KUL

Principles of Database Management

PART II : Prof Dr. Wilfried Lemahieu

Ysaline de Wouters

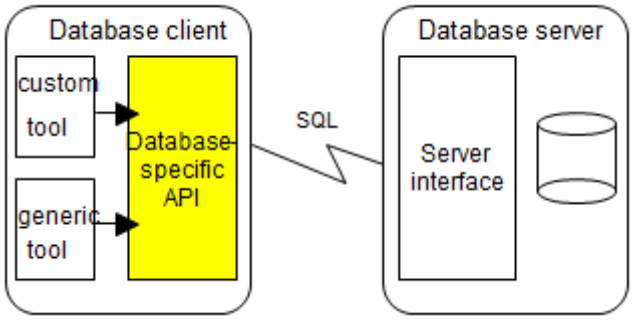
2015 - 2016

Table of Contents

1. Universal interfaces to relational database systems

1.1 Embedded database API versus call-level database API

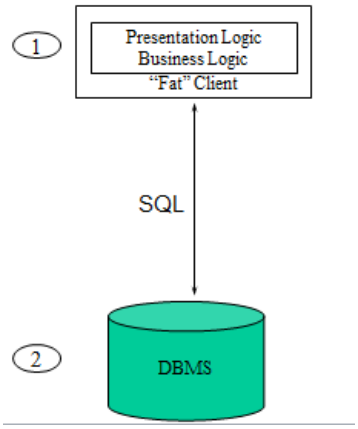
1.1.1 Client/Server interaction



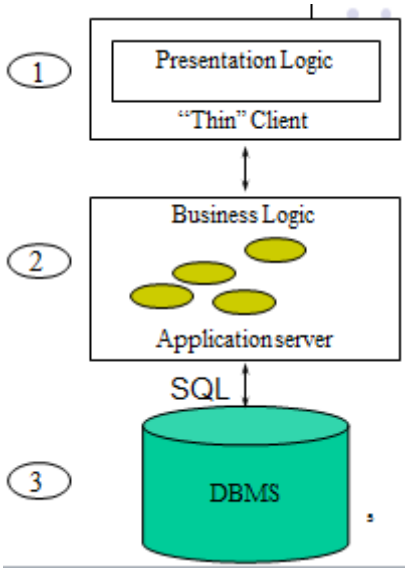
- Database server: it contains the actual data and the database system.
- Database client

An API is needed to interact with the database system.

1.1.2 2 Tiers versus 3 Tiers



In **a 2-tiers** architecture, all the information is on the client side, except from the DBMS. That's why it is called a "fat" client. Two-tier architecture is a client/server architecture, where a request to do some task is sent to the server and the server responds by performing the task.
This architecture works if we don't have too many users and if the business logic is not too complex.



In **3 tier application**, we have a **middle tier** between the DBMS and the client. Here, the Clients are said to be thin clients, which means that these clients only contain presentation logic. Therefore, there is no need for a "strong" computer. Apart from that there are two more layers: application layer and database layer. In three-tier architecture, the data and applications are split onto separate servers, with the server-side distributed between a database server and an application server. The client is a front end, simply requesting and displaying data. The client request is sent to the server and the server in turn sends the request to the database. The database sends back the information/data required to the server which in turn sends it to the client.
Thanks to this three-tiers architecture we do have the World Wide Web and all kinds of applications.
Ex: KuLoket.

1.1.3 Database API

The database API acts as a mediator between application and database. The application talks to the DBMS through the API. The "conversation language" used by both is SQL. Two types of database API can be discerned:

- Embedded API: the SQL instructions are embedded in a host language: a general –purpose programming language such as C, Java or COBOL. There are two separated compilation phases. You translate the text into executing code. We hereby have two languages: SQL code and the host language. The SQL code is processed separately during a pre-compilation phase.
- Call-level API: it provides methods that can be called to perform the desired database operations: establishing a database connection, buffering and execution SQL instructions, processing query results and returning status information. Here, there is only one language compiled.

1.1.4 SQL binding time

SQL is a declarative language and is not a programming language!

- **Declarative**: you tell the system what you want. For instance, I want to know how many products there are.
- **Programming**: tell the system step by step what it should do. It is a procedural language.

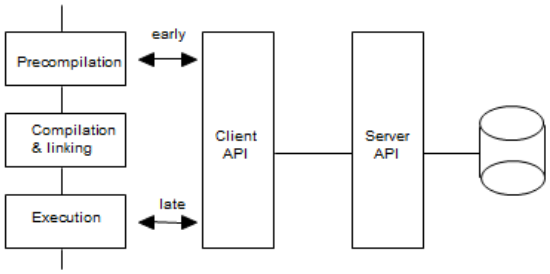
The term **SQL binding** pertains to the conversion of SQL code to an internal format that can be executed by the database engine. The binding time is the moment at which the SQL code is

compiled. At the beginning SQL is just a statement and then it is interpreted by the system. The following tasks are performed:

- Validation of **table and column names**: do these columns exist?
- **Authorization checks**: do I have the required authorization to do this?
- **Generation and optimization** of an **access path**: the optimizer will receive the query, what is the most optimal way to solve it?
- Generation of **machine code**: executable code, stoered in the database catalog to be executed later.
- **Binding** of the access path to the database.

Timing of this conversion

- **Static SQL:** early binding. There is a pre-compilation phase.
- **Dynamic SSQL:** late binding: you bind your query at execution time; the compilation of the programming code invokes a method.



- **Early**: pre-compilation phase. The program contains some SSQL statements and will store it in the catalog. In early binding we have all the steps of binding and after that everything is executed, the access path is used in the execution phase.
- **Late**: bind your query ate execution time. There is no pre-compilation phase. The compilation of the program invokes a method.

What are the advantages and disadvantages of both binding times?

- Early binding (+): if something is wrong in the query, it is detected before whereas, in late binding, it is discovered at execution time.
- Early binding (+): it makes the code more efficient as we have more time to find an optimal path.
- Late binding (+): the SQL code only needs to be known at execution time (dynamic). With early binding, you cannot change the code (static).

	Static	Dynamic
Embedded	SQLJ	✕
Call-level	Stored procedure calls through ODBC or JDBC	ODBC JDBC

- For embedded API, at the pre-compilation phase, the SQL statement has to be known: static.
- Call-level API: we mostly have dynamic compilation.

1.1.5 Stored procedure

A stored procedure is a piece of precompiled executable code, which may consist of both SQL and program language instructions, and which is stored in the DBMSs catalog. The code is activated by an explicit call from an application program.

Advantages:

- Ability to store **behavioral specifications** into the database. You write it once and can then call it several times.
- **Grouping of logically related operations**
- **Host language independence**
- Increased **data independence** since you don't have to know about table and columns involved to calculate something. The stored procedure stays the same so you can simply call them.

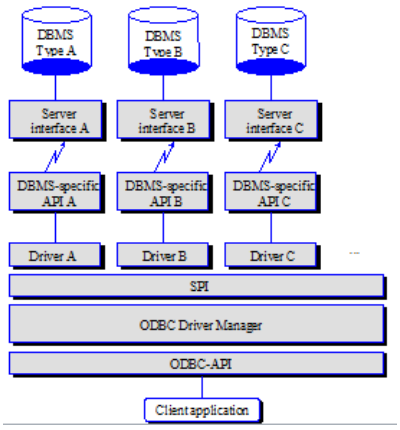
1.2 The *Open Database Connectivity* (ODBC) call-level API

1.2.1 ODBC architecture

ODBC stands for Open Database Connectivity. It was developed by Microsoft as an open standard to provide applications with a uniform interface to relation DBMSs. In this way, the ODBC API hides the underlying DBMS specific API of a particular DBMS vendor. It was the first database independent API.

The ODBC architecture has four components:

- **ODBC Application**: this is a call-level interface, with support for both early binding and late binding. It performs processing and calls ODBC functions to submit SQL statements and retrieve results. It allows opening a database, executing a query, etc.
- **Driver Manager:** he is responsible for selecting the appropriate drive to access a particular DBMS type. He loads and unloads drivers on behalf of an application.
- **The database Driver:** these are routine libraries that are tailored to interact with a particular DBMS type. It processes ODBC function calls, submits SQL requests to a specific data source, and returns results to the application. If necessary, the driver modifies an application's request so that the request conforms to syntax supported by the associated DBMS.
- **The Service Provide Interface (SPI):** this is the interface between the driver manager and the database drivers.



RMQ:

- **Multiple drivers** and data sources can exist, which allows the application to simultaneously access data from more than one data source.
- The **ODBC API is used in two places**: between the application and the Driver Manager, and between the Driver Manager and each driver. The interface between the Driver Manager and the drivers is sometimes referred to as the *service provider interface,* or *SPI*. For ODBC, the application programming interface (API) and the service provider interface (SPI) are the same; that is, the Driver Manager and each driver have the same interface to the same functions.

1.2.2 Typical ODBC functions

- Connect to a data source
- Prepare (bind) an SQL statement, without executing it
- Execute an SQL statement
- Call a stored procedure: if you have pre-compiled statements
- Retrieve query result and status
- Retrieve information form the catalog
- Retrieve information about a drive or data source
- Close a statement
- Close the connection

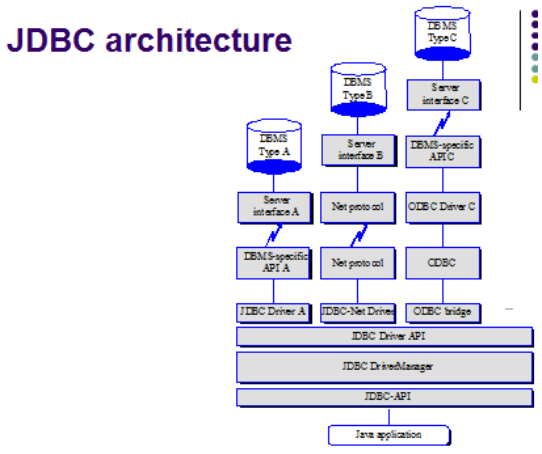
1.3 The *Java Database Connectivity* (JDBC) call-level API

1.3.1 JDBC

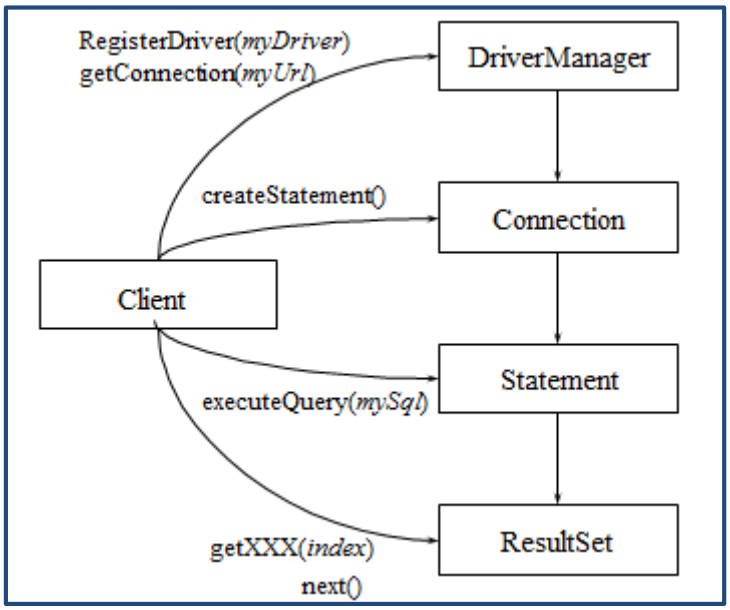
JDBC is not meant for a Microsoft environment but for a **Java language**. Just like ODBC, JDBC implements a **call-level database API** that is independent of the underlying DBMS. JDBC's structure strongly resembles to an ODBC structure. However in contrast to IDBC, JDBC is Java based and exclusively targeted at providing database access to Java applications.

JDBC features typical **Java advantages** such as entirely **object-oriented** constructs, platform **independence**, **dynamic code** loading, etc.

1.3.2 JDBC architecture



1.3.3 JDBC overview



- The **curved arrows** are the **invoke methods**
- The **normal arrows** are the "**results**"

In this JDBC overview we observe 4 objects type:

- **DriverManager:** he provides a uniform interface to establish a database connection. It manages the installed database drivers and provides transparent access to a wide range of database products. 2 functionalities:
  - Register an appropriate driver (if this has not been done already) for the database system that will be accessed.
  - A call to the GetConnection method then results in the database being connected by means of the corresponding driver. The database is identified by means of a parameter (e.g. its URL). Other optional parameters are a user ID and password.

  The driver interface: a driver object is never accessed directly by an application: theoretically, it is of no concern to an application. The Driver mediates between DriverManager and database.

  Driver types:
  - **Driver to access a DBMS through its native** interface: uses the Java Native Interface to make calls to a local database library API.
  - **DBMS-neutral 'net protocol' driver**: database driver implementation which makes use of middle-tier between the calling program and the database.
  - **JDBC-ODBC bridge driver:** database driver that uses the ODBC driver to connect the database. This driver translates JDBC method calls into ODBC function call. The bridge implements JDBC for any database for which an ODBC driver is available.

- **The connection interface:** A connection object represents an individual database session. It encompasses the functionality required to maintain a database connection. Also, it generates statements to execute SQL queries. As long as the connection object exists, we are connected to that objects.
  Another important function of a connection object is transaction control. For that purpose, it defines commit and rollback methods and supports locking.
- **The statement interface:** a statement object is the Java abstraction of an SQL query. It generates a ResultSet object that will contain the query result.
  Separate methods are provided to execute SELECT queries, UPDATE queries and stored procedures. Some subtypes of statement feature additional functionality.
    - **PreparedStatement:** separates binding and execution of a query;
    - **CallableStatement:** calls a stored procedure.
- **The ResultSet interface:** a ResultSet object contains the query result of a select query that was executed through a statement object. Its structure is conceived as rows and columns. Each ResultSet maintains a cursor, which indicates the current row in the result. The cursor can be moved by calling the next() method.
  The result's current row can be read by means of getXXX() methods. The XXX refers to the data type of the desired column: string, integer, Boolean, etc.
  It is also possible to inquire for metadata. Ex: regarding the data types and column names of the ResultSet (see how many columns and rows there are).

  There are other classes and interfaces:

    - The ResultSetMetaData interface: used to retrieve metadata regarding the query result
    - The DatabaseMetaData interface: used to retrieve metadata regarding the database. Ex: information about the number of users or number of tables.
    - The SQLException class: used for error handling. Examine what would happen if there was a mistake or an error.

1.3.4 JDBC examples

```
// ----> Initialisation
  int supplierNr;
  String supplierName;
  int minimumRating;

// ----> Setup database connection
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
  Connection myConnection
    = DriverManager.getConnection(myUrl, myUid, myPassw);

// ----> Specify and bind query
  PreparedStatement myStatement = myConnection.prepareStatement(
    "select suppliernumber, suppliername
     from suppliers where supplierrating > ?");

// ----> Set parameter and execute query
  minimumRating = 30;
  myStatement.setInt(1, minimumRating);
  ResultSet myResultSet = myStatement.executeQuery();

// ----> Show results
  while myResultSet.next()  {
     supplierNr = myResultSet.getInt("suppliernumber");
     supplierName = myResultSet.getString("suppliername");
     System.out.println(supplierNr + ": " + supplierName);
  };

// ----> Disconnect
  myStatement.close();
  myConnection.close();
```

JDBC example

```
myConnection.setTransactionIsolation(
   Connection.TRANSACTION_SERIALIZABLE);
myConnection.setAutoCommit(false);

Statement myStatement1 = myConnection.createStatement();
Statement myStatement2 = myConnection.createStatement();
Statement myStatement3 = myConnection.createStatement();
Statement myStatement4 = myConnection.createStatement();

myStatement1.executeUpdate(myQuery1);
myStatement2.executeUpdate(myQuery2);
Savepoint mySavepoint = myConnection.setSavepoint();

myStatement3.executeUpdate(myQuery3);
myConnection.rollback(mySavepoint);

myStatement4.executeUpdate(myQuery4);
myConnection.commit();
```

Transaction management in JDBC

```
String myQuery = "update suppliers set supplierrating = ? where suppliername = ?"
PreparedStatement myStatement = myConnection.prepareStatement(myQuery);
int numberOfRows;

myStatement.setInt(1, 35)
myStatement.setString(2, "Demey")
numberOfRows = myStatement.executeUpdate();

myStatement.setInt(1, 80)
myStatement.setString(2, "Dehaene")
numberOfRows = myStatement.executeUpdate();

myStatement.setInt(1, 60)
myStatement.setString(2, "Decock")
numberOfRows = myStatement.executeUpdate();
```

Prepared statements and parameters in JDBC

```
CallableStatement myStatement
  = myConnection.prepareCall(" {call calculate_supplierrating(?, ?)}");

myStatement.registerOutParameter(2, java.sql.Types.INTEGER);

myStatement.setString(1, "Demey")
myStatement.execute();
int statusDemey = myStatement.getInt(2);

myStatement.setString(1, "Dehaene")
myStatement.execute();
int statusDehaene = myStatement.getInt(2);

myStatement.setString(1, "Decock")
myStatement.execute();
int statusDecock = myStatement.getInt(2);
```

Invocation of stored procedures in JDBC

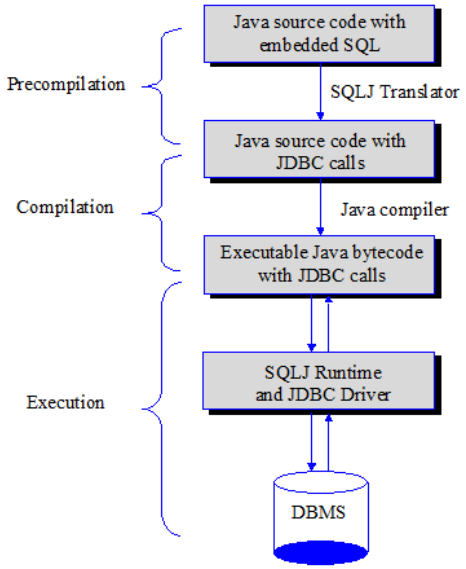
```
String myQuery
  = "select suppliernumber, suppliername, supplierrating
     from suppliers where suppliercity = 'Brussels'";

Statement myStatement = myConnection.createStatement(
  ResultSet.TYPE_SCROLL_SENSITIVE,
  ResultSet.CONCUR_UPDATABLE);
ResultSet myResultSet =  myStatement.executeQuery(myQuery);

myResultSet.first();
int supplierNr = myResultSet.getString(1);
String supplierName = myResultSet.getString(2);
int supplierRating = myResultSet.getString("supplierRating");

myResultSet.next();
myResultSet.next();
myResultSet.updateInt("supplierrating", 85);
myResultSet.updateRow();

myResultSet.absolute(40);
myResultSet.updateInt("suppliernumber", 103);
myResultSet.updateString(2, "D'Haese");
myResultSet.updateInt("supplierrating", 60);
myResultSet.insertRow();

myResultSet.previous();
myResultSet.deleteRow();
```

ResultSet manipulation in JDBC

1.4 The *SQLJ* embedded API

SQLJ implements the embedded SQK standard for JAVA. Only static SQL is supported. Dynamic SQL was already established with JDBC. The static code is precompiled into standard Java instructions, which access the database through an underlying call-level interface. The Java code is then compiled in its entirety.

At compile time, the SQL code can be checked for syntactical correctness, type compatibility and conformance to the corresponding database schema.



SQLJ concepts

- SQLJ-clauses delimit the SQLJ code, i.e. the code that is to be subject to precompilation. SQL statements are preceded by " #sql ". Everything that is after this is SQL code.
- Each SQLJ clause also defines a connection-context, as specified by means of JDBC. It is also possible to specify a default context.
- Host variables: Java objects can be used as parameters in SQLJ instructions, so as to be able to pass information between Java and SQL.
- Iterator: this is an object that contains the result rows of an SQL query, which can be accessed one by one (cf. cursor).
- CALL statement: used to invoke a stored procedure.

2. Transactions, recovery and concurrency control

2.1 Introduction

A transaction represents **series of database operations** that are to be **executed as one**, **undividable** whole because, on the one hand, users should never be able to see inconsistent data and, on the other hand, it should be impossible to complete a set of operations in such a way that the database is left in an inconsistent state.

In other words, a transaction consists of a set of database operations that are guaranteed to bring the database from one consistent state into a new consistent state. Ex: when we want to transfer money. Money is withdrawn from one account but it should also be added to the other account. If one transaction fails → cancel → rolled back. All the effects should then be wiped from the database.

Transaction processing systems are systems with **large databases** and hundreds of **concurrent users** executing database transactions.

Ex: airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, etc.

⇨ These systems require high availability and fast response time for hundreds of concurrent users.

Delimiting a transaction

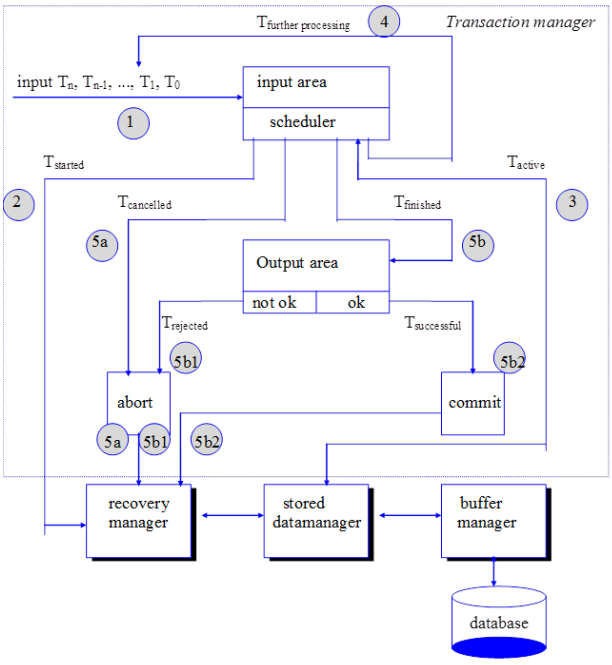
- **Beginning** of a transaction
  - Implicit or through **BEGIN TRANSACTION** instruction
  - The transaction is fed to a **transaction manager**
  - The transaction, together with other transactions, is put into a **schedule.** The schedule consists of SQL statements to be done in the database environment. It contains a planning of which database instructions should be executed and in which order they should be.
  - The transaction is started.
- **End** of a transaction:
  - Implicit or through **END TRANSACTION** instruction
  - There are two possibilities
    - **Successful** end: COMMIT. All changes made by the transaction are made persistent, definitive, stored in the database. We have to find out a way that data will remain persistent even if the system crashes.
    - **No successful** end: ABORT or ROLLBACK. If it made some partial change, it needs to roll back → if it didn't succeed as an whole

```
myConnection.setTransactionIsolation(
  Connection.TRANSACTION_SERIALIZABLE);
myConnection.setAutoCommit(false);

Statement myStatement1 = myConnection.createStatement();
Statement myStatement2 = myConnection.createStatement();
Statement myStatement3 = myConnection.createStatement();
Statement myStatement4 = myConnection.createStatement();

myStatement1.executeUpdate(myQuery1);
myStatement2.executeUpdate(myQuery2);
Savepoint mySavepoint = myConnection.setSavepoint();

myStatement3.executeUpdate(myQuery3);
myConnection.rollback(mySavepoint);

myStatement4.executeUpdate(myQuery4);
myConnection.commit();
```

Example: transaction management in JDBC

- **Serializable**: my transaction won't interfere with your transaction
- **SetAutoCommit** → true: all transactions are single SQL transactions. If it is false, we combine multiple transactions into one transaction
- No safe point: atomic. if you don't give safe point as a parameter you may lose all previous data.

If there is a safe point, in case there is a crash or any other problem, we can roll back to a certain point within the transaction. So we won't lose all changes made since we will be able to retrieve data until the save point.



Overview

Step 1: input transaction

Step 2: The **Recovery manager** is notified. He should be able to recover from mistakes, if a transaction fails, the manager will clean up the mess, undo what shouldn't have been done and redo other stuff. He will also keep a diary up to date.

The **scheduler** will then say when it is time for a transaction to be started.

Step 3: transaction is active and will be executing statements interacting with the database file. Stored datamanager: responsible for the physical interaction with the database files, writes changes to the buffer → The buffer writes. The buffer manager is in charge of buffering some date. Write to the buffer and read what is in the buffer. Updates are buffered and flushed from time to time.

Step 4: Pass on new SQL statements. Different statements are executed together.

Step 5:

- **The transaction is cancelled**, not completed. This means that something went wrong during the transaction. For instance, we did not receive input from the user. In this case, the transaction is aborted. The recovery manager will keep track of what happened.
- **The transaction is finished**: successfully or not
  - Finished but rejected: transaction conflicted with another transaction or not and it has to be aborted.
  - Finished and successful, the transaction is committed.

⇨ Whatever what happens, changes needs to be stored in the database

Locking

A **lock** is an access privilege that can be granted over an object, so as to ensure that the object is manipulated by only one operation at a time, in case where multiple operations simultaneously try to access the same object.
A **locking protocol** specifies the rules that determine when transactions can obtain locks, respectively release locks.

2.2 Transactions and transaction management

```
< begin delimiter >
WHENEVER SQLERROR GOTO UNDO
< end delimiter >

< begin delimiter >
UPDATE ACCOUNT                                  ← beginning of transaction
SET AMOUNT = AMOUNT - :TRANSFER
WHERE ACCOUNT = :ACCOUNTFROM
< end delimiter >

< begin delimiter >
UPDATE ACCOUNT
SET AMOUNT = AMOUNT + :TRANSFER
WHERE ACCOUNT = :ACCOUNTTO
< end delimiter >

< begin delimiter >
COMMIT WORK                                     ← end of transaction
< end delimiter >

UNDO
< begin delimiter >
ROLLBACK WORK                                   (or) ← end of transaction
< end delimiter >
```

2.2.1 Registration on the logfile

A **logfile** is a file with redundant information that is necessary to recuperate or restore the data, after some data have been lost or damaged. It is the diary of the log manager. A logfile is said to contain redundant information since all the data is also written to the database.

For each transaction, several pieces or data are registered on the logfile in so-called **logrecords**. The most important ones are:

- ID of the transaction, a mark that denotes the beginning of the transaction, the exact time at which the transaction was started and the type of the transaction (read-only or read-write transaction);
- ID of the records that are used in the transaction and the type of operations executed on these records (select, update, insert, delete);
- Before images: undo part of the logfile
- After images: redo part of the logfile
- The current state of the transaction: active, commited or aborted
- Checkpoint records: synchronisation points

- **Write ahead log strategy:** updates to the logfile should always precede the corresponding updates in the database

→ A transaction can only execute a database operation if the before images are already registered on the logfile.
→ It can only be committed if the after images of the corresponding data (as well as the commit sign) are registered on the logfile.

- **Force writing the logfile strategy:** the logfile should be written to disk before a transaction can be committed

A transaction table contains one row for each active transaction. Such row contains:

→ Transaction **identifier**
→ **Current state** of the transaction
→ **Log sequence number** of the most recent log record for this transaction

2.2.2 Transaction properties

Memo : « ACID » properties

- Atomicity: all operations, considered as multiple transactions, should be treated as one operation. In other words, it is everything or nothing.
- Consistency: the transaction should bring the database from one consistency to another. This is the responsibility of the programmer.
- Isolation : (concurrency control) : if the transaction by itself is correct, it should be the same result as the one of executing several transactions (= executing them in isolation). We have to make sure the transactions don't interfere.
- Durability : the database system should make sure the changes end up in the physical file of a database

2.3 Recovery

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database or that the transaction does not have any effect on the database or any other transactions.

⇨ Several kinds of **errors** or **calamities** may occur during the execution of a transaction:

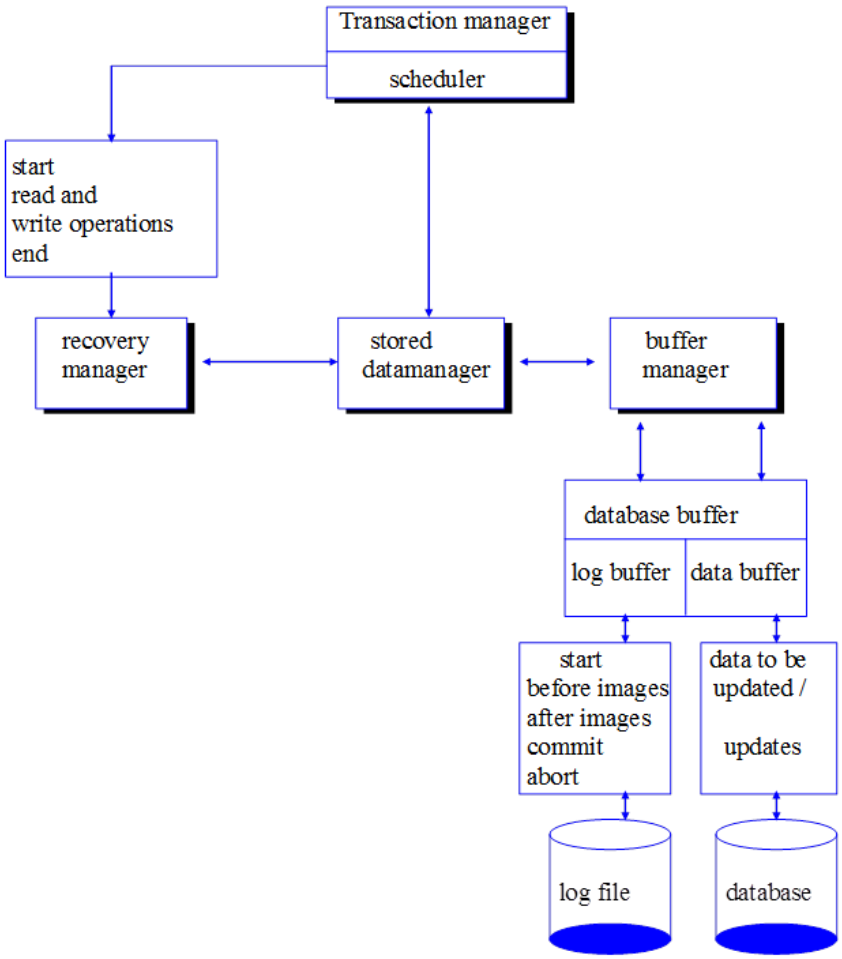
- **A head crash of a harddisk**: Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
- **A computer failure**: A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures. For instance, main memory failure.

- A system fault due to the operating system;
- **Physical problems and catastrophes**: this refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistakes.
- A system fault due to the database management system;
- A program fault (Ex: a syntax error);
- Concurrency control enforcement: the concurrency control method may decide to abort a transaction because it violates serializability or it may abort one or more transactions to resolve a state of deadlock among several transactions. Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.

⇨ As a consequence, the transaction won't be executed properly and therefore we have to deal with such problems.

> Recovery is the activity of ensuring that data can always be recuperated, regardless of the kind of error or calamity that causes the data to be lost or damaged

The DBMS uses a **recovery manager** to coordinate recovery. The latter ensures that only effects of successful transactions are persisted into the database and that all (partial) effects of unsuccessful transactions are undone.



2.3.1 Types of failures

First we have to find out which type of failure cause the transaction to crash.

- Transaction failure: a transaction 'aborts'. This decision can be made by the application or by the database system.
- System failure: the content of the main memory is lost. In particular, this means that the content of the database buffer is lost.
- Medium failure: some data in the database and/or the logfile are damaged or destroyed.

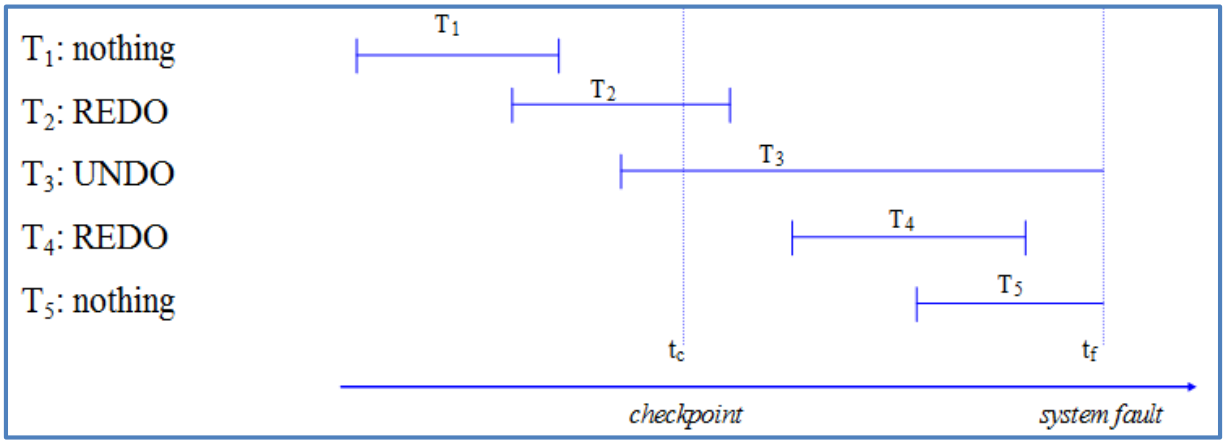
⇨ Techniques for **system recovery**: applied in the case of system failure and (sometimes) transaction failure.
⇨ Techniques for **medium recovery**: applied in the case of medium failure.

2.3.2 System recovery

A system fault has occurred and as a consequence, the content of the database buffer is lost. The transactions that have written data to this buffer belong to either of two categories:

- Transactions that had reached a 'committed' state at the moment of the fault.
- Transactions those were still active at the moment of the fault.

Whether UNDO and REDO operations are necessary, depends on whether the changes induced by the transactions up to that moment were already persisted into the database at the moment of the fault. The latter depends on how the recovery manager controls the buffer manager in writing the database buff to disk.



Assumption: the recovery manager periodically commands the buffer manager to empty the database buffer (i.e. write its content to disk). A 'checkpoint' is registered on the logfile and a list of transactions is created, which denotes which transactions where finished and which ones where still active at the same time when the checkpoint was registered.

RMQ: checkpoint= database buffer was written to the disk. System fault = database buffer is emptied.

T_1: transaction 1 is completely saved.

T_2: some changes of T2 are saved on the logfile but not all of them. After the system fault, some changes can be recovered since these are written on the disk, these are the changes made before the checkpoint. However, the part after the checkpoint has to be redone starting from the flushing point.

T_3: some changes are written into the disk (changes until the checkpoint). Since the transaction was not committed, it cannot be completed. It has to be undone, as if T_3 never happened.

T_4: everything has disappeared → use after images to write all the changes to the database

T_5: changes written to database buffer, system fault → buffer is empty → no changes of T_5 disappear

⇨ Once a transaction is committed, you cannot change things of the transaction

2.3.3 Medium recovery

Medium recovery is applicable when the database and/or logfile is damaged because of medium failure.

Solution: redundancy. Two approaches:

- **Disk mirrorring:** data are written in real-time to two or more disks. Updates should be applied on every mirror disk in the same sequence. Upon medium failure, the mirror version of the data can be used.
- **Archiving**: the database (and the logfile) is duplicated periodically.
    - When a medium failure occurs, the most recent backup is restored. If the log is not damaged, the effects of committed transactions can be reconstructed by means of a rollforward utility that uses the *REDO* part of the logfile. Effects of transactions that were still active at the time of the failure don't have to be cleaned up, because (by definition) they have disappeared as a consequence of the failure.
    - Full back up versus incremental backup

  With archiving, we are likely to lose a part of the data if the system crashes. Indeed, if we did the last back-up two hours ago, we won't be able to retrieve data collected between the back up and the crash.

⇨ It is saver to combine both approaches.

2.4 Concurrency control

When multiple transactions are executed simultaneously, and no preventive measures are taken, the following problems may occur because of **interference** between the transactions' actions. In other words, in a multiuser environment, transactions may interfere, causing some unwanted side effects.

- Lost update problem
- The uncommitted dependency problem
- The inconsistent analysis problem

> Concurrency control is the activity of coordinating the operations of simultaneous transactions, which affect the same data, in such a way that the data cannot become inconsistent.

⇨ If we don't take it into account, the scheduler will schedule statements in such a way it is the more efficient.

So as to avoid such inconsistencies, the scheduling of the transactions as planned by the scheduler, should be serializable.

LOST UPDATE PROBLEM

Time	T_1	T_2	$account_x$
t_1		begin transaction	100
t_2	begin transaction	read($account_x$)	100
t_3	read($account_x$)	$account_x = account_x + 120$	100
t_4	$account_x = account_x - 50$	write($account_x$)	220
t_5	write($account_x$)	commit	50
t_6	commit		50

In this lost update example transactions only interacts with one data element. We have 2 transactions, which each execute multiple statements. The final value is wrong. It should indeed be equal to 170. What goes wrong?

In t_1, transaction 2 begins. It then read account. Next, the value of account is changed and 120€ are added to the initial value. The new value is then written and the transaction is committed in t_5.

Regarding transaction 1, it starts in t_2. It then read account. In t_4, transaction 1 substracts 50€ to the initial account. The new amount is then written and the transaction is committed in t_6. The problem is that we should end with a total amount of 170. Here, transaction 1 doesn't know that transaction 2 has written a new value to the account, so it calculates with the original number.

The read and write operations are not scheduled properly so we end up with an incorrect state.

If T_1 is first executed and then T_2, we won't have any problems.

UNCOMMITTED DEPENDENCY PROBLEM

This is also called the **"dirty read problem".**

Time	T_1	T_2	$Account_x$
t_1		begin transaction	100
t_2		read($account_x$)	100
t_3		$account_x = account_x + 120$	100
t_4	begin transaction	write($account_x$)	220
t_5	read($account_x$)		220
t_6	$account_x = account_x - 50$	rollback	100
t_7	write($account_x$)		170
t_8	commit		170

T1 reads the changes done by T2 and calculates the amount based on the 220€ amount calculated by T2. So T2 has still some effect on the database. But it was rolled back so it shouldn't have any effects. We should end up with a final amount of 50€.

In this example, two properties are violated:

- Isolation

- Atomicity: the outside world doesn't need to see the calculations.

INCONSISTENT ANALYSIS PROBLEM

Time	T_1	T_2	$Account_x$	y	z	*sum*
t_1		begin transaction	100	75	60	
t_2	begin transaction	sum = 0	100	75	60	0
t_3	read($account_x$)	read($account_x$)	100	75	60	0
t_4	$account_x = account_x - 50$	sum = sum + $account_x$	100	75	60	100
t_5	write($account_x$)	read($account_y$)	50	75	60	100
t_6	read($account_z$)	sum = sum + $account_y$	50	75	60	175
t_7	$account_z = account_z + 50$		50	75	60	175
t_8	write($account_z$)		50	75	110	175
t_9	commit	read($account_z$)	50	75	110	175
t_{10}		sum = sum + $account_z$	50	75	110	285
t_{11}		commit	50	75	110	285

Here we have three values being manipulated. The 4th column is the calculation made. There is no problem in terms of values being overwritten. The final values are all correct. However, there appears an inconsistency in the sum. Account x is read before the 50 is subtracted. Account z is read after the 50 added.

⇨ This problem is less severe because the final values are correct.

2.4.1 Serial and serializable schedules

A schedule S consisting of n transactions is defined as a sequential ordering of the operations of these n transactions, in such a way that for each transaction T in S the following holds: if operation i precedes operation j in T, the operation i precedes operation j in the schedule S. The schedule must always respect the order of the operations, but there is no ordering enforced on operations belonging to different transactions. The ordering of operations between the respective transactions can be scheduled randomly.

⇨ Each alternative ordering results in a different schedule.

The problem is that the throughput will be very low and operations will have to wait a long time. Therefore, we have to schedule operations from different transactions in an interleafed way.

> A schedule S is **serial** if for each transaction T of the schedule holds that all operations of the transaction are **processed consecutively**. For a set of n transactions, there exist n! serial schedules.

If we assume that a transaction is correct if it is executed in isolation and if the transactions in the schedule are independent from one another then we can conclude that a serial schedule is

always correct. Such a schedule guarantees that database consistency is not affected by interference between transactions, since the transactions are executed one by one.

But serial schedules put a heavy burden on transaction throughput. We need non-serial schedules that are still correct. Non-serial schedules are correct if they are serializable, i.e. if they are equivalent to a serial schedule.

⇨ A schedule is **serializable** if it produces the same output and effects on the database as a fully serial execution of the same transactions. It is better to have concurrent executions but we have to make sure they do not interfere.
Techniques for serializability: a.o. locking.

2.4.2 Equivalent schedules

Two schedules S_1 and S_2 with the same transactions T_1, T_2, ..., T_n are equivalent if the following two conditions are satisfied:

- For each operation $read_x$ of T_i in S_1, the following should hold: if the value x that is read by this operation was last updated by an operation $write_x$, executed by a transaction T_j in S_1, then the same operation $read_x$ of T_i in S_2 should read the value x as written by the same operation $write_x$ of T_j in S_2.
- For each value x for which the schedules contain an update operation, the following should hold: the last operation $write_x$ as induced by T_i in S_1, should also be the last update operation of T_i on x in S_2.

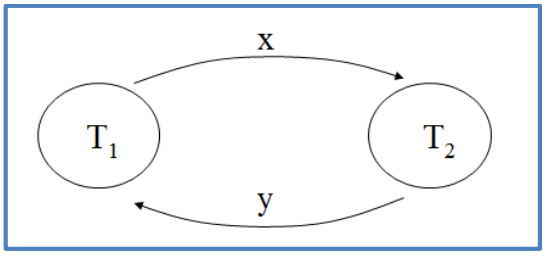
The scheduler could check for each non-serial schedule whether it is serializable.

Alternatively, we could apply transaction scheduling methods that guarantee serializability: locking protocols. These protocols make sure that we cannot have non-serializable schedules. So, for each schedule, it verifies if it is serializable.

2.4.3 Checking for serializability

To check a schedule for serializability, a so called **precedence graph** can be created. A precedence graph consists of a **node** for each transaction and a **directed edge** $T_i \rightarrow T_j$ if T_j reads a value that was last written by T_i or T_j writes a value after it was read by T_i. The schedule is serializable if and only if the precedence graph has **no cycles**.

⇨ The problem of this method is that it would **slow down** the system and it causes loads of **overheads**.



	schedule S_1 *serial schedule*		*schedule S_2* *non-serial schedule*	
Time	T_1	T_2	T_1	T_2
t_1	begin transaction		begin transaction	
t_2	read($account_x$)		read($account_x$)	
t_3	$account_x = account_x + 50$		$account_x = account_x + 50$	
t_4	write($account_x$)		write($account_x$)	
t_5	end transaction			begin transaction
t_6	read($account_y$)			read($account_x$)
t_7	$account_y = account_y - 50$			$account_x = account_x \times 2$
t_8	write($account_y$)			write($account_x$)
t_9	end transaction			read($account_y$)
t_{10}		begin transaction		$account_y = account_y \times 2$
t_{11}		read($account_x$)		write($account_y$)
t_{12}		$account_x = account_x \times 2$	read($account_y$)	end transaction
t_{13}		write($account_x$)	$account_y = account_y - 50$	
t_{14}		read($account_y$)	write($account_y$)	
t_{15}		$account_y = account_y \times 2$	end transaction	
t_{16}		write($account_y$)		
t_{17}		end transaction		

Serial and (non-) serializable schedules: example

2.4.4 Concurrency control solutions in practice

Theoretically, each non-serial schedule could be checked for serializability. However, this would cause a lot of overhead for the DBMS. Instead, **protocols** will be used that guarantee **serializability**. Generally, we can distinguish between

- **Optimistic protocols** (cf. *optimistic schedulers*): chances that there are problems are very low. I will schedule everything right away and I won't test all the time. Only test if the transaction tries to commit then I perform a test. If there are interferences, I abort the transaction, rollback, and start again.
  - ➔ **The optimistic scheduler** assumes that conflicts between simultaneous transactions occur rarely. Each operation for each transaction is scheduled without delay. At the moment when a transaction is completed, and is about to be committed, the scheduler checks whether conflicts have occurred between this transaction and other transactions. If there were conflicts, all changes induced by this transaction have to be undone by means of a rollback.
- **Pessimistic protocols** (cf. *pessimistic schedulers*): high risk of interferences. So we won't just schedule everything at the same time. We will postpone the schedule until we can make sure the transactions won't interfere. It is better to avoid interferences.
  - ➔ **Pessimistic scheduler** assumes that it is rather likely that transactions will interfere and cause conflicts. Therefore, the execution of the operations is delayed a bit, until the scheduler can 'overview' the situation so as to enforce a schedule that minimises the risk for conflicts. Extreme case: serial scheduler.

2.4.5 Locking and locking protocols

As a reminder, a lock is a piece of information about who is accessing the data.

Two operations from different transactions **conflict** if:

- They are to be executed on the same database object
- (at least) One of them is a 'write' operation. Nevertheless, if they are all reading, we won't have conflicts since no data is changed.

⇨ Such database objects can (a.o.) be a row in a table or a record in a file.

A **Locking protocol** is a set of rules enforcing that, in case where two conflicting operations try to access the same object, the access to this object is granted to only a single operation at a time. For that purpose, a lock is placed on the object. A **lock** is a variable associated with a database object, for which the value determines which operations are allowed (at this time) on the object.

- Locking: Locks are granted, depending on the types of the operations that are to be executed on the data.
- Unlocking: existing locks are released
- Locking manager: grants and releases locks. These decisions are based on a 'lock table' and a locking protocol.

Types of locks:

- **Exclusive lock** (write lock): a transaction obtains the exclusive right to access an object. No other transactions can read from or write to this object until the lock is released.
- **Shared lock** (read lock): a transaction obtains the guarantee that, as long as it holds the lock, no other transactions can write to that object. The lock is granted to one or more transactions. It can be shared by multiple transactions.

COMPATIBILITY MATRIX

State of a lock on an object

type of the requested lock on the object	*unlocked*	*shared*	*exclusive*
unlocked	-	yes	yes
shared	yes	yes	no
exclusive	yes	no	no

SCHEDULING STRATEGY

When an **exclusive lock** is released, any of the waiting transactions is a candidate to acquire a lock on the object at hand. The lock manager applies a **scheduling strategy** to determine which transaction gets the lock.

When a **shared lock** is released, it is possible that (shared) locks from other transaction remain on the same object. The lock manager will use a priority schema to determine whether...

- New shared locks are granted to other transactions
- Or the shared locks are gradually removed in favour of a transaction that waits to acquire an exclusive lock on this object
- An 'unfair' priority schema may result in livelocks, with some transactions remaining in an 'endless' waiting state.

Locking can be done explicitely and implicitely. Ifyou don't do it explicitely, it will be done implicitely at the moment we read the data.

- Lock requests
  - s-lock instruction
  - x-lock instruction
- Releasing locks
  - Unlock instruction
- Implicit locking
  - Read operation → s-lock
  - Write operation → x-lock
- Implicit release of locks
  - Commit or abort instruction → unlock

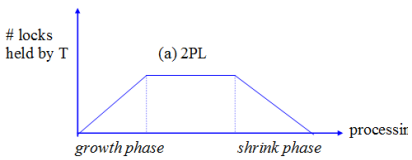
THE TWO-PHASE LOCKING PROTOCOL

Application of the compatibility matrix still does not guarantee serializable schedules. To guarantee serializability in all circumstances, a protocol is needed that enforces rules about the **moment**(s) on which lock and unlock instructions are allowed in a transaction. Such a protocol is the **Two-Phase Locking Protocol.**

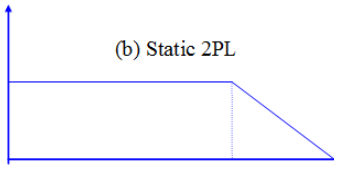
The 2PL-Protocol applies the following rules:

- When a transaction wants to ''read' resp. 'write' from/to an object, it will first have to request and acquire a read lock resp. write lock with the lock manager.
- The lock manager determines, based on the compatibility matrix, whether some operations are conflicting or not and grants the lock right away or determines to postpone it.
- Requesting and releasing locks occurs in two phases:
  - A **growth phase** in which the transactions acquires new locks without releasing any locks. If the number of locks held by a single transaction increases.
  - A **shrink phase** in which the transaction releases locks without acquiring new ones. The total number of locks decreases.

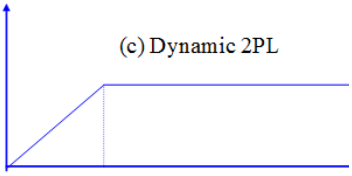
⇨ A transaction satisfies the latter rule if all of its locking instructions preceede the first unlock instruction.



We acquire more and more locks. Then we have a stable phase. Then we have the first unnlock phase and the locks start decreasign until we reach the last one.



Here, we acquire all locks since the beginning. Iknow which data I will be interacting with.



We first have a growth face. We start acquiring locks and we keep all of them until the moment we commit. Then we relase all of them.

⇨ Which one is the best? It depends on what we call the best. If the best is to have the most throughput, then (a) is the best because we hold locks for a shorter time. Holding a lock means transactions will have to wait. It will reduce the throughput.
For case (b), you have to know which data you have to be accessing, lock all the employees up front even though you don't change all the employees, you lock more data then you actually need, you can't lock data later on.

Time	T_1	T_2	$Account_x$
t_1		begin transaction	100
t_2	begin transaction	x-lock($Account_x$)	100
t_3	x-lock($Account_x$)	read($Account_x$)	100
t_4	wait	$Account_x = Account_x + 120$	100
t_5	wait	write($Account_x$)	220
t_6	wait	commit	220
t_7	wait	unlock($Account_x$)	220
t_8	read($Account_x$)		220
t_9	$Account_x = Account_x - 50$		220
t_{10}	write($Account_x$)		170
t_{11}	commit		170
t_{12}	unlock($Account_x$)		170

Solution to the lost update problem

➔ Compare with other lost update problem. Account x is locked → exclusive → change is made and written and the transaction is committed and unlocked. T_1 has to wait until the lock on account x is released and it can read the correct value. They have acknowledged each other's updates.

Time	T_1	T_2	$Account_x$
t_1		begin transaction	100
t_2		x-lock($Account_x$)	100
t_3		read($Account_x$)	100
t_4	begin transaction	$Account_x = Account_x + 120$	100
t_5	x-lock($Account_x$)	write($Account_x$)	220
t_6	wait	rollback	100
t_7	wait	unlock($Account_x$)	100
t_8	read($Account_x$)		100
t_9	$Account_x = Account_x - 50$		100
t_{10}	write($Account_x$)		50
t_{11}	commit		50
t_{12}	unlock($Account_x$)		50

Solution to the uncommitted dependency problem

➔ Exclusive lock on account x. Roll back the original value and then T1 can work with the original value after waiting and unlocking

Time	T_1	T_2
t_1	begin transaction	
t_2	x-lock($Account_x$)	
t_3	read($Account_x$)	
t_4	$Account_x = Account_x - 50$	
t_5	write($Account_x$)	
t_6	x-lock($Account_y$)	
t_7	wait	
t_8	wait	
		begin transaction
		x-lock($Account_y$)
		read($Account_y$)
		$Account_y = Account_y - 30$
		write($Account_y$)
		x-lock($Account_x$)
		wait

Consequences of applying the 2PL-protocol

New problem

T_1 has locked account x but waits for the unlocking of account y and T_2 has locked account y but wait for the unlocking of account x.

T_1 waits for T_2 and T_2 waits for T_1 → **unsolved**

- → Static 2PL protocol will solve this problem
- → But you hold the locks for a long time and you only need the locks at the dashed line.
- → You hold more data then you need → severe impact on the throughput

⇨ **Deadlock prevention:** this is a.o. realised by static 2PL. With this protocol, each transaction should acquire all of its locks from the start. If a transaction fails in acquiring all necessary locks at that time, it is put in a wait state. As a result, deadlocks are avoided. The downside is a (possibly severe) decrease of the throughput.

⇨ As a consequence, it seems to be better not to take measures to avoid deadlocks, but to detect and resolve them (deadlock detection and resolution).

In order to detect a deadlock situation, a **wait-for-graph** can be used. A wait-for-graph consist of a node for each (active) transaction and a directed edge $T_i \rightarrow T_j$ if transaction T_i waits for a lock on an object that is currently locked by transaction T_j. A deadlock exists iff there exists a cycle in the wait-for-graph.

Because the presence of a cycle is a necessary and sufficient condition for a deadlock situation, a deadlock detection algorithm will have to investigate the wait-for-graph at regular intervals. The choice of an appropriate interval is important. If it is too short, the algorithm will cause much unnecessary overhead. If it is too long, deadlock situations may exist for a long time without being detected.

Once a deadlock is detected, it should be resolved. Victim selection is the activity of selecting one or more transactions involved in a deadlock, and consequently aborting them. As a result, changes made to the database data by these transactions should be rolled back. So as to minimise the amount of overhead, it is (given equal priorities) better to avoid aborting transactions that made a lot of changes to the database.

2.4.6 Variation points

- **Transaction isolation levels**
  - In practice, multiple isolation levels are applied, so as to increase the throughput
  - A short-term lock on an object is a lock that is only held during the execution of the operation associated with the lock. If short term locks are used, rule 3 of the 2PL-protocol is violated, i.e. serializability can no longer be guaranteed
  - Depending on the requirements of the transaction, several isolation levels can be distinguished:
    - Uncommitted Read (UC): no concurrency control
    - Committed Read (CR): long-term write locks + short-term read locks (problem: inconsistent analysis)
    - Serializable: cf. 2PL
- **Locking granularity levels**
  - The 'object' of a lock can be a row, a table, a physical block, a tablespace, ...
  - Tradeoff:
    - Lock on fine grained objects: more throughput
    - Lock on coarse grained objects: less overhead
  - Multiple-Granularity Locking- Protocol (MGL-protocol): extends 2PL to allow for locking at multiple granularity levels (row, table, ...)
    - consistency required between different levels of hierarchy, e.g. a lock on a table and a lock on a row in that table
    - More complex compatibility matrix

2.4.7 Cascading rollback

The rollback of a transaction may cause a cascade of rollbacks of transactions that used the data that were updated by the initial transaction that was rolled back. This phenomenon can only be avoided if a transaction holds all its locks until it commits. If that is not the case, there is the chance that a cascading rollback is required if two or more transactions interfere.

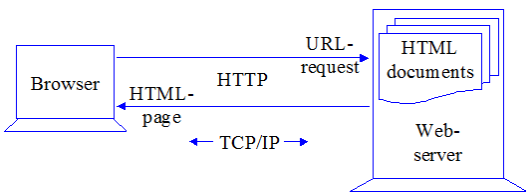
3. Web-database connectivity and database systems in an n-tier environment

3.1 The Web as client/server medium

3.1.1 Architecture of the World Wide Web

The World Wide Web was developed by chemist. They wanted to be able to share their researches.
The WWW contains 3 elements.

- **HTML** (HyperText Markup Language): specifies the **document format** used to write Web documents. A web document consists of a tagged text document, with the tags denoting the document's layout. HTML also features limited constructs for user input, e.g. buttons, checkboxes, input fields … in so-called HTML forms. A HTML form is a subset that allows entering data.
  Actually, HTML specifies how documents should be visualized. In other words, it expresses the **mark up** of a document, what should be italic, left, etc. It is all about the layout.
- **URL** (Uniform Resource Locator): The URL **uniquely** and **unambiguously identifies** all Web documents. The general format is: protocol://hostname/path
  Ex: or example: http://www.kuleuven.be/info/mydocument.html
  It is a uniform way to identify documents and specify their address. It is used to refer to other documents.
- **HTTP** (HyperText Transfer Protocol): this is the high-level protocol that is used for requesting and fetching Web documents, given their URL. HTTP builds upon (and hides the details of) the underlying **TCP/IP** network protocol stack.
  - **TCP/IP:** Low level set of protocols. It thinks in terms of bits and bytes.
    Ex**:** streaming data from a webcast, email
  - **HTTP:** High level protocol.



- **The (web) browser** is a tool that is able to visualize HTML documents, according to its specifications and it knows about HTML text.
- **TCP/IP:** low level protocol to exchange bits and bytes. This is what we call the internet.

- HTTP: high level protocol that thinks in terms of webpages. It transforms hypertext into the web.
- URL: unique identifier. It serves as indication where the document is positioned.
- The web server contains the HTML documents.

3.1.2 HTML forms

```
<h1>Order Form</h1>
<p>Please enter the following details: </p>
<form action="/cgi-bin/purchase.EXE" method="POST">
   <p>Product Name: <input type="text" size="20" maxlength="30"
   name="PName"> </p>
   <p>Order Quantity: <input type="text" size="20"
   maxlength="30" name="PQuantity"> </p>
   <p><input type="checkbox" name="Year">1999
   <input type="submit" value="Order now"> </p>
</form>
</body>
</html>
```

Begin tag: <>
End tag: </p>
Everything surrounded by <h1> should be visualized as header of type 1.
Visual aspects of the webpage
You can add inputs

3.1.3 The Web as platfomr for client/server computing

The **goal** is to define an architecture where end users with a webbrowser can transparently access database data, without worrying about underlying connection details and without having to install specific client software.

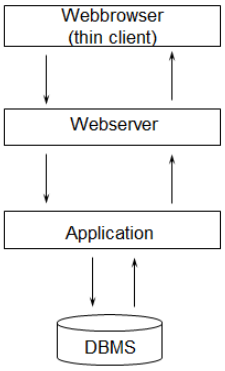
The above corresponds to the "**thin client**" principle, with client functionality being limited to accepting user input, possibly validating this input and presenting query results on screen. What you have on the client is nothing more than just a webbrowser. All core "business logic" is implemented server-side.

⇨ It can exist on tablets, smartphones or computers with a webbrowser.

3.1.4 Shortcomings of the Web for client/server computing

- HTML pages are **static** text pages. We need something dynamic and executable. HTML is not meant for database interaction.
- **Limited GUI capabilities** of HTML. If we want interactive web application, we might come up with some better capabilities. It is not possible to execute SQL queries. we want to get input from the user → HTML is very basic, not user friendly
  Ex: we need user interface capabilities this is not offered by HTML. → We don't have out of the box capabilities to execute executable code.
- The HTTP protocol is **not connection oriented** and stateless. HTTP is connectionless, we just send a request and get a result and that's it, no memory of what happens between different requests

3.1.5 General architecture for Web-based database access



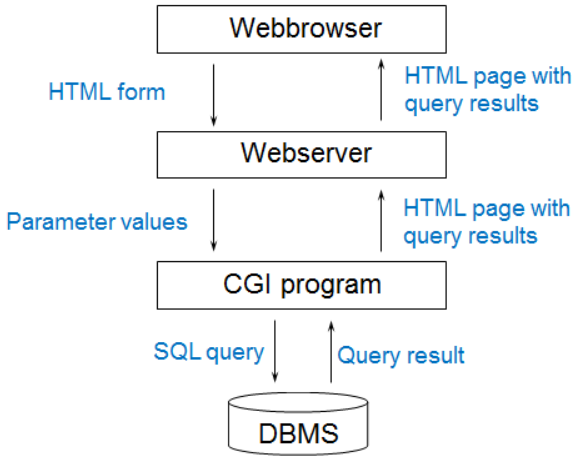
- We have to find a way to position executable code that interacts with the database (= application).
- The result of the query has to be transformed to a HTML page. It has to be a package in such a way that it can be visualized in between: we have a server and www. We will need an application that is able to query the database.

3.2 Executable code in a web environement

3.2.1 The common Gateway Interface (CGI)

CGI is an API that allows for executable code stored at a webserver to be activated from a web browser. The request is similar to a call to a static HTML page.

The CGI API provides a mechanism to pass parameter values, e.g. input by the user by means of an HTML form to the application. CGI provides a mechanism to pass parameters to a program. We can put a "?" with a set of parameters. The application can query the database, after which the query results are to be "packaged" as a standard HTML page, which is then returned for visualization in the browser.



Execution of a CGI program

CGI is low level programming, prone to errors. The **web browser** can send a request to the **webserver** with HTML forms being showed.
HTML form can have input fields behind the input, is a URL reference.
The webserver knows if the URL reference goes in the right directory. The CGI program is responsible for generating HTML that can be visualize by the web browser. It then returns a HTML page with query results. The, the output of the code (should be HTML) is returned by the web server to the web browser.

```
#include <stdio.h>
int main() {
printf("Content-type: text/html\n\n");
printf("<html>\n");
printf("<head>\n");
printf("<title>Hello, World example </title>\n");
printf("</head>\n");
printf("<body>\n");
printf("<h1>Hello, World</h1>\n");
printf("</body>\n");
printf("</html>\n");
return 0;
}
```

This is an example of a CGI program, written in C. This is a program to write “Hello world”. This page will be reported by the webserver to the web browser.

"The" advantage of CGI is that it was the first, universal, technology for accessing executable code from a web browser. However, it has some shortcomings:

- Complex parameter binding: it is not easy to program. It is easy to make mistakes and it would cause the program to crash.
- Each request requires an entirely new server process
- Performance & scalability
- Security

⇨ CGI is available to most programming languages.

3.2.2 Java applets

Java is an object-oriented programming language and real computing platform:

- **Portability**: "write once, run anywhere": Java source code is compiled into platform-independent byte code. This code can be run on any platform where a Java Virtual Machine (JVM) is installed. **Virtual machine**, principle that makes that JAVA can be executed in any platform.
- **Dynamic** code loading: "mobile code"
- **Security**: "untrusted" code is run in a ‘sandbox’
- **Database access**: JDBC, SQLJ, ...
- Web integration: applets, servlets, ...
- Distributed object computing & components: RMI and Enterprise JavaBeans (EJB)
- World wide deployment: a JVM exists in any web browser, webserver, database server or application server

Applets are pieces of Java byte code that can be downloaded and executed in JVM of a web browser, hence at the client side. Applet invocations are embedded in a page’s HTML code. The applet runs in the web page. A designated portion of the page is reserved for the applet’s user interface. Applets can use the entire set of Java GUI objects. They are not limited to HTML forms.

⇨ A Java applet is a small application which is written in Java and delivered to users in the form of byte code. The user launches the Java applet from a web page, and the applet is then executed within a Java Virtual Machine (JVM) in a process separate from the web browser itself.

Applet code:

```
public class HelloWorldApplet extends Applet {
   public void paint(Graphics g) {
   g.drawString("HelloWorld!", 50, 25);
  }
}
```

HTML code which embeds the applet:

```
<HTML>
  <HEAD>
    <TITLE> A Simple Program </TITLE>
  </HEAD>
  <BODY>
    <APPLET CODE="HelloWorld.Applet"
      WIDTH=150  HEIGHT=50>
    </APPLET>
  </BODY>
</HTML>
```

Example of an applet

The Java applet is executed on the web browser. It will also contain a Java Machine. WIDTH = 150 – HEIGHT = 50 refers to the part of the screen reserved for the applet.

Theoretically, it would be possible to directly access a DBMS from a Java applet. However, this would go straight against the "thin client" approach. On the other hand, although applet technology will not be applied for direct database access, applets will prove very useful to improve client functionality.

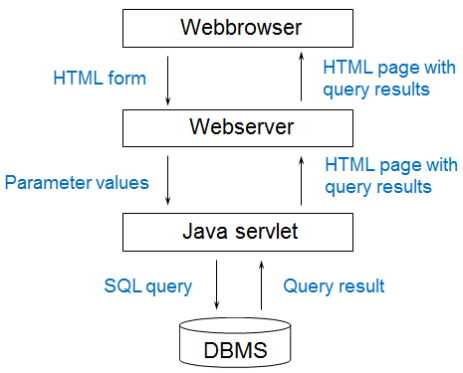
- To provide better GUI features in comparison to HTML forms;
- To receive and validate user input;
- To communicate with server side applications by means of a non-HTTP based interaction mechanism, cutting short the webserver.

3.2.3 Java Servlets

Java servlets are Java code that is run in the webserver and that we don't download into the web browser. For that purpose, the webserver should encompass a particular kind of JVM, called a servlet engine. It doesn't make the client becoming fat.

⇨ A **Java servlet** is a **Java program** that extends the capabilities of a server. Although servlets can respond to any types of requests, they most commonly implement applications hosted on Web servers. Such Web servlets are the Java counterpart to other dynamic Web content technologies such as PHP and ASP.NET.

The interaction mechanism strongly resembles the CGI based approach. Just like CGI programs, servlets should generate a standard HTML page in response to a request. In contrast to applets, servlets do not have a GUI.



- The web browser may contain HTML forms so that we can enter data that will be passed to the Java Servlet.
- Java servlet: files position on the web server. It is written in Java and can access the database. It has to make sure that it generates HTML.

```
ServletOutputStream  out = res.getOutputStream();
res.setContentType("text/html");
out.println("
   <HEAD><TITLE> SimpleServlet  Output  </TITLE></HEAD><BODY>");
out.println("<h1> SimpleServlet  Output  </h1>");
out.println("<P>Hello  World!");
out.println("</BODY>");
out.close();
```

Evaluation of servlets

- Servlets are platform-independent server-side programs, accessible through a universal API.
- Servlets remain active after handling a request; hence they are able to contain session information. It can keep an open database connection. It keeps on running. → More efficient.
- Multiple requests can be dealt with by the same servlet instance. These requests can be handled in isolation, i.e. in separate threads.
- Good performance in comparison to CGI programs.
- Excellent security: it is far more efficient in terms of security and errors. It is a good way to write simple programs.

Servlets	Applets
• Run on a **webserver**, so it can be used to write database programs. • Allows for only an HTML based GUI. • Can handle very **complex tasks**. ⇨ Don't provide user interface since they don't run on the client.	• Run on web browser • Can offer a very rich GUI • Can only handle fairly simple tasks. But they do it in a very nice looking way.

3.2.4 Client side and server side scripting

A **script** is a piece of code that is not compiled but that is interpreting. **Scripting code** is typically code that is interpreted rather than compiled. It can be embedded in a static HTML page, i.e. intertwined with the static HTML code. In that way, it can provide dynamic features to the HTML page.

Server-side scripts are pieces of code that are executed when the document is accessed on the server, i.e. before it is sent back to the client. In this way, certain page fragments can be defined as the result of a query, which is executed by the server side script. A static page is then returned to the browser. In terms of behavior, a server side script is comparable to a servlet.

Client-side scripts are executed when the document is received by the browser, hence on the client. This allows for a certain degree of dynamism at the client side. Client side scripts, although not as powerful, more or less behave like applets.
Here, we don't want to write an entire database application, otherwise, the client would be fat.
The webserver will do nothing. The browser will discover that there is executable code and will execute it.

> **Compile**: translate into executable code (bits and bytes)
>
> **≠**
>
> **Interpretable**: only have a source code and you don't compile it but when it is executed, it is done line by line.

- **JSP** ('Java Server Pages'): descriptive language used in the Java environment.
  - To be used if server side system is Java based
  - Compiled transparently into servlet for better performance

- **ASP** ('Active Server Pages')
  - To be used if server side system is based on Microsoft platform
  - Embedded in HTML or separate file
  - More recent: ASP.NET (For Microsoft's .NET framework)

- **PHP** ('PHP Hypertext Preprocessor')
  - Strong integration with open source technologies such as the mySQL DBMS and the Apache webserver

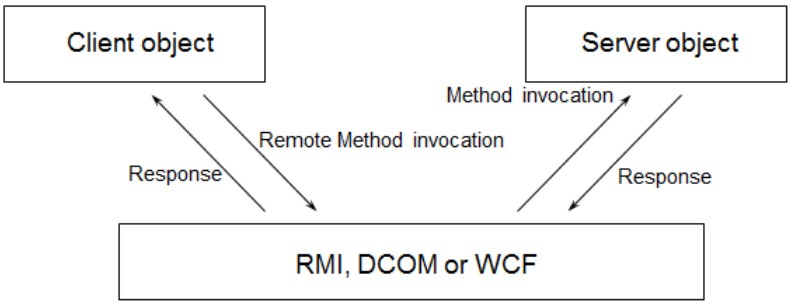
3.2.5 Distributed object architectures

⇨ This is not typical for the web. It is always used in the network environment.

The distributed object architectures supports method based interaction between objects on different hosts: the objects interact by remotely calling one another's methods. It only works if they are on the same machine. But if we install middleware then we can invoke methods over 2 different machines. Underlying connection details such as network "plumbing" are hidden from the application programmer; it appears as if they all run on the same host. It is not necessarily web based !

- Java environment: RMI (Remote Method Invocation)
  - Part of Java EE (Java Platform, Enterprise Edition), the Java component framework
  - Interaction between Java components (called "enterprise Beans")

- Microsoft environment:
  - DCOM (Distributed Common Object Model), earlier pre-Web approach
  - WCF (Windows Communication Foundation): more recent, part of Microsoft's .NET component framework



Distribute object architectures: principle

The client method can invoke a method on the server object. With a middleware, a method can be invoked from one machine to another.
Middleware = RMI, DCOM, WCF
Response: the return value is passed on the same trajectory to the client.

3.3 Client/server interaction on the Web

⇨ There are a lot of possibilities and combination of technologies

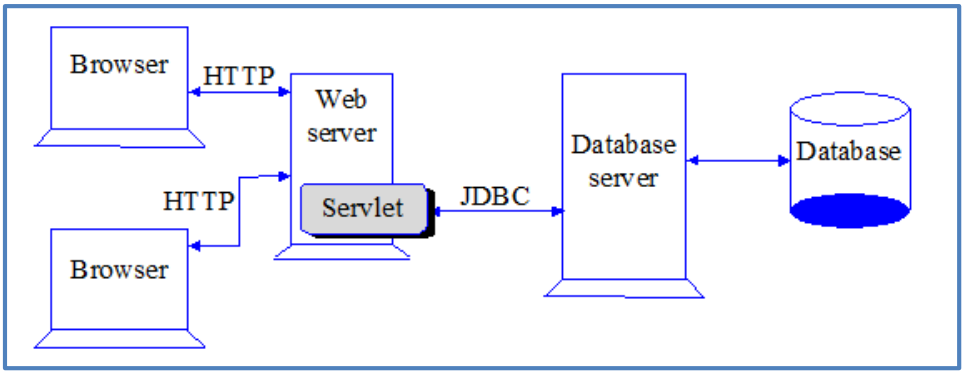
- Only executable code on the server
  - Through servlets and HTTP
  - Through server side scripts and HTTP
- Executable code in the browser and on the server
  - Through socket-to-socket communication and TCP/IP
  - Through distributed object interaction (Ex: RMI)

3.3.1 Client/server interaction by means of servlets

As a reminder, a Servlet is the special type of java class that produces dynamic web contents and develops web application. The Java Servlet API defines a standard interface for handling **request and response messages between the client and server**.

⇨ Servlets have access to the entire family of Java APIs, including the JDBC API to access enterprise databases.

Applications, those are accessed by a web browser using HTTP protocol are called Web Applications. Servlets and JSPs are used to development a web application.



The client, located in the **web browser**, sends a **request** to the **web server**, through **HTTP**. The user can only request static webpage from the server. The web server receives the HTTP request and forwards it to the respective Servlet. A Servlet is nothing but a Java class that processes business operations, builds a response and sends back to the web server. That response is dynamically built, and the content of the response usually depends on the client's request. Then, the web server sends the response back to the client. **JDBC** is used to interact with the database.

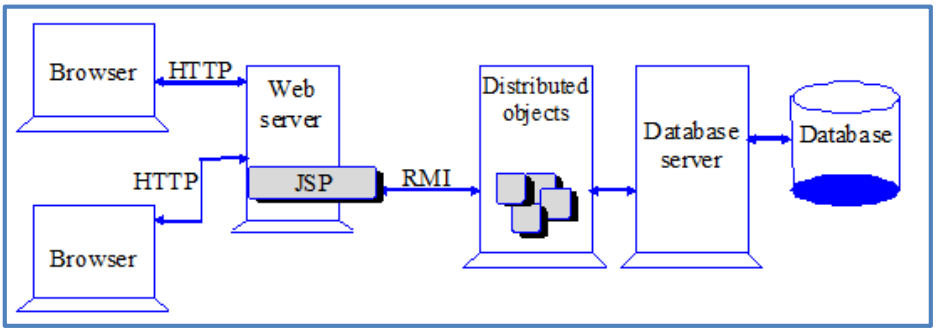
⇨ The basic idea of Servlet container is using Java to dynamically generate the web page on the server side.

EVALUATION

- The servlet is invoked just like a **static page**. It performs database access and generates a static HTML page with the query results. The interaction is entirely HTTP based.
- The client requirements are minimal. Indeed, you don't have to execute anything in the web browser.
  →Only a browser is needed. No applets are used.
- No client side "intelligence": input validation is to be performed server-side.
- The GUI is entirely HTML based
- Very simple, no firewall problems.

➔ A firewall is a wall around my own local network or computer that screens for values. It controls the traffic between my computer and the internet. It is a way to block viruses!

3.3.2 Client/server interaction by means of server side scripts



The client is the system on which the Web browser is running. JavaScript is the main client-side scripting language for the Web. Client-side scripts are interpreted by the browser.

The user requests a Web page from the server. The server finds the page and sends it to the user. The page is displayed on the browser with any scripts running during or after display.
JSP contains method calls to a java application.

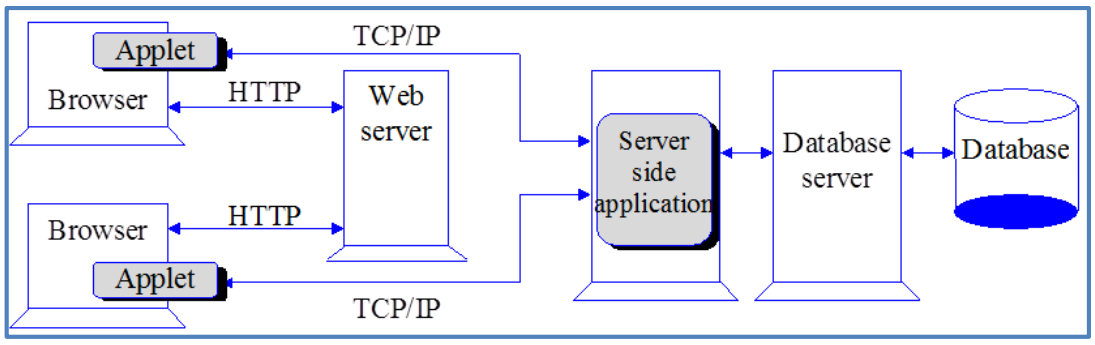
Normally, the server side script won't contain the Business logic. It will contain other applications which contain the actual business logic.

⇨ We don't install anything on the browser.

EVALUATION

- It is very similar to servlet based interaction: the communication between browser and script is still HTTP based.
- The server side code consists of a script, embedded in an HTML page. This script is responsible for calling upon distributed objects that implement the actual business logic and execute database queries. The query result is incorporated in the HTML page by the script.
- (+): separation of business logic and HTML layout.

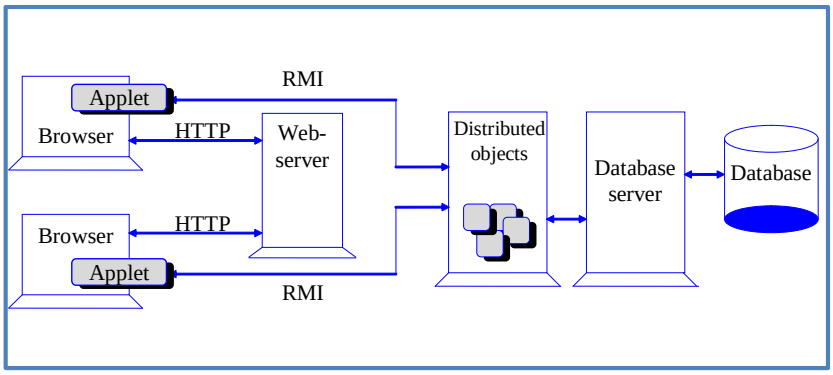
3.3.3 Client/Server interaction by means of socket-to-socket communication



EVALUATION

- Sockets define low-level TCP/IP connections. The client side and server side code interact trough these sockets, i.e. directly in TCP/IP instead of the much slower HTTP.
- The webserver is only used to download an initial HTML page with an applet. After that, the applet directly interacts with the server side application, without the webserver as intermediary.
- Interaction by means of variables, instead of entire HTML pages.
- (+) much faster than HTTP and session based.
- (-) very low-level, considerate amount of "plumbing" code required.

3.3.4 Client/Server interaction by means of a distributed object architecture



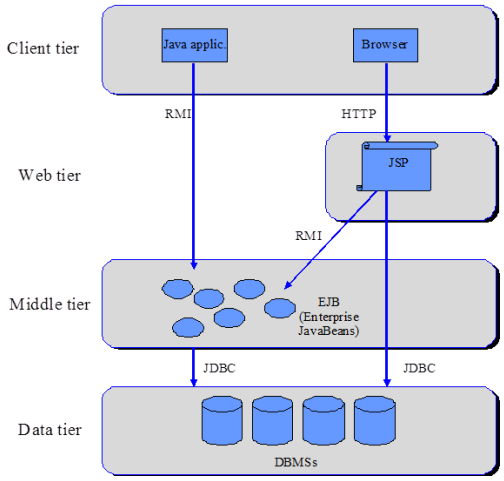
EVALUATION

- Objects in the applet and on the application server interact by remotely calling one another's methods by means of RMI.
- The webserver is only used to download an initial HTML page with an applet. After that, the applet directly interacts with the server side application, without the webserver as intermediary.
- Much higher level and less "plumbing" than socket-to-socket communication.
- RMI is used as high-level protocol instead of HTTP. Underneath, there is still TCP/IP based interaction going on, but these details are hidden from the application developer.
- Possible firewall problems with RMI, therefore only suitable on an intranet.

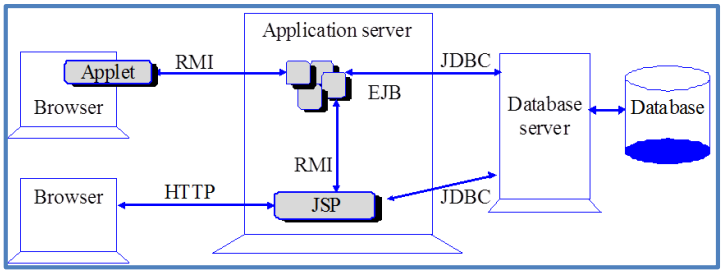
3.4 A global architecture for web-based database access

3.4.1 Java EE

⇨ Java Platform, Enterprise Edition or Java EE is a widely used enterprise computing platform developed under the Java Community Process. The platform provides an API and runtime environment for developing and running enterprise software, including network and web services and other large-scale, multi-tiered, scalable, reliable, and secure network applications.

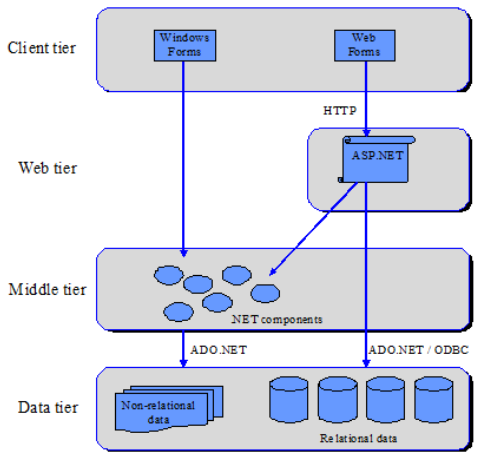


- Either client tier that contains a **java application** which contains user interface objects.
  Objects access database through JDBC.
- Web browser may contain an applet or not.



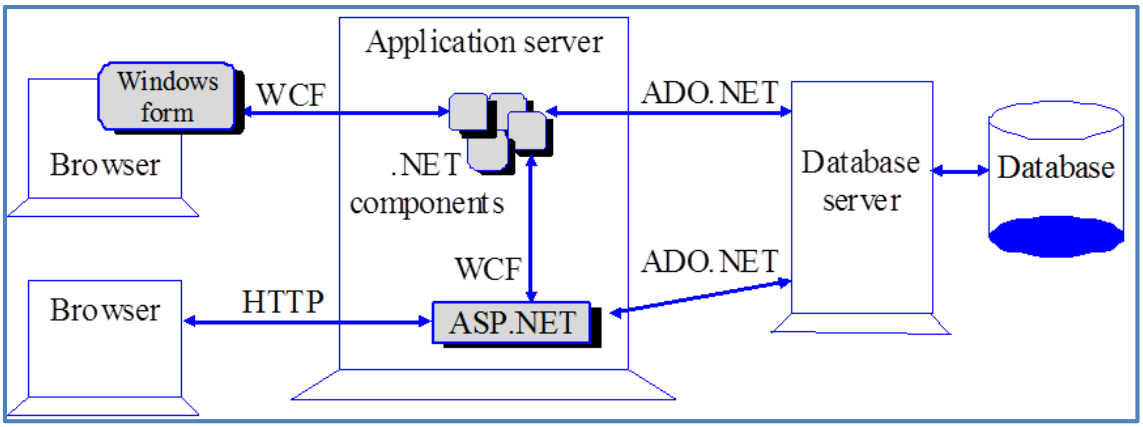
- The applet talks to the server application through RMI. Inputs are used by EJB to query queries in the database. Then, the result is retrieved.
  ➔ More efficient
- Either we have just a browser with nothing else. The browser will talk to the server side script in HTTP. If no business logic is involved then it will direct access the database. If it contains a business lgic, it will first access the EJB server through RMI, retrieve, then database.
  ➔ More suitable

3.4.2 .Net



2 possibilities

- Standalone program that contains windows forms.
- Web browser, accesses the server side script. Accesses business logic or info business logic, direct accesses the database.
  Microsoft .NET provides web forms, nice than HTML web forms.
  Data tier: it contains relation and non-relational data so we need more



- Either we have a normal browser that accesses the WCF and directly accesses the database.
- Either we have a browser with Windows form. It accesses .Net over the .Net middleware. The business logic will access the database through ADO.NET.

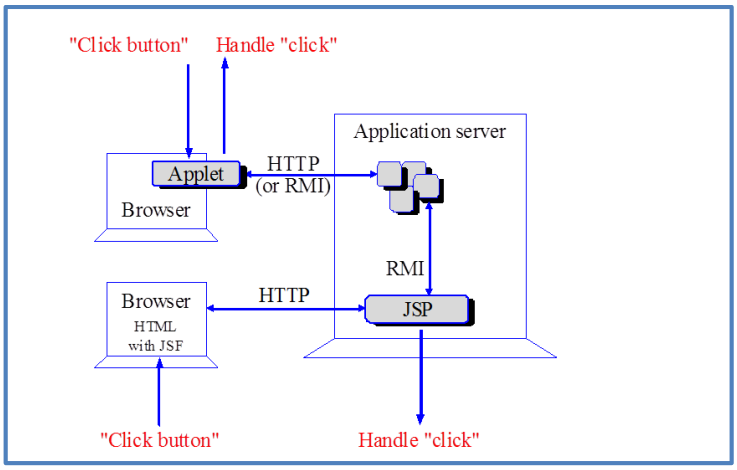
3.4.3 From thin clients towards Rich Internet Applications

⇨ Rich Internet Applications (**RIA**): web based applications with "rich" look and feel, similar to desktop applications

Two lines of technologies:

- **Browser based** (AJAX; 'Asynchronous JavaScript and XML'):
  - GUI functionality is split over browser and (web) server
  - No "refresh" of entire HTML pages, but rather of individual page fragments
  - Client side scripting combined with server side technology
  - XML based data exchange
  - Often used by GUI technologies that allow for handling GUI events server side (a.o. JavaServer Faces, Microsoft Web Forms)
- **Plugin based:** You won't extend functionality of the browser but just download piece of executable code.
  - Executable code downloaded and run in web browser
  - Rich, non HTML based GUI objects
  - Communication with server side business logic
  - Examples: Flash, Java applets, Microsoft Silverlight

Ex: email package such as Outlook. Part of Microsoft Office applications. You install but you can also run it on a web browser. This is a functionality that makes the user experience in web browser even as good as if you would run in on an application.



Goal: enhance the user interface

- With the **browser based approach**, we have a library based on scripting language. You enter data and click button. Then, the data is passed over HTTP as a fragment. The query result is then sent to the browser to be visualized on the browser.
- With the **applet approach:** full user interface object, if you click the button, it will be handled directly in the browser.

⇨ In both cases we have thin clients interfaces. However, the browser approach is thinner than the plug in approach.

3.5 Conclusions

- Access database data through web browser without installation of client side software
- Possibly transparent downloading of plugins into browser
- Different technologies with different distribution of GUI functionality over browser and server: not all thin client are equally 'thin'
- Business logic remains server side; different technologies for server side executable code
- Importance of separating page layout and GUI design from developing business logic