Transações (ACID)

Atomic: ou corre ou não corre.
Consistency preservation: DB de estado consistente para outro estado consistente.
Isolation: Ts concorrentes não devem interferir.
Durability: o resultado de uma T válida deve ser tornado persistente (mesmo com falhas).

Escalonamentos

Cascadeless: não contém dirty reads.
Recuperável: admite dirty reads se a T que fez overwrite terminar antes da outra.
Estrito: não tem dirty reads nem dirty writes.
Equivalência: escalonamentos são equivalentes se a ordem de suas operações conflituantes for a mesma.
Série: as opers de cada T são consecutivas.
Serializável: ⇔ a um escalonamento serie.

Anomalias

1. **Dirty write**: E ñ é estrito ainda que cascadeless Lost update; overwriting uncommited data <w(t1, x1), w(t2, x1) ...>
2. **Dirty read**: E ñ cascadeless <w(t1, x1), r(t2, x1) ... >
3. **Non-repetable read**: E ñ serializavel, ainda que cascadeless e recuperável <r(t1, x1), w(t2, x1), ...>
4. **Phantom read**: <r(t1, P), w(t2 in P) ...>

ISOLATION LEVEL	1	2	3	4
READ UNCOM	✓	x	x	x
READ COMMITTED	✓	✓	x	x
REPEATABLE READ	✓	✓	✓	x
SERIALIZABLE	✓	✓	✓	✓

✓: don't occur x: may occur

2 Phase lock (2PL)

read: shared lock; **write**: exclusive lock

	Unlock	Shared	Exclusive
Unlock	✓	✓	✓
Shared	✓	✓	X
Exclusive	✓	X	X

Ação bem formada: lock -> action -> unlock
Ação de 2fases: lock -> action -> ... unlock(commit)

Isolation level	Leutura
Read Uncommited	ñ e bem formada
Read Committed	bem formada
Repeatable Read	▪ ▪ and two fases
Serializable	▪ ▪ and ▪ ▪ + predicate locking

Escritas são sempre bem formadas e de 2 fases
Row-level locks: forçam o uso de locks de 2 fases nas instruções

Solicita/detem	1	2	3	4
for key share (1)	✓	✓	✓	X
for [illegible] share (2)	✓	✓	X	X
for no key update (3)	✓	X	X	X
For update (4)	X	X	X	X

Declarar variáveis

```
DECLARE
  nome [CONSTANT] type [NOT NULL]
[{DEFAULT |:= |=} expression]
```

Afetar

```
nome {:= |=} expression;
SELECT expression INTO nome FROM
{INSERT|UPDATE|DELETE} … RETURNING
expression INTO nome;
```

Condição

```
IF boolean-expresion THEN
  statements;
ELSEIF boolean-expresion THEN …
ELSE …
END IF;
```

```
CASE search-expresion
  WHEN expression [, expression, …]
THEN
    Statements;
[WHEN expression [, expression, …]
THEN
    Statements;
[ELSE
  Statement]
END CASE;
```

Loops

```
[<<lable>>] LOOP
  Statements;
END LOOP [<<lable>>];

WHILE boolean-expresion LOOP
  statements;
END LOOP [<<lable>>];

FOR name IN [reverse] expression [by
expression] LOOP
  statements;
END LOOP [<<lable>>];

FOR target IN query LOOP;
  statements;
END LOOP;
```

Procedimentos armazenados

```
CREATE OR REPLACE PROCEDURE name(args)
  LANGUAGE plpgsql
AS $$
DECLARE
  …
BEGIN
  …
END; $$;
CALL nome(args);
```

Funções

```
CREATE OR REPLACE FUNCTION name(args)
  RETURN {type | TABLE(i INT, …)}
  LANGUAGE plpgsql
AS $$
DECLARE
  …
BEGIN
  …
END; $$;
{SELECT |PERFORM} query;
```

Vistas

```
CREATE OR REPLACE VIEW name{[collums]}
  AS query …;
```

Gatilhos

```
CREATE OR REPLACE TRIGGER nome
{BEFORE| AFTER| INSTEAD OF} {INSERT|
UPDATE [OF columns]| DELETE|TRUNCATE}
ON table
FOR {EACH {ROW| STATEMENT}}
EXECUTE FUNCTION name();
```

When	Row-level	Statement-level
Before	Tables	tables and views
After	Tables	tables and views
Instead	Views .	----

Mapper

Realiza operações CRUD associados a entidade

```
public interface IMAPPER<T> {
  void create (T entity);
  T read (T entity);
  void update (T entity);
  void delete (T entity);
}
```

Repositoy

Operações mais complexas, sobre conjunto de entidades.

```
public interface IRepo<T, TK> {
  T findByKey(TK key);
}
```

UnitOfWork

Gere o ciclo de vida das entidades JPA.

JPA

```
EntityManagerFactory emf =
Presistence.createEntityManagerFactory
("nome");
EntityManager em =
emf.createEntityManager();
EntityTransaction emt =
em.getTransaction();
em.clear();
em.close();
em.flush();
em.commit();
em.presist(entity);
em.find(target.class, key, lock)
em.merge(entity);
em.remove(entity);
em.contains(entity);
emt.isActive(); -> emt.rollback();
```

Anotações

```
@Entity
@Table(name = "")
@Id
@GenerateValue(statagy =
GenerationType.Identity)
@Embedable
@EmbeddedId
@Column (name= "")
@JoinColumn(name = "targetProp",
referencedColumnName = "srcProp")
@OnetoOne(fetch = Fetchtype.Lazy,
[mappedby = "propName"])
@OnetoMany([mappedBy = "propName"],
cascade = , [orphanremoval={true|
false}])
@ManytoOne(fetch = , [mappedBy =
"propName"])
@ManytoMany([cascade=
CascadeType.{REMOVE| PRESIST| MERGE|
REFRESH}], [mappedBy = "propName"])
@JoinTable(name = "targetTable",
joinColumns = {@JoinColumn(), …}
inverseJoinColumns = {@JoinColumn(),
…})
@NamedQuery(name = "", query = "")
@NamedStoredProcedureQuery(name = "",
procedureName = "",resultSetMappings =
"namedResult", parameters = {
@StoredProcedurePArameter(mode =
ParameterMode.[IN|OUT|INOUT], type =
primitive.class, … })
@SqlResultSetMapping(name="
namedResult",
classes={
@ConstructorResult(targetClass=dst.cla
ss,columns={@ColumnResult(name="column
Name", type= targert.class), …})
```

Queries

```
em.createQuery("", resultClass);
q.[getSingleResult()|getResultList()]
em.createNamedStoredProcedure(name(?1
…)
sp.setParameter(1, a.getId());
sp.execute();
q.executeUpdate();
```

Transações

- Fornecem mecanismos de recuperação em caso de falha do sistema
- Facilita o tratamento de erros ao nível aplicacional
- Fornecem mecanismos de controlo de interferências entre aplicações a concorrer no mesmo acesso a dados

Propriedades ACID

- Atomicidade - ou é executada totalmente ou não (em caso de erro por ex.)
- Consistência - deixa a DB num estado consistente
- Isolamento - transações concorrentes não devem interferir umas com as outras na sua execução
- Durabilidade - o resultado de uma transação deve ser tornado persistente

Tipos de Ações

- Não protegidas - efeitos não necessitam de ser anulados
- Protegidas* - efeito pode e tem de ser anulado se a transação falhar
- Reais - efeito não pode ser anulado

* só o objeto de tratamento transacional

Escalonamento

- Sendo um conjunto de transações $\{T_1, \ldots, T_n\}$
- É uma ordenação S das operações de cada T_i de modo a q todas as ações de T_i apareçam por ordem.

ex: $T_1 = <r(x_1), w(x_2)>$
$T_2 = <r(x_2)>$

✓ $S_1 = <(r(t_1, x_1), r(t_2, x_2), w(t_1, x_2)>$
✗ $<w(t_1, x_2), r(t_2, x_2), r(t_1, x_1)>$

Conflito

Duas operações num escalonamento conflituam se:

- são de transações diferentes
- ambas acedem aos mesmos dados
- pelo menos uma é de escrita

Dirty read - uma transação lê um item escrito por outra ainda não terminada

Dirty write - uma transação escreve por cima de um item escrito por outra não terminada

Escalonamento cascadeless

- não contém dirty reads
- não exibe o efeito de cascading abort ou cascading rollback

Escalonamento Recuperável

- Não existe nenhuma transação q faça commit tendo lido um item escrito por outra ainda não terminada

Escalonamento Estrito

- Não tem dirty reads nem dirty writes

Escalonamento Série

- Todas as transações são executadas sem interposição de operações de outras

N transações => N! escalonamentos série

Equivalência (do ponto de vista do conflito)

- Dois escalonamentos são equivalentes do ponto de vista do conflito se a ordem das duas operações conflituantes for a mesma nos dois escalonamentos

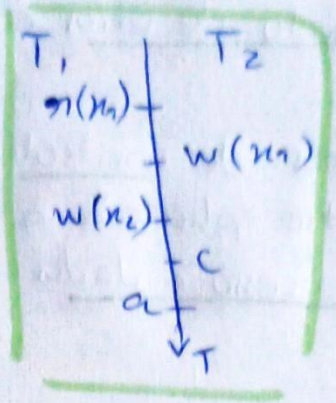
Escalonamento Serializável

- Se for equivalente do ponto de vista do conflito a um dos escalonamentos série

Nota: existem outras formas de serializabilidade, ver slide 14

Representação de Escalonamentos

$S = \langle r(t_1, x_a), w(t_2, x_a), w(t_1, x_c), c(t_2), a(t_1) \rangle$



ou

Nota:
a - abort
c - commit
r - read
w - write

Anomalias

Dirty writes (w/w)

- overwriting uncommited data
- também conhecido como lost update

Dirty reads (w/r)

- uncommited dependency
- leitura de dados escritos por uma transação ainda não terminada

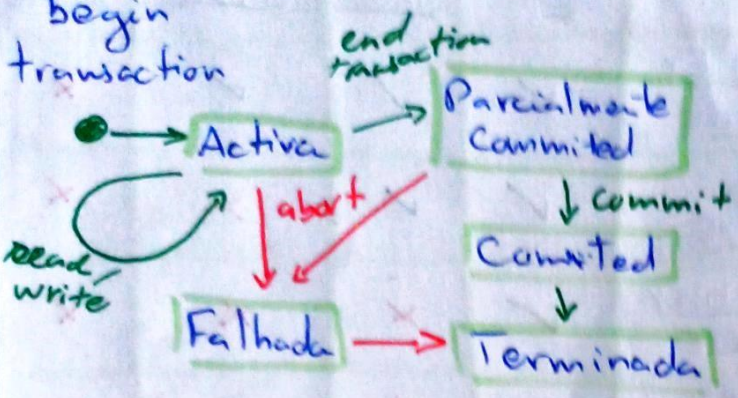
Non-repeatable read (r/w)

- tbm conhecido como fuzzy read
- durante uma transação ocorrem dois reads do mesmo registo com valores diferentes
- quando uma transação escreve num registo lido por outra ainda não terminada

Phantom tuples (r/w)

- ocorre quando uma transação escreve com um predicado num registo já lido por outra com esse predicado

Estados das transações



- **Ativa** - estado após o início da transação, onde ocorrem reads e writes
- **Parcialmente commited** - quando se indica q a transação deve terminar com sucesso.
- **Commited** - quando é escrito o resultado do commit no log
- **Falhada** - quando é abortada ou quando os testes realizados no estado "parcialmente commited" falha (escreve abort no log)
- **Terminada** - a transação deixa de existir no sistema

Níveis de isolamento

- definem como uma transação é isolada das outras

Isolation Level	Dirty Writes	Dirty Reads	Fuzzy Reads	Phantom Reads
READ UNCOMMITED	✗	✓	✓	✓
READ COMMITED	✗	✗	✓	✓
REPEATABLE READ	✗	✗	✗	✓
SERIALIZABLE	✗	✗	✗	✗

✓ - possível
✗ - não possível

- quanto maior o nível, mais os problemas de performance

Two-Phase Locking (2PL)

- protocolo de controlo de concorrência de transações
- matriz de compatibilidade:

	Unlock	Shared	Exclusive
Unlock	✓	✓	✓
Shared	✓	✓	✗
Exclusive	✓	✗	✗

✓ - compatível
✗ - não compatível

Ação bem formada - protegida por um par lock/unlock

Ação de duas fases - não executa unlock antes de locks de outras ações da mesma transação (só no fim da transação)

Nota: uma ação pode ser bem formada e de 2 fases simultaneamente

Predicate locking

↳ em vez de "lockar" registos, "locka" predicados

Chaos

↳ não possível na norma ISO SQL
↳ apresenta dirty writes

Deadlocks

- ocorre quando 2 ou mais transações dão lock ao mesmo registo
- uma das transações tem de abortar
- em PSQL, o mecanismo de deteção de deadlock só é ativado quando o tempo de espera para um deadlock é expirado (deadlock_timeout)
- Staruation - quando uma transação tem de esperar por tempo indeterminado

- Podem ser usadas cláusulas na instrução select q utilizam locks de duas fases em reads
- Todas as escritas são de duas fases e colocam um lock

- Exclusive locks: FOR UPDATE, FOR NO KEY UPDATE
- Shared locks: FOR SHARE, FOR KEY SHARE

- Matriz de compatibilidade:

	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE	✓	✓	✓	×
FOR SHARE	✓	✓	×	×
FOR NO KEY UPDATE	✓	×	×	×
FOR UPDATE	×	×	×	×

✓ - compatível
× - não compatível

Graus de Isolamento

- Grau 0 - chaos
- Grau 1 - read uncommited
- Grau 2 - read commited
- Grau 3 - repeatable read + serializable

Controlo de Concorrência

- Timestamps
↳ a cada transação T é associada uma time stamp de quando foi criada -> ts(T)
↳ cada item x tem um par de ts das últimas transações que leram e escreveram -> $(t_w(x), t_r(x))$:

```
read(T, x):
  if (tw(x) ≤ ts(T)) &
      read
      tr(x) = Max(tr(x), ts(T))
  else
    abort T
```

continua →

```
write (T, x):
 if tw(x) ≤ ts(T) AND tr(x) ≤ ts(T):
    write
    tw(x) = ts(T)
 else
    abort T
```

- Versões
  - para cada item X são mantidas várias versões $x_1, \ldots, x_n$, cada uma com as timestamps de escrita e leitura:

```
read(T, x):
 while (tw(xi) > ts(T)):
    i--
 read
 tr(xi) = Max(tr(xi), ts(T))
```

```
write(T, x):
 while (tw(xi) > ts(T)):
    i--
 if tr(xi) > ts(T):
    abort
 else:
    if ts(T) = tw(t): sobrepor versão
    else:
       write, inserindo nova versão k:
       tw(xk) = tr(xk) = ts(T)
```

  - Problemas:
    - espaço gasto em memória
    - escalonamentos não recuperáveis e não cascadeless
    - escalonamentos não escritos

- Snapshot
  - variante do protocolo multi-versão
  - cada item tem várias versões validadas, tendo cada uma o ts da sua criação
  - quando uma linha é commited, é criada uma versão q tem acesso às anteriores

```
read(T, x):
 while tw(xi) > ts(T):
    i--
 read
```

  - se uma transação tentar atualizar um item com versão mais recente, é abortada
  - Variantes:
    - First committer wins
    - First updater wins