Transaktionsverwaltung

- TRANSAKTION: Seite 12–13 bzw. 15/31
  - → erfolgreicher Abschluss mittels commit
  - → erfolgloser Abschluss (dann Zurücksetzen)
    - => durch Benutzer/in (abort (auch rollback))
    - => durch DBMS-Fehler
- TRANSAKTIONSVERWALTUNG IN SQL: ab Seite 22 bzw. 28/31

Recovery

- SPEICHERORGANISATION: mittels Seiten (Pages) – Seite 7 bzw. 9/181
- SPEICHERVERWALTUNG (DBMS-Puffer):
  - Ersetzen von Pufferseiten: Seite 9 bzw. 12/181
    - Strategien:
      - steal: jede nicht fixierte Seite ist Kandidat für die Auslagerung
      - ¬steal: Seiten die von einer noch aktiven Transaktion verändert wurden => nicht auslagern
  - Auslagern von Pufferseiten: Seite 10 bzw. 13/181
    - Strategien:
      - force: geänderte Seiten bei Transaktionsende ausgelagert
      - ¬force: dirty pages dürfen im Puffer bleiben (Auslagern nicht erzwungen)
- EINBRINGSTRATEGIEN: Seite 12 bzw. 17/181
  - Update-in-Place:
    - direkte Einbringstrategie
    - beim Auslagern wird Seite auf diesen Platz kopiert; alter Zustand der Seite überschrieben
  - Twin-Block-Verfahren:
    - indirekte Einbringstrategie
    - für jede Seite 2 Kopien im Hintergrundsp. & Bit das angibt ob Kopie aktuell ist
    - beim Auslagern wird Seite auf aktuelle Kopie geschrieben
  - Schattenspeicherkonzept:
    - indirekte Einbringstrategie
    - nur für veränderte Seiten existieren 2 Kopien
- FEHLERKATEGORIEN: Seite 14 bzw. 21/181
  - Lokale Fehler: (nur 1 Transaktion betroffen), Fehler schnell zurücks.: (lokales Undo)
  - Fehler mit Verlust im DBMS-Puffer: durch abgeschl. Transaktionen erzeugte Änderungen erhalten (Redo) sonst zurücksetzen (Undo)
  - Fehler mit Hintergrundspeicherverlust: Wiederherstellung nur mit Archivkopie.

- AUSWIRKUNG STRATEGIE DER AUSLAGERUNG AUF RECOVERY:
  Seite 15 bzw. 28/181

	force	¬force
¬steal	- kein Redo - kein Undo	- Redo - kein Undo
steal	- kein Redo - kein Undo	- Redo - Undo

→ ¬steal + force: erzwungenes Auslagern ist teuer & es muss immer gesamte Seite gesperrt werden

→ meist steal + ¬force verwendet

- PROTOKOLLIERUNG (Logging):
  - Problem:
    - Globales Undo: Hintergrundspeicher kann Änderungen nicht abgeschlossener T. enthalten (steal)
    - Globales Redo: Änderungen abgeschlossener T. evt. noch nicht im Hintergrundspeicher (¬force)

  => Lösung: Log-Datei mit zusätzlichen Informationen
  → Beginn & Ende Transaktion
  → für Redo / Undo Informationen

  → Struktur: [LSN, TransaktionsID, PageID, Redo, Undo, PrevLSN]
  LSN = Log Sequence Number (für Redo)
  - Beispiel Log-Einträge: Seite 24 bzw. 43/181
  - WAL-Prinzip: Seite 31 bzw. 52/181
  - Compensation Log Record (CLR): für Undo
    <LSN, TransaktionsID, PageID, Redo, PrevLSN, UndoNxtLSN>
    → Undo-Informationen unnötig
  - Beispiel Log-Einträge & Wiederanlauf: Seite 42 bzw. 98/181
  - Beispiel Logging & Recovery: Seite 44 bzw. 118/181
    bzw. folgende Seiten für Recovery

    → Analyse (Winner / Loser)
    → Redo
    → Undo (nur Loser)
  - ARIES-Verfahren: Seite 48 bzw. 161/181
  - Wiederanlauf nach lokalem Fehler: Seite 50. bzw. 164/181
  - Sicherungspunkte: Seite 55 bzw. 171/181

  - <u>WAL-PRINZIP:</u>
    - bevor T festgeschrieben (in Hauptspeicher) müssen alle Log-Einträge in Datei stehen
    - bevor Seite ausgelagert → alle Log-Einträge die dazu gehören in Datei
    - Reihenfolge in Datei muss chronologisch sein

Mehrbenutzersynchronisation:

- **NEBENLÄUFIGKEIT:**
  - Vorteile: Seite 7 bzw. 10/281
  - Mögliche Fehler:
    - Lost Update: durch W-W
    - Dirty Read: durch W-R
    - Unrepeatable Read & Phantomproblem R-W
- **KLASSIFIKATION VON HISTORIEN:**
  - Definition Transaktion: Seite 17 bzw. 28/281
  - Definition Historien: Seite 19 bzw. 34/281
  - Definition Serialisierbarkeit: Seite 29 bzw. 51/281
  - Konfliktäquivalenz: Zwei Historien sind konfliktäquivalent, wenn sie sämtliche Konfliktoperationen der nicht abgebrochenen Transaktionen in der selben Reihenfolge enthalten.
  - Konfliktserialisierbarkeit: Historie ist konfliktserialisierbar, wenn sie konfliktäquivalent zu einer seriellen Historie ist.
  - Serialisierbarkeitsgraph:

$$r_1(A) \rightarrow w_1(A) \rightarrow w_1(B) \rightarrow c_1$$
$$r_2(A) \rightarrow w_2(B) \rightarrow c_2$$
$$r_3(A) \rightarrow w_3(A) \rightarrow c_3$$

$\Rightarrow$ T_1, $T_2 \rightarrow T_3$

→ Knoten: erfolgreiche Transaktionen
→ gerichtete Kanten: falls für Paar (p_i, q_j) von ~~zw. $T_i \rightarrow T_j$~~ zw. $T_i \rightarrow T_j$ von Konfliktoperationen gilt $p_i <_H q_j$

$\Rightarrow$ H konfliktserialisierbar, wenn Graph azyklisch
→ jede topologische Sortierung gibt konfl. H.

  - Eigenschaften:
    - Falls keine Transaktion abbricht: jede konfliktserialisierbare H ist serialisierbar
    - Sonst: nicht alle auch serialisierbar

- Rücksetzbare Historien:
  → eine Transaktion darf erst mit commit abschließen nachdem alle Transaktionen von denen sie gelesen hat, abgeschlossen sind
  → Leseabhängigkeiten: Seite 41 bzw. 96/281
    - $w_j(A) <_H r_i(A)$
    - $a_j \not<_H r_i(A)$ → nicht zurückgesetzt
    - Alle Schreibvorgänge auf A vor dem Lesen von T. zurückgesetzt.
  → Definition: Seite 42 bzw. 98/281
- Historien ohne kaskadierendes Rücksetzen: Seite 44 bzw. 101/281
  → Historie vermeidet kaskadierendes Rücksetzen falls T_i ~~darf~~ erst lesen von T_j, ~~darf T_i erst~~ nachdem T_j committet hat
- Strikte Historien: Seite 45 bzw. 103/281
  → Auf ein von einer Transaktion geschriebenes Datum dürfen andere Transaktionen erst nach commit oder abort zugreifen.

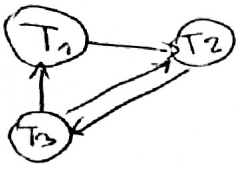
- CONCURRENCY CONTROL:
  - Sperrbasierte Synchronisation:
    - 2-Phasen-Sperrprotokoll (2PL):
      - garantiert Konfliktserialisierbarkeit aber keine Rücksetzbarkeit
      - widersprüchliche Reihenfolgen => Deadlock

Sperre		aktuell: NL	aktuell: S	aktuell: X
angefordert	S	✓	✓	–
angefordert	X	✓	–	–

→ jedes von T genutzte Objekt muss gesperrt werden
→ nur Sperren angefordert, die noch nicht besessen
→ falls Sperre nicht gewährt => warten
→ wenn eine Sperre freigegeben dürfen keine mehr angefordert werden
→ am Transaktionsende alle Sperren wieder frei

  - Strenges 2PL-Protokoll:
    → Alle Sperren werden bis zum Transaktionsende gehalten
    => nur strikte Historien erlaubt
  - Deadlock-Erkennung: Seite 62 bzw. 144
    1) Time-out-strategie: Transaktion zurücksetzen wenn keine Fortschritte in bestimmter Zeit; einfach aber Wartezeit wählen schwer

2) Wartegraph:
- Knoten = aktive Transaktionen
- Kanten = gerichtete Kante $T_i \rightarrow T_j$ falls T_i auf T_j wartet



→ Deadlock: falls Zyklus; auflösen durch Rücksetzen

- Deadlock – Vermeidung:
  1) Preclaiming (= Conservative 2PL): Seite 65 bzw. 150/281
     → Transaktionen fordern alle Sperren die sie brauchen zu Beginn an & geben alle am Schluss frei
     ⇒ Problem: benötigte Sperren zu Transaktionsbeginn meist nicht bekannt
  2) Zeitstempelbasiertes Verfahren: Seite 66 bzw. 158/281
     → jede Transaktion erhält eindeutigen Zeitstempel $TS(T_i)$
     → jüngere Transaktionen erhalten höhere Zeitstempel
     → Strategien:
       - Wound-Wait: ältere T setzen sich durch
         → T_1 will Sperre, die T_2 hat
         → T_1 älter als T_2: T_2 abbrechen
         → sonst: T_1 wartet bis Sperre wieder frei
       - Wait-Die:
         → T_1 will Sperre, die T_2 hat
         → T_1 älter als T_2: T_1 wartet
         → sonst: T_1 abbrechen

- Hierarchische Sperrgranulate: Seite 69 bzw. 165/281 – MGL
  → einheitliche Sperrgranulate manchmal ineffizient
  ⇒ Flexibilität durch unterschiedl. Granularität steigern

Sperre		aktuell				
		NL	S	X	IS	IX
angefordert	S	✓	✓	–	✓	–
	X	✓	–	–	–	–
	IS	✓	✓	–	✓	✓
	IX	✓	–	–	✓	✓

Seite 73 bzw. 172/281

- Sperrprotokoll von MGL: Seite 74 bzw. 197/281
  → Sperren anfordern: top-down
  => damit Transaktion sperren kann müssen alle Vorgänger eine Sperre halten
  → für S oder IS: IS- oder IX-Sperre
  → für X oder IX: IX
  → Sperren freigeben: bottom-up
  => Sperren erst freigeben, wenn alle Nachfolger keine Sperren mehr halten
- Sperren beim Löschen & Einfügen: Seite 81 bzw. 199/281
- Zeitstempelbasierte Synchronisation: Seite 86 bzw. 214/281
  → jede Transaktion erhält Zeitstempel ($TS(T_i) < TS(T_j)$ wenn i älter j)
  → jedem Datum A werden 2 Werte zugewiesen
  - readTS(A): Ts der jüngsten lesenden Transaktion
  - writeTS(A): Ts der jüngsten schreibenden Transaktion
  - Lesezugriff: Seite 87 bzw. 220/281
    - $TS(T_i) < writeTS(A)$: T_i älter als andere T die A geschr. => T_i zurücksetzen
    - $TS(T_i) \geq writeTS(A)$: lesen erlaubt
  - Schreibzugriff: Seite 88 bzw. 226/281
    - $TS(T_i) < readTS(A)$: jüngere T hat A bereits gelesen hätte aber Wert lesen sollen, den T_i schreiben will => T_i zurücksetzen
    - $TS(T_i) < writeTS(A)$: jüngere T hat A geschrieben & wird überschrieben => T_i zurücksetzen
    - Sonst: T_i darf schreiben

=> keine Deadlocks, konfliktserialisierbar, Reihenfolge entspricht Zeitstempeln
→ nicht rücksetzbar!
=> Erweiterung: Rücksetzbar: commit verzögern bis alle T von denen gelesen beendet
Strikt: alle Schreibvorgänge atomar am Ende Transaktion
Alternativ: nicht festgeschriebene Daten dirty markieren & Zugriff verzögern
Bsp) Seite 92 bzw. 262/281

- Optimistische Synchronisation: Seite 94 bzw. 266/281 und Folgeseiten

- Isolation levels: Seite 99 bzw. 279/281
  - Read uncommitted: kann nicht festgeschriebene Änderungen lesen, also Inkonsistenzen sehen
  - Read committed: jede Operation sieht nur Datensätze die vor Beginn der Operation committed waren.
  - Repeatable read: alle Operationen der T. sehen nur Datensätze, die vor der ersten Aktion committed waren.
  - Serializable: höchste Stufe (keine Fehler möglich)

PLpgSQL:

- USER DEFINED FUNCTIONS: Seite 17 bzw. 22/99

```
CREATE [OR REPLACE] FUNCTION
  name ([ [argname] argtype [,...]])
  [RETURNS rettype
   |RETURNS TABLE (colname coltype [,..])]
AS $$
... Code
$$ LANGUAGE
```

  - Argumente: IN, OUT, INOUT, VARIADIC (unbestimmte Menge Eingabewerte vom gleichen Typ)
  - Rückgabewert: verpflichtend; explizit mit Returns; implizit mit OUT/INOUT
    keine Werte: void
  - Beispiele: Seite 19 bzw. 26 von 99

- BLÖCKE: Seite 23 bzw. 36/99 und Folgeseiten

```
[<<label>>]          -> Blockname
[DECLARE local variables]
BEGIN
  statements
[EXCEPTIONS
  excpthandling]
END [label];
```

- VARIABLENDEKLARATION: Seite 27 bzw. 49 von 99

```
name [CONSTANT] type [NOT NULL] [{DEFAULT | = } expression];
```

  - Copying Types: Typ andere Variable oder Tabellenspalte übernehmen
    Bsp) matrnr Studenten.MatrNr%TYPE;
  - Row Types: aus mehreren Feldern zusammengesetzte Variable; auch gesamte Zeile
    Bsp) student Studenten%ROWTYPE;

- Record Types: dynamische Struktur
  Bsp) ergebnis RECORD;

- AUSDRÜCKE:
  - Befehle ohne Ergebnis: INSERT, UPDATE, ... (ohne RETURNING) normal aufrufbar
  - Befehle mit Ergebnis: ausführen Abfragen / Funktionen mit Seiteneffekten PERFORM statement
  - Befehle mit einzeiligem Ergebnis: Seite 42 bzw. 64 von 99
    SELECT expr INTO [STRICT] target FROM ...;
    STRICT: muss exakt eine Zeile zurückliefern
  - EXECUTE: führt dynamisch erstellte Befehle aus
    Seite 43 bzw. 67 von 99

- RÜCKGABEWERTE EINER FUNKTION: Seite 46 bzw. 72 von 99
  - OUT und INOUT: geben Wert an Funktionsende zurück
  - RETURN expr; beendet F. und gibt expr zurück
    für void, INOUT / OUT einfach RETURN;
  - RETURN NEXT expr;
    RETURN QUERY query; erweitert Ausgabe um entspr. Wert; beendet F. nicht; RETURN QUERY existiert Variante mit EXECUTE

- IF-THEN-ELSE: Seite 48 bzw. 74 von 99

```
IF expr. THEN
    statements
ELSIF expr. THEN
    statements
    :
ELSE statements
END IF;
```

- CASE: Seite 49 bzw. 76 von 99

```
CASE expr.
  WHEN expr. THEN statements.
  :
END CASE;
```

- SCHLEIFEN: Seite 53 bzw. 81 von 99 — Beispiel Seite 55 bzw. 84 von 99

```
[<<label>>]
WHILE Expr. LOOP
    statements
END LOOP [label];
```

```
[<<label>>]
FOR name IN expr. LOOP
    statements
END LOOP [label];
```

EXIT: bricht Schleife ab; weiter nach END LOOP
CONTINUE: bricht aktuellen Schleifendurchlauf ab; startet am Anfang vom Schleifenkörper

- FEHLERBEHANDLUNG: Seite 38 bzw. 87 von 99

```
BEGIN
  statements
EXCEPTION
  WHEN cond THEN
    handling
  :
END;
```

nicht abgefangene Fehler: werden nach außen weitergegeben → Rollback
abgefangener Fehler: letzte gute Werte bleiben erhalten, Rollback von Änderungen in Block

- CURSOR: Seite 62 bzw. 91/99 & 65 bzw. 94/99

Integritätsbedingungen:

- Einteilung in statische und dynamische Integritätsbedingungen
  - statische: Bedingungen an Zustand Datenbasis, welche in jedem Zustand erfüllt sein muss
  - dynamische: Bedingungen an Zustandsänderungen in der Datenbank
- Statische Integritätsbedingungen: unique, not null, foreign key, primary key, default
  → check Bedingungen von Werten (in Bereich, ...)
- REFERENTIELLE INTEGRITÄT: Beispiel Seite 24 bzw. 43 von 97
  → Möglichkeiten Seite 23 bzw. 41 von 97
- ZYKLISCHE ABHÄNGIGKEITEN:
  - ~~Einfügen~~ Anlegen: FOREIGN KEY CONSTRAINT erst später anlegen (ALTER TABLE)
  - Insert: Transaktionen verwenden
  - Drop: Drop foreign key constraint zuerst oder CASCADE

→ ANLEGEN: Seite 38 bzw. 75 von 97

```
create table name1 ( ref2 integer,);
create table name2 ( ref1 integer,... );
alter table name1
  add constraint constname
    foreign key (ref2) references name2;
```

→ EINFÜGEN: Seite 39 bzw. 78 von 97

```
alter table name1
  add constraint constname
    foreign key (ref2) references name2
    initially deferred deferrable;

begin;
  insert into name1 ...
  insert into name2 ...
commit;
```

→ DROP, Seite 41 bzw. 82 von 97
ALTER TABLE name1 DROP
CONSTRAINT constname;
DROP TABLE name1,

bzw. DROP TABLE name2 CASCADE;

- TRIGGER: Beispiel Seite 50 bzw. 95 von 97

CREATE TRIGGER name
BEFORE/AFTER INSERT/UPDATE/DELETE ON tabk
FOR EACH ROW / FOR EACH STATEMENT
WHEN ... (new and old verwendbar)
EXECUTE PROCEDURE function (arguments)
↑ darf keine haben

Syntax: Seite 47 bzw. 90 von 97

Funktion: Seite 48 bzw. 92 von 97
CREATE FUNCTION name() RETURNS trigger
AS $$ OLD/NEW verwendbar $$ LANGUAGE plpgsql;

SQL-VERTIEFUNG:

- GESCHACHTELTE ANFRAGEN: können unübersichtlich werden
  → mit WITH in temporäre Tabellen aufteilen:
  Beispiel Seite 7 bzw. 8 von 75
  → Rekursive Abfragen: WITH RECURSIVE: Seite 10 bzw. 12 von 75
  Bsp) Seite 15 bzw. 18 von 75
- SICHTEN: Seite 21 bzw. 25 von 75
  CREATE VIEW name AS
  SELECT row1, row2, row3
  FROM table
  → Generalisierung kann Realisierung mit Sichten haben
  Bsp) Seite 26 bzw. 32 von 75
- WINDOW FUNCTIONS: erlauben Berechnung von Funktionen über definierten Fenster
  Seite 35 bzw. 45 von 75
- SEQUENCES: Seite 44 bzw. 57 von 75; Bsp) 45 bzw. 58 von 75
  CREATE SEQUENCE name
  [INCREMENT [BY] increment]
  [MINVALUE value | NO MINVALUE] gleiches mit MAXVALUE
  [START [WITH] value] [CACHE cache]
  [[NO] CYCLE]
  Zugriff: nextval(seqName), currval(name), setval(sname, val)

- DATENTYPEN: Seite 47 bzw. 61 von 75
  CREATE TYPE name AS ENUM ('wert1', 'wert2', 'wert3');
- VORDEFINIERTE FUNKTIONEN:
  - Character-Funktionen: Seite 49 bzw. 63 von 75
  - Numerische Funktionen: Seite 50 bzw. 64 von 75
  - Konvertierungsfunktionen: Seite 51 bzw. 65 von 75
  - DATENTYP DATE: Seite 52 bzw. 66 von 75