# Information Systems

WS 2005, JKU Linz
Course 7: On-Line Transaction Processing

Gábor Bodnár

URL: `http://www.risc.uni-linz.ac.at/education/courses/ws2005/is/`

## Overview

- Security issues
- Transactions
- Locking
- ACID requirements.

# Discretionary Security Control

A discretionary control scheme consists of *authorities*: named security clauses that grant certain privileges to certain users on certain tables (or views).

GRANT { privilege [, privilege]* | ALL [ PRIVILEGES ] }
    ON [ TABLE ] tablename [, tablename]*
    TO user [, user]* [ WITH GRANT OPTION ]

where

privilege = { SELECT | INSERT | UPDATE | DELETE |
    REFERENCES | TRIGGER }
user = { username | GROUP groupname | PUBLIC }

# Discretionary Security Control

REVOKE [ GRANT OPTION FOR ]
{ privilege [, privilege]* | ALL [ PRIVILEGES ] }
ON [ TABLE ] tablename [, tablename]*
FROM user [, user]* [ CASCADE | RESTRICT ]

Warning: A user has the "sum" of privileges granted to him directly or via groups.

Privileges of users on view depend on the privileges on the underlying tables.

If the creator of a view loses the SELECT privilege on any of the underlying tables, the view is dropped.

# Mandatory Security Control

Data objects have *classification levels*, users have *clearance levels* (both coming from the same ordered set).

User at clearance level $i$ can retrieve data at classification levels $j \leq i$.

User at clearance level $i$ can modify data only at classification level $i$. (A user can change to smaller clearance levels).

## Statistical Security

Databases created for statistical purposes should allow agglomerative queries to be performed, but they should forbid to obtain (or even to deduce) information about individual rows.

# Cryptography: the RSA Algorithm

Key generation:

- Generate two random primes $p, q$ of approximately the same bit-length such that $n = pq$ is of the required length (e.g. 1024 bits).
- Set $h = (p-1)(q-1)$, choose an integer $0 < e < h$ such that $\gcd(e, h) = 1$
- Compute $0 < d < h$ such that $ed \equiv 1 \mod h$.

The public key is the pair $(n, e)$ and the private key is $(n, d)$; the values $p, q, h$ must also be kept secret.

# Cryptography: the RSA Algorithm

Encryption:

- Get the public key $(n, e)$ of the recipient.
- Let the message be $m$.
- The encrypted message corresponding to $m$ is: $c = m^e \mod n$.

Decryption:

- Let the encrypted message be $c$.
- Get the secret key $(n, d)$.
- The corresponding decrypted message is: $m = c^d \mod n$

# Recovery and Concurrency

Goals:

- To be able to recover from system crashes and faulty database modifications without data and consistency loss.

  (Power failure, disk fill-ups, network connection problems, etc.)

- To maintain the consistency of the database in concurrent use.

  (Schedule database modification requests and avoid deadlocks.)

From the conceptual level we only see a mechanism (of the DML) that allows user defined sequences of DB modifications, called *transactions*, to transfer the database from one consistent state to another in *one logical step*.

# Atomicity

The recovery is solved by using *log* (or *journal*) files, into which the DBMS archives states of the database for each transaction.

The actual implementations can be highly complex, especially when concurrency is supported.

The "all or nothing" property of the transactions, or in other terms the fact that either the whole transaction commits or everything is rolled back to the initial state, is also called the *atomicity* property.

A typical application program is then consists of a sequence of transactions.

# Transactions

The logical step (or unit) of modifications is called a *transaction*.

In DBMSes it is usually start with the keyword

BEGIN TRANSACTION

and ends with one of the keywords

COMMIT
ROLLBACK.

Inside the transaction unit a sequence of DDL and/or DML statements can be placed.

Transactions cannot be nested.

# Transaction Management

At transaction start:

- The *transaction manager* of the DBMS "saves" the current state of the database.

In the transaction block:

- If an error occurs, a ROLLBACK is requested (unsuccessful termination). (Implicit ROLLBACK)
- If the block terminates successfully a COMMIT is requested (if the COMMIT is executed with no error, the modifications are permanently recorded in the database).

# Example

```
/* cid has to be the pointer to the course ID converted to string,
   returns 0 on success 1 on error, SQL_OK_Code is expected to be defined. */
int cancelCourse(char *cid){
   char query1[50]; char query2[50];
   strcpy(query1, "DELETE FROM Attendance WHERE cid = ");
   strcpy(query2, "DELETE FROM Course WHERE cid = ");
   strcat(query1, cid); strcat(query2, cid);
   EXEC SQL BEGIN TRANSACTION;
   EXEC SQL PREPARE q1 FROM :query1;
   EXEC SQL EXECUTE q1;
   if(SQLSTATE != SQL_OK_Code){ EXEC SQL ROLLBACK; return 1; }
   EXEC SQL PREPARE q2 FROM :query2;
   EXEC SQL EXECUTE q2;
   if(SQLSTATE != SQL_OK_Code){ EXEC SQL ROLLBACK; return 1; }
   EXEC SQL COMMIT;
   if(SQLSTATE != SQL_OK_Code){ return 1; }else{ return 0; }}
```

# On-Line Transaction Processing (OLTP)

Properties:

- It is mostly concerned with processing small amount of data per transaction.
- It is often used in time-constrained environment (e.g. real time systems).

The goal is to resolve any interference between concurrent transactions.

In other words: To guarantee that a correct database gets transferred into a correct one by an execution of individually correct transactions concurrently.

# Interference: Lost Update

The transactions read and update the same row of a table concurrently.

| Time | Transaction 1 | Transaction 2 |
|---|---|---|
| $t1$ | retrieve $r$ | |
| $t2$ | | retrieve $r$ |
| $t3$ | update $r$ | |
| $t4$ | | update $r$ |

At time $t4$ the update made by Transaction 1 is lost.

# Interference: Uncommitted Dependency

Left: TA1 works on false assumptions.

| Time | TA 1 | TA 2 |
|---|---|---|
| $t1$ | | update $r$ |
| $t2$ | retrieve $r$ | |
| $t3$ | | rollback |

| Time | TA 1 | TA 2 |
|---|---|---|
| $t1$ | | update $r$ |
| $t2$ | update $r$ | |
| $t3$ | | rollback |

Right: After the rollback the update of TA1 is lost.

# Interference: Inconsistent Analysis

The result computed in Transaction 1 is wrong because it retrieved the old value of $r_1$ and the new value of $r_3$.

| Time | Transaction 1 | Transaction 2 |
|---|---|---|
| $t1$ | retrieve $r_1$ | |
| $t2$ | aggregate $r_1$ | |
| $t3$ | | update $r_3$ |
| $t4$ | retrieve $r_2$ | |
| $t5$ | aggregate $r_2$ | |
| $t6$ | | update $r_1$ |
| $t7$ | retrieve $r_3$ | |
| $t8$ | aggregate $r_3$ | |

# Locking

The interference problems can be handled by locking, which denies access to a row/set of rows/table for other transactions until the transaction which holds the lock finishes its modifications.

Types of locks:

**Exclusive lock** (write lock, X-lock) Requests for any other lock on the object must wait till the active exclusive lock is released.

**Shared lock** (read lock, S-lock) Requests for other shared locks on the objects are immediately granted, exclusive locks must wait until the object is released.

# Locking Protocol

For a transaction

- to retrieve a row in a table of the database first requires to get an S-lock on it.
- to modify a row (alter a table) in the database first requires to get an X-lock on it (if it already has an S-lock it must "upgrade" it to an X-lock).

Locking requests that cannot be immediately granted get into a waiting queue.

"Overlapping" S-locks can make X-lock requests wait for too long: this phenomenon is called *starvation*.

# Deadlocks

This is the situation when two or more transactions wait on each other to release a lock.

| Time | Transaction 1 | Transaction 2 |
|---|---|---|
| $t1$ | acquire S-lock on $r$ | |
| $t2$ | retrieve $r$ | |
| $t3$ | | acquire S-lock on $r$ |
| $t4$ | | retrieve $r$ |
| $t5$ | X-lock request on $r$ | |
| $t6$ | wait | X-lock request on $r$ |
| $t7$ | wait ... | wait ... |

# Deadlock Resolution

To predict and avoid deadlocks can be difficult and time consuming (finding circles in graphs).

In practice if a transaction does not make any progress within a given time limit it is assumed to be deadlocked.

If there are presumably deadlocked transaction, the transaction manager chooses a “victim” and requests an implicit rollback on it.

When this happens all the locks the “victim” held get released.

# Serializability

A (possibly interleaved) execution of a set of transactions is called as a *schedule*.

A schedule of a set of transactions, which are individually correct by assumption, is called *serializable* if and only if it produces the same result as some serial execution of the same set of transactions; and this does not depend on the (a priori correct) initial state we start from.

Serializable schedules transfer the database from a correct state into a correct state.

Serializability can be difficult to test for an arbitrary schedule.

# Two-Phase Locking Protocol

The *two-phase locking protocol* requires that

- a transaction must acquire a lock (of the appropriate kind) on a tuple before it operates with it
- after releasing a lock, the transaction must never acquire a lock again.

In other terms, the transaction must acquire locks and it is expected to separate the lock acquisition and the lock release phases.

**Theorem.** If all transactions obey the two-phase locking protocol, then all possible schedules are serializable.

# Uncommitted Dependency and Locking

| Time | Transaction 1 | Transaction 2 |
|---|---|---|
| $t1$ | | acquire X-lock on $r$ |
| $t2$ | | update $r$ |
| $t3$ | | release X-lock on $r$ |
| $t4$ | acquire S-lock on $r$ | |
| $t5$ | retrieve $r$ | |
| $t6$ | release S-lock on $r$ | |
| $t7$ | commit | |
| $t8$ | | rollback |

To achieve recoverability, we require that a transaction which uses data modified by another transaction must not commit before the other terminates (and if that rolls back, the first must be rolled back too). Such a schedule is called *cascade-free*.

# The ACID Requirements

**Atomicity** Results of transactions are either all committed or rolled back ("all or nothing" principle).

**Consistency** (or Correctness in the more rigorous sense) The database is transformed by a transaction from a valid state into another valid state. (Consistency, however need not be preserved at intermediate points of the transactions.)

**Isolation** (or serializable and non-cascaded in the more rigorous sense) The result of a transaction is invisible for other transactions until the transaction commits.

**Durability** Once a transaction has committed, its results are permanently recorded in the database.

# Summary

- Discretionary and mandatory security control
- RSA algorithm
- Recovering consistency using transactions
- Interferences in OLTP: lost update, uncommitted dependency, inconsistent analysis
- Locking protocol, deadlock
- Serializability, two phase locking
- ACID requirements