DBS P2 TO GO

Transaktion sind ACID

- **A**tomicity: kleinste nicht zerlegbare Einheit
- **C**onstistency: Immer nur von konsistentem Zustand zu konsistentem Zustand
- **I**solation: Nebenläufige Transaktionen dürfen sich nicht beeinflussen
- **D**urability: Auswirkungen einer erfolgreich abgeschlossenen Transaktion gehen nicht mehr verloren

Strategien zur Ersetzung von Pufferseiten

- **steal**: jede nicht fixierte Seite kann ausgelagert werden
- ¬**steal:** Seiten die von einer noch aktiven Transaktion verändert dürfen nicht ausgelagert werden

Strategien zur Auslagerung von geänderten Pufferseiten (bei erfolgreichem Abschluss der Transaktion)

- **force:** Geänderte Seiten werden am Transaktionsende ausgelagert.
- ¬$\boldsymbol{force}$**:** Auslagerungen am Ende einer Transaktion wird nicht erzwungen → tw. dirty pages im Puffer

Einbringstrategien

- **Update-in-Place:** pro Seite genau ein Platz im Hintergrundspeicher, nach der beim Auslagern kopiert wird
- **Twin-Block-Verfahren:** pro Seite 2 Kopien im Hintergrundspeicher; beim Auslagern wird auf aktuelle Kopie geschrieben
- **Schattenspeicherkonzept**: nur für veränderte Seiten existieren 2 Kopien

Struktur der Log-Einträge [LSN, TransaktionsID, PageID, Redo, Undo, PrevLSN]

- **LSN**: eindeutiger Kennung des Log-Eintrags in chronologischer Reihenfolge
- **TransaktionsID:** Kennung der Transaktion die die Änderung durchgeführt hat
- **PageID:** Kennung der betroffenen Seite
- **Redo:** Information um Änderung nachzuvollziehen
- **Undo:** Information um Änderung rückgängig zu machen
- **PrevLSN:** Zeiger auf vorhergehenden Log-Eintrag dieser Transaktion

CLRs (compensation log record) $\langle LSN, TransaktionsID, PageID, Redo, PrevLSN,\ UndoNxtLSN \rangle$

- kein Undo nötig
- **PrevLSN:** Zeiger auf rückgängig gemachten Log-Eintrag
- **UndoNxtLSN:** Zeiger auf Log-Eintrag dessen Aktion als nächstes Rückgängig gemacht werden muss

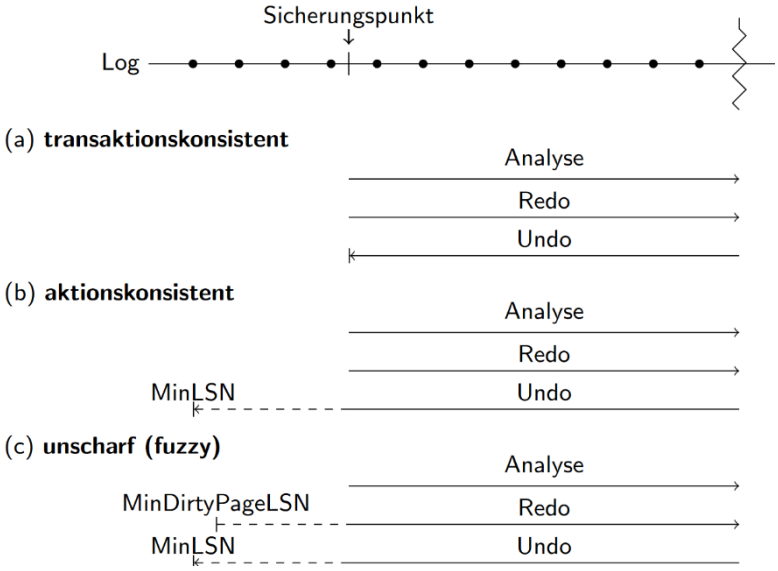
WAL-Prinzip (Write Ahead Log)

- bei *steal, ¬force & update-in-place*
- bevor eine Transaktion festgeschrieben wird, müssen alle Log-Einträge ausgeschrieben werden
- bevor eine modifizierte Seite ausgelagert wird, müssen alle zu dieser Seite gehörenden Log-Einträge ausgeschrieben werden

3 Phasen des Wiederanlaufs

1. **Analyse**
   - Identifizieren der Loser Transaktionen & deren Log-Einträgen
2. **Redo aller Änderungen** (Winner & Loser)
   - Datenbasis wird auf Stand beim Absturz gebracht
   - alle Log-Einträge werden chronologisch durchgegangen
     - LSN des Log-Eintrags > Seiten-LSN: Redo-Operation wird ausgeführt & Seiten-LSN aktualisiert
3. **Undo aller Loser-Änderungen**
   - Für jeden Log-Eintrag einer Loser-Transaktion Undo-Operation ausführen
   - CLRs erzeugen

Checkpoint (Sicherungspunkte)



- erzwingen das Ausschreiben von geänderten Seiten
- dadurch wird Log erst ab einer bestimmt LSN benötigt
- **Transaktionskonsistente Sicherungspunkte**
  - „beste“ Qualität für Recovery, sehr teuer anzulegen
  - Hintergrundspeicher soll alle Änderungen von zum Zeitpunkt Si abgeschlossene Transaktionen haben
  - kein Redo über Si hinaus nötig; Zum Zeitpunkt Si darf es keine aktiven Transaktionen geben
- **Aktionskonsistente Sicherungspunkte**
  - Undo benötigt „ältere“ Log-Einträge, erzeugt Last-Spitzen
  - alle aktiven Änderungsoperationen werden abgewartet
  - danach werden alle modifizierten Seiten ausgelagert
- **Unscharfe (Fuzzy) Sicherungspunkte**
  - Undo & redo benötigen „ältere“ Log-Einträge, kontinuierliches Ausschreiben
  - „hot-spots“ (laufende benötigte Seiten) werden lange nicht ausgeschrieben

mögliche Fehler bei unkontrollierter Nebenläufigkeit

- **Lost Update:** W-W
  - Read1, Read2, Write2, Write1
  - → T1 überschreibt Änderungen von T2; Update geht verloren
- **Dirty Read:** W-R
  - Read1, Write1, Read2, Write2, Read1, Abort1
  - Lesen nicht freigegebener (committed) Änderungen
  - Änderungen von T2 basieren auf inkonsistentem DB-Zustand
- **Unrepeatable Read:** R-W
  - Read1, Read2, Write2, Commit2, Read1
  - Wiederholtes Lesen durch T1 liefert unterschiedliche Ergebnisse
- **Phantomproblem:** R-W
  - T1: Select SUM(y) from X; T2: Insert INTO X; T1: Select SUM(y) from X;
  - T1 berechnet unterschiedliche Werte da “Phantom” eingefügt wurde

Konfliktoperationen

- paare von Operationen, die auf dasselbe Datenobjekt zugreifen; mind. 1 davon schreibend

Transaktion besteht aus einer Menge an Operationen sowie einer (partiellen) Ordnung $<_i$ auf den Operationen

Historien (Schedules) mit Ordnung $<_H$

- zeitliche Anordnung der elementaren Operationen einer Menge von Transaktionen
- $<_H$ ist verträglich mit allen $<_i$
- für Konfliktoperationen p, q gilt entweder $p <_H q$ oder $q <_H p$

Serielle Historie: jede Transaktion wird vollständig abgearbeitet bevor die nächste beginnt

Serialisierbare Historie: (verzahnte) Historie mit selbem Effekt wie irgendeine serielle Ausführung

konfliktäquivalent: 2 Historien sind *konfliktäquivalent,* wenn sie sämtliche Konfliktoperationen (der nicht abgebrochenen Transaktionen) in der selben Reihenfolge ausführen

konfliktserialisierbare Historien sind konfliktäquivalent zu einer seriellen Historie

Serialisierbarkeitsgraph SG(H) einer Historie H

- Knoten: erfolgreiche Transaktionen T1, T2, .., von H
- Gerichtete Kante $T_i \rightarrow T_j$ falls für (mind.) ein Paar (p_i, q_j) von Konfliktoperationen gilt $p_i <_H q_j$

Eine Historie H ist genau dann *konfliktserialisierbar,* wenn der zugehörige Serialiserbarkeitsgraph SG(H) azyklisch ist.

Jede *topologische Sortierung* von SG(H) gibt eine *konfliktäquivalente serielle Historie* an

Schreib/Leseabhhängigkeiten zwischen Historien

- T_i liest von T_j in einer Historie H falls
  - $w_j(A) <_H r_i(A)$: T_j schreibt einen Wert den T_i liest
  - $a_j \not<_H r_i(A)$: T_j wird nicht zurückgesetzt bevor T_i gelesen hat
  - $Wenn\ ein\ w_k(A)\ mit\ w_j(A) <_H w_k(A) <_H r_i(A) existiert, dann\ auch\ a_k <_H r_i(A)$
    - alle zwischenzeitlichen Schreibvorgänge anderer Transaktionen auf A werden vor dem Lesen von T_i zurückgesetzt

rücksetzbare Historien: eine Historie H heißt *rücksetzbar*, wenn T_i von T_j liest, dann darf T_i nicht vor T_j commiten

Strikte Historie: Auf ein von einer Transaktion geschriebenes Datum dürfen andere Transaktionen erst nach deren Beendigung schreibend oder lesend zugreifen.

Zwei Sperrmodi

- S Shared, read lock, Lesesperre
- X eXclusive, write lock, Schreibsperre

Deadlock-Vermeidung

- **Preclaiming**: Transaktionen werden nur gestartet, wenn alle benötigten Sperren am Beginn verfügbar sind
- **Zeitsetempelverfahren:** Zeitstempel pro Transaktion, der entscheidet ob eine TA auf eine Sperre wartet

Sperrbasierte Synchronisationsprotokolle

- **Zwei-Phasen-Sperrprotokoll (2PL)**
  - nachdem eine Transaktion eine Sperre freigegeben hat darf sie keine weiteren Sperren anfordern
  - Wachstumsphase
    - Transaktion darf Sperren anfordern, aber keine Sperren freigeben
  - Schrumpfungsphase
    - Transaktion darf erworbene Sperren freigeben, aber keine weiteren mehr anfordern
  - garantiert Konfliktserialiserbarkeit aber keine Rücksetzbarkeit
- **Strenges Zwei-Phasen-Sperrprotokoll (strict 2PL)**
  - wie 2PL aber alle Sperren werden bis zum Transaktionsende gehalten
  - lässt nur strikte Historien zu
- **Zwei-Phasen-Sperrprotokoll mit Preclaiming** (conservative 2PL)
  - Variante von strict 2Pl
  - Transaktion fordert alle Sperren, die sie brauchen wird bereits zu Beginn, und hält alle bis zum Schluss
- **Sperrprotokolle mit Zeitstempel**
  - jede Transaktion erhält eine eindeutigen Zeitstempel (jüngere einen größeren)
  - 2 mögliche Strategien falls T1 eine Sperre erwerben will, die von T2 gehalten wird
  - **wound-wait** Strategie (ältere Transaktionen haben Vorrang)
    - falls T1 älter als T2: T2 wird abgebrochen
    - falls T1 jünger als T2: T1 wartet auf Freigabe der Sperren durch T2
  - **wait-die** Strategie (jüngere Transaktionen sind „schüchtern“)
    - falls T1 älter als T2: T1 wartet auf die Freigabe der Sperre
    - falls T1 jünger als T2: T1 wird abgebrochen

- **Hierarchische Speergranulate (MGL)**
  - Sperren auf verschiedenen Ebenen (DB, Segment, Pages …)
  - zusätzlicher Sperrmodi (IS, IX)
    - zeigen an, dass weiter unten in der Hierarchie bestimmte Sperren existieren
Sperre		aktuell				
		NL	S	X	IS	IX
angefordert	S	✓	✓	–	✓	–
	X	✓	–	–	–	–
	IS	✓	✓	–	✓	✓
	IX	✓	–	–	✓	✓
  - Anforderung benötigter Sperren erfolgt **top-down**
    - bevor eine Transaktion einen Knoten mit S oder IS sperren kann, benötigt sie für alle Vorgänger in der Hierarchie eine IS- oder IX- Sperre.
  - Freigabe von Sperren erfolgt **bottom-up**
    - Eine IS- oder IX-Sperre an einem Knoten darf erst freigegeben werden, wenn alle Sperren auf Nachfolgerknoten bereits freigegeben sind

Isolations Levels in SQL

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
`READ UNCOMMITTED`	möglich	möglich	möglich
`READ COMMITTED`	–	möglich	möglich
`REPEATABLE READ`	–	–	möglich
`SERIALIZABLE`	–	–	–

Protokoll	Äquivalente serielle Historie	Weitere Eigenschaften	Deadlock möglich
2PL	Reihenfolge der Sperranforderungen bei Konflikten	im allgemeinen nicht rücksetzbar	ja
Strict 2PL	wie 2PL	strikt	ja
Strict 2PL + Deadlock-Vermeidung	wie 2PL	strikt	nein
Zeitstempel-basierend	Zeitpunkt von BOT	strikte Variante existiert	nein
optimistisch	Zeitpunkt der Validierung	strikt	nein

SQL User Defined Functions

```
CREATE FUNCTION nsum (IN a integer,
  IN b integer, OUT c integer) AS $$
   BEGIN
    c = a + b;
  END; $$ LANGUAGE plpgsql;
```

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

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
[ EXCEPTION
  excpthandling ]
END [ label ];
```

```
FOR s IN SELECT * FROM Studenten LOOP
 INSERT INTO hoeren VALUES (s.MatrNr,184686);
END LOOP;
```

Statische Integritätsbedingungen

- Schlüsselkandidaten: unique (Attributeliste)
- Primärschlüssel: primary key (Attributeliste)
- Fremdschlüssel foreign key (Attributeliste)
- Keine Nullwerte: not null
- Default wert angeben: default

```
create table Studenten
       (MatrNr    integer primary key,
        Name      varchar(30) not null,
        Semester  integer default 1
          check (Semester between 1 and 13));
```

Check

- Überprüfung allgemeiner statischer Integritätsbedingungen;
- Auswertung bei Update & Insert (nur möglich wenn check != false)

Constraints

- Spalten- oder Tabellen-Contraints; Teil der Tabellendefinition (CREATE TABLE) oder später (ALTER TABLE ADD)
- können benannt & entfernt werden (ALTER TABLE DROP)

Referentielle Integrität

- *Fremdschlüssel* müssen entweder auf existierende Tupel einer anderen Relation verweisen oder einen Nullwert enthalten
- **Dangling References:** Verweise von Fremdschlüsseln auf nichtexistierende Datensätze
- Einhaltung der Referenziellen Integrität in SQL
    - on update bzw. on delete
        - no action
        - cascade
            - on delete cascade: kaskadierendes Löschen kann Problematisch sein
        - set null
        - set default

Zyklische Abhängigkeiten

- DEFEREABLE: Constraint wird am Transaktionsende überprüft
- NOT DEFERRABLE: Constraint muss immer sofort überprüft warden
- INITIALLY DEFERRED, INITIALLY IMMEDIATE: Default-Einstellung für Transaktion kann in TA geändert werden

Constraint Löschen

- ALTER TABLE t1 DROP CONSTRAINT
- DROP TABLE t1 CASCADE;

```
alter table chicken
    add constraint chickenrefegg
        foreign key (eID) references egg
        initially deferred deferrable;

begin;
insert into chicken values(1,11);
insert into egg values(11,1);
commit;
```

Trigger

Definition	create trigger	CREATE TRIGGER test1
Auslösende Ereignisse	insert, update, delete	AFTER INSERT ON table1
Zeitpunkt	before/after (insert)	
auf Tupel- bzw. Quersebene	for each row/statement	FOR EACH ROW
Einschränkung	when	WHEN NEW.x > 6
Pre- & Current-Image	*old* bzw. *new*	

Beispiel mit PostgreSQL

```
CREATE OR REPLACE FUNCTION ouf()
   RETURNS TRIGGER AS $$
BEGIN
 IF (OLD.Rang = 'C3' AND NEW.Rang='C2') THEN
  RETURN OLD;
 END IF;
 RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER keineDegradierung
BEFORE UPDATE ON ProfesorIn
FOR EACH ROW WHEN (OLD.Rang is not NULL)
EXECUTE PROCEDURE ouf();
```

Zerlegung komplexer Anfragen mit WITH

- Bsp.: WITH test3 AS (select x,y from z)

```
WITH avgNote AS (
  SELECT MatrNr, avg(Note) AS dsN
  FROM pruefen
  GROUP BY MatrNr ),
 bestAvgNote AS (
  SELECT MatrNr, dsN
  FROM avgNote
  WHERE dsn = (SELECT max(dsN)
               FROM avgNote))
SELECT MatrNr, dsN
FROM bestAvgNote
```

Rekursive Abfragen

```
WITH RECURSIVE tabName(AttrListe) AS (
    Nicht rekursiver Abschnitt
  UNION [ ALL ]
    Rekursiver Abschnitt
    )
SELECT ...
```

```
WITH RECURSIVE voraus(v) AS (
    SELECT vorgnr
    FROM voraussetzen
    WHERE nachfnr = 5216
  UNION
    SELECT voraussetzen.vorgnr
    FROM voraus, voraussetzen
    WHERE voraus.v=voraussetzen.nachfnr
)
SELECT v FROM voraus;
```

- *working table*
  - Ergebnis des nicht rekursiven Abschnitts
  - Ergebnis des rekursiven Abschnitts (wird in jeder Iteration ersetzt)
  - dient als Input für Selbstreferenz im rekursiven Teil
- *intermediate table*
  - enthält Ergebnis der aktuellen Iteration
- *Ergebnis/ result table*
  - Ergebnis des nicht rekursiven Abschnitts & von jedem Rekursiven

Ablauf der Rekursiven Abfragen

1. nicht rekursiven Abschnitt auswerten; (distinct bei UNION); Ergebnis *in working table* & *result table* kopieren
2. so lange working table != leer
   1. rekursiven Abschnitt auswerten; *working table* als Input; speichert Ergebnis in intermediate table
   2. bei UNION: alle Duplikate & Tupel die schon in *result table* sind entfernen
   3. Tupel aus *intermeidate table* an *result table* anfügen & Inhalt aus *working table* durch *intermediate table* ersetzen
3. nicht rekursives select (am Ende) auf *result table* anwenden

Views

- Realisierung von Inklusion & Vererbung mittels Views
- Update auf Sichten ist eingeschränkt möglich

```
create view prüfenSicht as
select MatrNr, VorlNr, PersNr
from prüfen;
```

Sequence

- Zahlenfolge, kein Wert wird doppelt vergeben;

```
CREATE SEQUENCE name
[INCREMENT [ BY ] increment]
[MINVALUE minvalue | NO MINVALUE]
[MAXVALUE maxvalue | NO MAXVALUE]
[START [WITH] start] [CACHE cache]
[[NO] CYCLE]
```

```
CREATE SEQUENCE persnr_seq
 START WITH 2200 INCREMENT BY 5
MINVALUE 2200 MAXVALUE 15000
 NO CYCLE;
```