle Integrität anstatt Referentielle Konsistenz
- Constraints (Integritätsbedingungen) spezifizieren
  - akzeptable DB-Zustände
  - nicht aktuelle Zustände der Miniwelt
- Änderungen werden nur zurückgewiesen wenn sie
  - entsprechend der Integritätsbedingungen
  - als falsch erkannt werden

Arten

Semantische Integrität edeutet logische Widerspruchsfreiheit der Daten; physische Integrität wäre z.B. korrekte Implementierung der DB bzw. der Hardware

strukturelle Integritätsbedingungen Vorhandensein bestimmten von Relationen und Attributen

wertabhängige Integritätsbedingungen

operationale Integritätsbedingungen (Trigger) korrekter Übergang von einem Zustand zum nächsten

Deklarative Integritätsbedingungen

- PRIMARY KEY
- UNIQUE
- NOT NULL/NULL
- FOREIGN KEY/
- REFERENCES
- CHECK
- ASSERTION
- (DEFAULT)

geringe Reichweite, sofort, statisch, zurückweisend

Als Attribut- oder als Tabellen-Integritätsbedingung spezifiziert

Name kann vergeben werden, über den später auf die Integritätsbedingung Bezug genommen werden kann (sehr wichtig für verständliche Fehlermeldungen)

Prozedurale Integritätsbedingungen (Trigger)

- BEFORE-Statement-Trigger
- BEFORE-Row-Trigger
- AFTER-Row-Trigger
- AFTER-Statement-Trigger

größere Reichweite, verzögert, dynamisch, Korrekturmaßnahmen möglich

- Prüfung bei bestimmten Situationen oder Ereignissen
  - **Wenn** <Bedingung verletzt> **dann** <Führe Korrekturmassnahmen durch>
- Zusammenhangsregel (kausale, logische oder „beliebige" Verknüpfung) statt statischem Prädikat
  - **Wenn** Mitarbeiter.Gehalt um mehr als 10% erhöht wird, **dann** benachrichtige Top-Level-Manager
- Je mehr Integritätsbedingungen des modellierten Systems explizit repräsentiert sind, umso mehr kann das DBS „aktiv" werden!

Auslöser

- DML-Ereignis: INSERT, UPDATE, DELETE

```
CREATE [OR REPLACE]
        TRIGGER <TriggerName>
        (BEFORE | AFTER | INSTEAD OF)
        (INSERT | UPDATE [OF <Spalte>]
        | DELETE) ON <Tabelle/View>
        [<REFERENCING-Klausel>]
        [FOR EACH ROW]
        [WHEN <Bedingung>]
        <Trigger-Aktion>
END [<TriggerName>];
```

- System- oder Benutzerereignis (STARTUP, SHUTDOWN, SERVERERROR, LOGON, CREATE, ALTER, GRANT...)

```
CREATE [OR REPLACE]
        TRIGGER <TriggerName>
        (BEFORE | AFTER)
        (<Systemereignis>
        | <Benutzerereignis>)
        ON (DATABASE | SCHEMA)
        [WHEN <Bedingung>]
        <Trigger-Aktion>
END [<TriggerName>];
```

Wofür

- Automatischen Ableiten von Spalten-Werten
- Erzwingen von Integritätsbedingungen ("Implementierung" von statischen/dynamischen Integritäts- u. Konsistenzregeln)
- Auditing von Datenbankaktionen
- Transparentes Ereignis-Logging
- Statistiken über Tabellen-Zugriff
- Propagierung von Datenbankänderungen
- Synchronisieren von Tabellen-Replikaten
- Komplexe Geschäftsregeln

- Historisierung und Archivierung von Daten
- Zugriffsschutz

INSTEAD OF-Trigger: Verwendung

- Sichten für den Zugriffsschutz
- Ergänzung von NOT NULL-Spalten, die in der Sicht fehlen.
- Einschränkung der Datenmanipulation, wenn z.B. bestimmte Benutzer bestimmte Werte nur innerhalb vorgegebener Grenzen verändern dürfen.
- Transparenz von Schemaänderungen, d.h. eine Änderung in der Tabellenstruktur kann mit einer Sicht und darauf definierten INSTEAD OF-Trigger verborgen bzw. gekapselt werden.

Transaktionen

Basics

Transaktion Eine Transaktion ist eine Folge von Operationen (Aktionen), die die Datenbank von einem konsistenten Zustand in einen konsistenten, eventuell veränderten, Zustand überführt, wobei das ACID-Prinzip eingehalten werden muss.

ACID

Atomicity Atomarität oder Unteilbarkeit

- Transaktion wird ganz oder gar nicht ausgeführt
- Nach Abbruch keine Zwischenstände in der DB

Consistency Konsistenz oder Integritätserhaltung

- DB ist vor und nach Transaktion in konsistenten Zustand
- Nach Abbruch muss immer konsistenter Zustand hergestellt werden

Isolation Mehrbenutzerbetrieb

- Nutzer, der mit einer DB arbeitet, sollte den Eindruck haben, dass er mit dieser DB alleine arbeitet

Durability Dauerhaftigkeit oder Persistenz

- Nach erfolgreichem Transaktionsabschluss muss das Ergebnis dieser Transaktion „dauerhaft" in der DB gespeichert werden

SQL

begin of transaction (bot)

- kennzeichnet den Beginn einer Transaktion (in SQL implizit!)

commit

- alle Änderungen der Transaktion in der DB dauerhaft festschreiben

abort (rollback)

- bricht Transaktion ab (explizit oder implizit)
- DB muss sicherstellen, dass Datenbasis wieder in den Zustand zurückversetzt wird, der vor der Transaktion existierte

define savepoint

- definiert einen Sicherungspunkt, auf den sich die noch aktive Transaktion zurücksetzen lässt
- Änderungen bis zu diesem Sicherungspunkt werden noch nicht in die Datenbasis eingebracht (es kann noch ein abort geben)

backup transaction (rollback to savepoint)

- aktive Transaktion wird auf den jüngsten (letzten) oder den angegebenen Sicherungspunkt zurückgesetzt

Es gibt **KEINE** transaktionslosen DB-Operationen, d.h. jeder **lesende** oder **schreibende** Zugriff auf die DB kann nur innerhalb einer Transaktion stattfinden

Mehrbenutzerbetrieb

Sperrbasierte Verfahren

Sperrende (pessimistische) Verfahren stellen während des laufenden Betriebs sicher, dass der resultierende Ausführungsplan serialisierbar bleibt.

Bevor eine Transaktion eine Operation ausführen darf, muss sie eine entsprechende Sperre für das betreffende Datenbankobjekt erhalten haben.

Sperrprotokoll: = Vorschrift bzgl. des Setzens von Sperren, garantiert Konfliktserialisierbarkeit ohne zusätzliche Tests!

Zwei-Phasen-Sperrprotokoll (2PL)

Jede Transaktion durchläuft **zwei Phasen:**

- **Anforderungsphase**, in der Sperren angefordert werden, aber keine freigegeben werden dürfen
- **Freigabephase**, in der bisher erworbene Sperren freigegeben werden, aber keine neunen angefordert werden dürfen

Bei **Transaktionsende** müssen alle erworbenen Sperren zurückgegeben werden

- 2PL garantiert serialisierbaren Ablauf paralleler Transaktionen, verhindert aber weder das Phantom-Problem noch Deadlocks

Strict Two-Phase Locking

Vermeidet kaskadierendes Rücksetzen!

- Alle Sperren werden bis zum Transaktionsende gehalten damit ist kaskadierendes Rücksetzen ausgeschlossen

Conservativ Two-Phase Locking

Transaktionen werden erst dann begonnen, wenn alle Sperranforderungen beim Transaktionsbeginn erfüllt sind

- Würde Deadlocks vermeiden
  - Aber: Woher weiß man welche DB-Objekte benötigt werden?
  - Deswegen in der Praxis **kaum realisierbar**!

Nicht-sperrbasierte Verfahren

Zeitstempel-basierende Verfahren

Nicht-sperrende Verfahren versuchen, Konflikte bei der verschränkten Ausführung von Transaktionen zu erkennen und aufzulösen.

- Einfache Auflösung durch Zurücksetzen von Transaktionen
- Keine Deadlocks
- Potentiell höhere Parallelität als bei Sperrverfahren

- Bei **Konflikt**, wird Transaktion, die zu dem Konflikt geführt hat, **abgebrochen**
  - lange Transaktionen bearbeiten ihre späteren Schritte länger nach der Vergabe der Zeitmarke
  - kommen damit häufiger "zu spät" und müssen abgebrochen werden
- Zum Erkennen von Konflikten werden **Transaktionen** und **Datenobjekten Zeitstempel** zugeordnet
  - Jede Transaktion erhält als eindeutigen Zeitstempel den Zeitpunkt des Transaktionsbeginns
  - Jedem Datenobjekt O werden zwei Marken zugeordnet:
    - $tr(O)$Zeitstempel der jüngsten T, die O gelesen hat
    - $tw(O)$Zeitstempel der jüngsten T, die O geschrieben hat
- Zeistempel-Bedingungen stellen sicher, dass Abarbeitung der Transaktionen in Zeitstempel-Reihenfolge äquivalent zu einer seriellen Ausführung ist:
  - $T1$mit Zeitstempel t_1 will O lesen, also $r_1(O)$
    - $T1$darf O mit Schreibstempel $tw(O)$ nicht lesen, falls $tw(O) > t_1$, also wenn O von jüngerer/späterer Transaktion verändert wurde: **Transaktion** $T1$ **muss zurückgesetzt werden**
  - T_1mit Zeitstempel t_1 will O schreiben, also $w_1(O)$
    - T_1darf O mit Lesestempel $tr(O)$ und Schreibstempel$tw(O)$ nicht verändern, falls $tr(O) > t1$ **oder** $tw(O) > t1$ also wenn O von jüngerer/späterer Transaktion gelesen oder überschrieben wurde: **Transaktion** $T1$ **muss zurückgesetzt werden**

Serialisierungsgraph-Tester

verwalten einen Konfliktgraphen und realisieren daher eine direkte Überprüfung der Konfliktserialisierbarkeit

MVCC

Siehe kleiner Zettel

Isolation(sebenen in SQL)

READ UNCOMMITTED

- Lesen von nicht endgültig geschriebene Daten
- z.B. für statistische Transaktionen
  - ungefährer Überblick, Trendanalyse
  - nur lesender Zugriff möglich
- Keine Sperren auf Tabellenebene nötig, keine Lesesperren
  - effizient ausführbar
  - keine anderen Transaktionen werden behindert

READ COMMITTED

- Nur Lesen endgültig geschriebener Daten
- Non-Repeatable Read und Phantom- Problem möglich
- Realisierungsmöglichkeiten:
  - Lesesperre wird unmittelbar nach Lesevorgang zurück gegeben
  - Letzter gültiger Zustand des Datenwertes wird für Leseoperation aufbewahrt (Multiversion concurrency control (MVCC))
  - Zeitstempelverfahren

REPEATABLE READ

- Gelesene Daten einer Transaktion können durch andere Transaktionen nicht verändert werden
- Phantom-Problem möglich
- Realisierung
  - basiert auf einer Lese-Sperre, die bis zum Transaktionsende gehalten wird

SERIALIZABLE

- Garantiert Serialisierbarkeit einer Transaktion
- Transaktion sieht nur Änderungen, die zu Beginn der Transaktion endgültig geschrieben wurden (inkl. eigene Änderungen)
- Realisierungsmöglichkeiten
  - Lese-Sperre der gesamten Tabelle
  - Range-Locks