

# Transactions

Ferd van Odenhoven  
Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Bedrijfsmanagement

26 augustus 2009

## Some loose ends about databases in particular and concurrency in general.

A databasedesigner has good knowledge of the following definitions:

- Atomicity
- Transactions
- A.C.I.D.
- 2 Phase commit

Most of these properties are dealt with (as implemented) by the database vendor, so we take them for granted at the database level<sup>1</sup>. Normally, a database course mentions these concepts, but we forgot 😞.

---

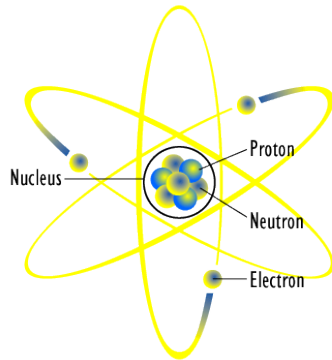
<sup>1</sup>Although this may not always hold on all popular databases in all implementations.

# Atomicity

The greek word “Atom” means indivisible: there is no smaller part.

In this context we use the word atomicity to denote “All or Nothing”. That is if we describe an action or activity to be *atomic* it either succeeds (completely) or fails (completely).

Failure in this case: As if nothing had happened.



# Transactions in applications

In many applications you would like an action to succeed, but cannot always rely on this fact.

- 1 There maybe interference from other programs or parts of program
- 2 Certain (pre) conditions may not hold
- 3 There may be a technical malfunction like network or disk failure
- 4 Not all of these aspects can always be tested in advance

In single user systems and applications, some of the above problems are less likely to happen, but in multiuser and server (database) applications, the first and second point are less easy to hold.

## Typical examples

Transaction processing is quite common in daily life.

- Bank money transfer; Two accounts are involved, both the credit to the receiving account and the debit to the paying account must succeed
- Booking a flight reservation (including payment); You either want a seat and pay or you want neither.
- Source code repository commit (subversion); You want all your changes to be committed before any other commit (or checkout) can start. This keeps the repository in a *consistent* state.

## Example bank transaction

Asume two accounts A, nr 1234567 and B, nr 7654321. A transfers 100 € to B. The payment is described by a description 'D'. Then the sql statements could look like this:

— In this example 10000 Cents are transfered from A to B.

```
BEGIN WORK;  
SELECT nextval() AS transid FROM transaction_sequence;  
INSERT INTO transaction_log  
    (transid ,from_account ,to_account ,description)  
    VALUES ($transid ,123456 ,7654321 , 'D');  
UPDATE account_balance SET balance = balance -10000 WHERE  
    account_id=1234567;  
UPDATE account_balance SET balance = balance +10000 WHERE  
    account_id=7654321;  
COMMIT;
```

Note that this is not syntactically correct for a known sql dialect. There should be some statement in the application language (or embedded database language) that picks up the sequence number \$transid and puts it into the insert statement.

# Acidity of a database

Modern and complete databases have 4 properties that are commonly summarised in the acronym A.C.I.D.



# A C I D

- A Atomicity refers to the ability of the DBMS to guarantee that either all of the tasks of a transaction are performed or none of them are.
- C Consistency refers to the database being in a legal state when the transaction begins and after it ends.
- I Isolation refers to the ability of the application to make operations in a transaction appear isolated from all other operations.
- D Durability refers to the guarantee that once the user has been notified of success, the transaction will persist, and not be undone.

## Implications of ACID

These properties do not fall from the sky. They require quite some effort of the database implementors.

- Atomicity involves locking out other parties, as does isolation.
- Consistency dictates that even failure should be well behaved. Leaving a mess for someone else to clean up is not acceptable.
- Durability can to some extent be reached with precautions in the software, but also says: **BACKUP BACKUP BACKUP!**  
And backups and restores in databases are not just simple file copy actions. . .

# Two phase commit protocol

As an example to implement the A.C.I.D. properties of an (database) application we present the **Two-phase commit protocol**. (text from wikipedia 2008-02-10

[http://en.wikipedia.org/wiki/Two-phase\\_commit](http://en.wikipedia.org/wiki/Two-phase_commit)

- In computer networking and databases, the two-phase commit protocol is a distributed algorithm that lets all nodes in a distributed system agree to commit a transaction.
- The protocol results in either all nodes committing the transaction or aborting, even in the case of network failures or node failures.
- The two phases of the algorithm are the commit-request phase, in which the coordinator attempts to prepare all the cohorts, and the commit phase, in which the coordinator completes the transactions.

## Assumptions for two phase commit

The wikipedia article assumes a few things. More reading or some explanation may be needed to understand it all:

The protocol works in the following manner: one node is the **coordinator**; the rest of the nodes are designated as **cohorts**. The protocol assumes that there is stable storage at each node with a write-ahead log, that no node crashes forever, that the data in the write-ahead log is never lost or corrupted in a crash, and that any two nodes can communicate with each other.

The protocol is initiated by the coordinator after the last step of the transaction has been reached. The cohorts then respond with an agreement message or an abort message depending on success.

## The 2 Phases

The name of the protocol comes from the two phases in the protocol:

- I: **Commit-request phase** The coordinator request all cohorts.  
After that the second phase is started.
- II: **Commit phase** On agreement in the first phase the the coordinator initiates the commit.

# I Commit request

- 1 The coordinator sends a **query to commit** message to all cohorts.
- 2 The coordinator waits until it has a message from each cohort.
- 3 The cohorts execute the transaction up to the point where they will be asked to commit. They each write an entry to their undo log and an entry to their redo log.
- 4 Each cohort replies with an **agreement** message (cohort votes **Yes** to commit), if the transaction succeeded, or an **abort** message (any cohort votes **No**, not to commit), if the transaction failed.

## II Commit phase, success

If the coordinator received an agreement message from all cohorts during the commit-request phase:

- 1 The coordinator sends a commit message to all the cohorts.
- 2 Each cohort completes the operation, and releases all the locks and resources held during the transaction.
- 3 Each cohort sends an acknowledgment to the coordinator.
- 4 The coordinator completes the transaction when acknowledgments have been received.

## II Commit phase, failure

If any cohort sent an abort message during the commit-request phase:

- 1 The coordinator sends a rollback message to all the cohorts.
- 2 Each cohort undoes the transaction using the undo log, and releases the resources and locks held during the transaction.
- 3 Each cohort sends an acknowledgement to the coordinator.
- 4 The coordinator completes the transaction when acknowledgements have been received.

Transactions are a central notion in data processing. What you need to know boils down to: **Atomicity**: either complete success or complete failure (without any traces, effects or residues of the failure).

In transactions, multiple parties (programs or threads, sometimes on different machines) are involved. This needs extra attention to prevent hazards like **deadlock** (indefinite blocking) etc.

Two phase commit is one example of a protocol that helps coordinating the steps in a transaction.

## More reading

Any good book on database design and implementation will have a section on transactions (the essence of real life data base applications), the ACID properties and several commit models.

There you will also find things like journaling or logging, rollback, write ahead log, undo logs, roll forward etc. It makes interesting reading for (would be) database experts!