Database Systemer, Forår 2006

IT Universitet i København

Lecture 10: Transaction processing

6 april, 2006

Forelæser: Esben Rune Hansen

Today's lecture

Part I: Transaction processing

- Serializability and atomicity
- ACID properties and rollbacks
- Dirty reads and isolation levels in SQL

Problem Session on Constraints (5 min)

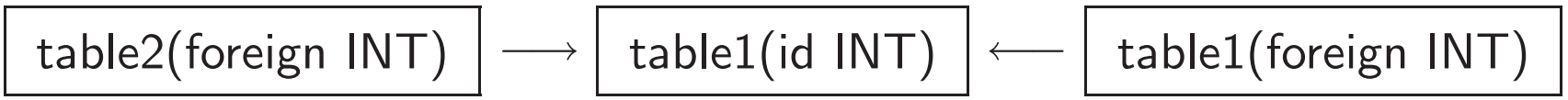
Suppose that you have created table2 by:

```
CREATE TABLE table2(foreign INT)
```

Write down the SQL-statements needed if we want to:

1. Create a table `table1(id)` with `id` as a primary index
2. Make `table2.foreign` be a foreign key on `table1.id`
3. Create a table `table3` with a foreign key foreign on `table1.id`. If a key in `table3` references to a primary key in `table1` that is deleted you want the foreign key in `table2` to be deleted as well.

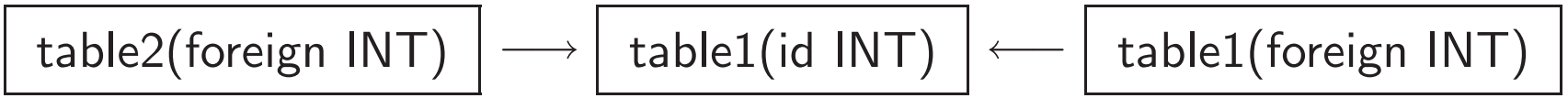
Illustration:



Solutions to Problem Session

1. Create a table `table1(id)` with `id` as a primary index:
   CREATE TABLE table1(id INT PRIMARY KEY);
2. Make `table2.foreign` be a foreign key on `table1.id`
   ALTER TABLE table2 ADD CONSTRAINT EsbensConstraint FOREIGN KEY (foreign) REFERENCES table1(id);
3. Create a table `table3` with a foreign key foreign on `table1.id`. If a key in `table3` references to a primary key in `table1` that is deleted you want the foreign key in `table2` to be deleted as well.
   CREATE TABLE table3(foreign REFERENCES table1(id) ON DELETE CASCADE);

Illustration:



Databases and concurrency

In most large database systems, many users and application programs will be (and *must* be) accessing the database at the same time.

Consecutive updates to a database that "belong together" can grouped into **transactions** by the database programmer.

Having **concurrent** users updating the database raises a number of problems that, if not properly dealt with by the DBMS, could leave the database in an inconsistent state, even if all users "did the right thing".

Serializability

We would like the DBMS to make transactions satisfy **serializability**:

> Even though many transactions may be performed at the same time, the state of the database should look *as if* transactions were performed one by one (i.e., in a **serial schedule**).

The DBMS is allowed to *choose* the order of transactions:

- It is not necessarily the transaction that is started first, which is first in the serial schedule.
- The order may look different from the viewpoint of different users (but at the end, they will see the same state).

Atomicity

A similar issue is what happens if a transaction, for some reason (e.g., power or hardware failure) is aborted while executing.

We would like the DBMS to make transactions satisfy **atomicity**:

> Even though a transaction may involve many updates, the state of the database should look *as if* either the *whole* transaction or *no part* of the transaction has been carried out.

Serializability and atomicity in SQL

As a default, the SQL standard requires the DBMS to process transactions atomically.

As a default, SQL also executes transactions in a serializable manner. (This is not the Oracle default – more on this later.)

However, in some situations the SQL programmer might give the DBMS permission to execute transactions in a non-serializable manner. (More on this later.)

Atomicity and constraints

Some kinds of constraints require several updates to be performed in an atomic manner.

Example (The "Chicken-and-egg" problem): Suppose we have foreign key constraints stating that

- Every value of attribute A should also be a value of attribute B.
- Every value of attribute B should also be a value of attribute A.

Then values for A and B must be inserted and deleted *simultaneously*:

- The constraints should be declared `DEFERRABLE`.
- The transaction inserting or deleting should begin by setting the constraints as `DEFERRED`, so that they are only checked at the end of the transaction.

Problem Session (5 min)

How can we make a "chicken-and-egg" constraint?

This will solution will not do:

```
CREATE TABLE Chicken (ChickenId INT PRIMARY KEY REFERENCES egg(EggId));
CREATE TABLE Egg(EggId INT PRIMARY KEY REFERENCES chicken(ChickenId));
```

Consider the following:

- What is wrong with the two statements above?
- How can we make a chicken-and-egg constraint (difficult)
- Write down the SQL-commands that will create two tables with a chicken-and-egg constraint (very difficult)

Illustration:

Solution to Problem Session

This will solution will not do as the table Chicken need the existence of the table Egg in order to be created and vice versa:

```
CREATE TABLE Chicken (ChickenId INT PRIMARY KEY REFERENCES egg(EggId));
CREATE TABLE Egg(EggId INT PRIMARY KEY REFERENCES chicken(ChickenId));
```

Instead we:

1. Create both tables without the foreign key constraints:
   - **CREATE TABLE Chicken (ChickenId INT PRIMARY KEY);**
   - **CREATE TABLE Egg(EggId INT PRIMARY KEY);**
2. Add the foreign key constraints to the tables:
   - **ALTER TABLE Chicken ADD CONSTRAINT ChickenConstraint FOREIGN KEY (ChickenId) REFERENCES Egg(EggId) INITIALLY DEFERRED DEFERRABLE;**
   - **ALTER TABLE Egg ADD CONSTRAINT EggConstraint FOREIGN KEY (EggId) REFERENCES Chicken(ChickenId) INITIALLY DEFERRED DEFERRABLE;**

Next: Transactions in SQL and ACID properties

Transactions in SQL

By the SQL standard, transactions consisting of more than one statement are started by the command

```
START TRANSACTION
```

and ended with one of the following commands:

- `COMMIT`: Update the state of the database to reflect the changes performed in the transaction.
- `ROLLBACK`: Discard the transaction.

Transactions in SQL*Plus

In SQL*Plus the `START TRANSACTION` command is *implicit*, i.e., statements are by default grouped into transactions.

Transactions are ended when:

- You exit from SQL*Plus.
- A command alters the database schema.
- A `COMMIT` or `ROLLBACK` statement appears.

The ACID properties of transactions

Ideal transactions are said to meet the **ACID** test:

- **A**tomicity – the all-or-nothing execution of transactions.
- **C**onsistency – transactions preserve database constraints.
- **I**solation – the appearance that transactions are executed one by one.
- **D**urability – the effect of a transaction is never lost once it has completed.

A good DBMSs should fully implement **A**, **C** and **D**, and will allow the user to specify the extent to which **I** should hold (for efficiency reasons).

However, **I** always applies to any *single* SQL statement in a transaction.

Durability

We already discussed all ACID properties except **durability**, which is another reason why DBMSs are used for critical applications:

> A good DBMS is able to withstand a power outage, disk or hardware failure, with *little or no loss of data*, returning the database to a recent, *consistent* state.

Next: Dirty reads/writes and isolation levels in SQL

Dirty reads

Suppose that a transaction is in the process of updating a large relation.

Other transactions executing may see:

- Old data, not (yet) modified by the transaction.
- New data, updated by the transaction.

As long as a transaction has not committed, all data that it has changed is referred to as **dirty**.

A read of dirty data (called a **dirty read**) is a main source of trouble when executing transactions concurrently.

Dirty writes

Overwriting data that is not yet committed can also cause problems.

Example: Two concurrent bank transfers. One from account A to account B, and one in the other direction.

Any good DBMS prevents such *dirty writes*.

Dirty reads/writes and performance

The default in SQL is to execute transactions in a serializable way.

In particular, dirty reads cannot performed by the DBMS.

This can give performance problems:

- Sometimes a transaction must wait for other (potentially lengthy) transactions to commit before being able to finish.

Read-only transactions

We can tell the DBMS that the current transaction does not update the database, using the SQL statement

```
SET TRANSACTION READ ONLY;
```

It is a good idea to do this whenever we have a read-only transaction:

- Other transactions never need to wait for a read-only transaction to finish, since it does not change anything.
- Thus the DBMS potentially runs faster if it knows that a transaction is read-only.

Read committed

The lowest isolation level in which SQL allows transactions to do updates is `READ COMMITTED` (Oracle's default isolation level).

SQL statements of transactions running at this isolation level make no dirty reads or writes. In particular, they see other transactions as atomic, in the sense that either:

- No changes made by a transaction are seen, or
- all changes made by a transaction are seen.

However, different statements in the transaction may see the database in different states (i.e., some transactions may commit in between).

This is sometimes referred to as the **phantom phenomenon**.

Problem session (5 minutes)

In the same setting as before, assume that transactions run at isolation level `READ COMMITTED`. Consider the following sequence of statements.

A	B
`INSERT INTO Primes VALUES (2);`	
	`SELECT * FROM A.Primes;`
	`INSERT INTO A.Primes VALUES (2003);`
	`SELECT * FROM A.Primes;`
`COMMIT;`	
	`SELECT * FROM A.Primes;`
`SELECT * FROM Primes;`	
	`COMMIT;`
`SELECT * FROM Primes;`	

What output is possible for each `SELECT` statement?

The SERIALIZABLE isolation level

The highest isolation level in SQL is SERIALIZABLE, which makes sure that transactions always execute in a serial schedule.

A slightly weaker guarantee, ANOMALY SERIALIZABLE, gives the following guarantee in addition to that of READ COMMITTED:

- No value read or written by the transaction is changed before the transaction is committed.
- In particular, there is no phantom phenomenon.

Oracle defines the SERIALIZABLE isolation level, but the actual behavior is not always serializable (see exercises today). It seems to be at least ANOMALY SERIALIZABLE, though.

SQL isolation levels

The SQL standard defines four isolation levels in total:

- `SERIALIZABLE` (not really implemented in Oracle). Ideal, serializable transactions.
- `REPEATABLE READ`. Allows the result of a query to change during the transaction, in the sense that more tuples may be added.
- `READ COMMITTED` (Oracle default). The result of any statement reflects some set of committed transactions.
- `READ UNCOMMITTED`. Allow dirty reads. (Not recommended!)

Nonstandard isolation levels

Several DBMSs (DB2, SQL Server) have isolation levels that are not in the SQL standard:

- `CURSOR STABILITY`. Like `READ COMMITTED`, but additionally prevents "lost updates" (consider, for example, two transactions that simultaneously try to increase the balance of a bank account).
- `SNAPSHOT ISOLATION`. Transactions are guaranteed to only see data as it is "at the time they are started".

A thorough description of these and the SQL isolation levels is provided in the paper *A Critique of ANSI SQL Isolation Levels*, found in the course schedule.

Implementation of isolation levels

The most common implementation of isolation is using *locks*:

- A transaction reading a database element must first obtain a *read lock* on the element.
- A transaction writing to some database element must hold a *write lock* on this element, which can only be held if no other transaction has a read or write lock on the element.

If a transaction cannot obtain some needed lock, it waits for the lock to become available.

The various isolation levels differ mainly in the way in which they request and release locks. Higher levels keep locks for longer. (More details in the course *Advanced Database Technology*.)

System-generated rollbacks

Sometimes the DBMS must roll back one or more transactions to allow others to go on.

Example: Suppose that transaction A cannot commit before B has committed, and vice versa (e.g. if A waits to obtain a lock held by B, and B waits to obtain a lock held by A). Then we have a **deadlock**, and the only way out is to roll back one of the transactions.

Deadlock situations occur more often at higher isolation levels, which is another reason to use the lowest isolation level necessary.

Most important points in this lecture

As a minimum, you should after this week:

- Know and understand the ACID properties of transactions.
- Know how to create transactions in SQL.
- Be able to predict possible transaction behavior at isolation levels `SERIALIZABLE` and `READ COMMITTED`.

Course Related Projects

I perioden for 4-ugers projekter vejleder jeg følgende:

1. En Web-baseret browser for relationer:
   http://www.itu.dk/people/pagh/DBS06/project.html
2. En dynamisk hjemmeside ved hjælp af SQL og PHP/Servlets
3. Dit ynglingsprojekt der involverer SQL og PHP/Servlets

Alle projekter skal udføres enten ved hjælp af PHP eller Servlets.

Snak med mig efter denne forelæsning hvis du er interesseret. Eller skriv til esben@itu.dk