

---

Introduction to Databases, Fall 2003  
IT University of Copenhagen

**Lecture 10, part II: Transaction processing**

November 4, 2003

Lecturer: Rasmus Pagh

---

# — Today's lecture - part II —

---

## Transaction processing

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

## — What you should remember from previously —

In this lecture I will assume that you remember:

- That one or more updates in a database can be grouped into something called a **transaction** (we looked at this in connection with constraints).

---

**Next: Serializability and atomicity**

---

## — 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.

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”.

[Figure 8.22 shown on slide]

## — Serializability

---

There is a problem with the code of [Figure 8.22] if two transactions are performed simultaneously.

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.

# Atomicity

---

A similar issue is what happens if a transaction, for some reason (e.g., power or hardware failure) is aborted while executing. [Figure 8.24]

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.

We already saw the atomicity property in connection with database constraints: A transaction violating a constraint was **rolled back**.

## — 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:** 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*:

- In some systems, this can be done by grouping the updates together in a transaction (as stated in GUW).
- In other systems (e.g. Oracle) the transaction must explicitly remove the constraint, and invoke it again at the end of the transaction.

---

**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.

(NB! This is different from a “generic interface”, as described in G UW.)

Transactions are ended when:

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

## — Problem session (5 minutes) —

Suppose that user A has an empty relation Primes (p INT).

Then user A and B issue the below statements, in this order.

A	B
<pre>SELECT * FROM Primes;  INSERT INTO Primes VALUES (2);  SELECT * FROM Primes;  COMMIT;</pre>	<pre>SELECT * FROM A.Primes;  INSERT INTO A.Primes VALUES (2003);  SELECT * FROM A.Primes;  SELECT * FROM A.Primes;  COMMIT;  SELECT * FROM Primes;</pre>

Assuming serializable transactions, determine the output of each SELECT.

## — 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.

Commercial DBMSs tend to fully implement **C** and **D**, while **A** and **I** are only partially implemented (for efficiency reasons).

# 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 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 reads, serializability, and performance —

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

In particular, dirty reads cannot not 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.

## — **Allowing dirty reads** —

---

We may allow a transaction to perform dirty reads by the SQL command:

```
SET TRANSACTION READ ONLY ISOLATION LEVEL READ UNCOMMITTED;
```

The SQL standard only allows this for read-only transactions.

The term **isolation level** refers to the extent in which the transaction appears to execute in isolation.

## — Read committed —

---

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

Transactions running at this isolation level 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
<pre>INSERT INTO Primes VALUES (2);  COMMIT;  SELECT * FROM Primes;  SELECT * FROM Primes;</pre>	<pre>SELECT * FROM A.Primes; INSERT INTO A.Primes VALUES (2003); SELECT * FROM A.Primes;  SELECT * FROM A.Primes;  COMMIT;</pre>

What is the output generated by each `SELECT` statement?

## — The SERIALIZABLE isolation level —

The highest isolation level in SQL is SERIALIZABLE, which gives nearly, but not quite, the behavior of a serial schedule of transactions.

This isolation level gives the following additional guarantee:

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

It is possible, but not easy, to come up with transactions that, when executed at the SERIALIZABLE isolation level, may give a result different from any serial schedule.

## — SQL isolation levels —

---

The SQL standard defines four isolation levels, two of which are supported by Oracle.

- **SERIALIZABLE** (implemented in Oracle). Almost, but not quite, 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.

## — 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 (i.e., we have a **deadlock**). Then 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`.

## — Next lecture

---

Next time we will look at the basics of relational database efficiency:

- Indexes.
- Update versus query performance.
- Database tuning.