Theory guide

DBTech's Virtual Workshop on Distributed, Replicated, Embedded and Mobile Databases (Deliverable #28).

Antonio César Gómez Lora

Dpto. Lenguajes y Ciencias de la Computación

Universidad de Málaga

Introduction

Distributed, replicated, embedded and mobile databases are a huge area. It is not possible analyze with a minimal detail the main aspects of these fields in this simple workshop. This will need at least a complete course.

Here you can find a basic introduction on basic concepts concerning two first topics. We focus now on distribution and replication concepts, leaving embedded and mobile databases on a second part created by Tim Lessner (Reutlingen University/University of the West of Scontland).

We assume that you know what the difference between distribution and centralization is. If you know it, congratulations! Perhaps you can explain it to us.

All kidding aside, there exist many definitions of what a distributed database is, but no matter what definition you will use, there always be a real scenery that you catalogue as a distributed environment that will escape to your definition. In theory distribution and replication are simple concepts, but in practice the frontier is not clear at all.

Let's see what Wikipedia says today: someone has defined distributed database as:

"... a database that is under the control of a central database management system in which storage devices are not all attached to a common CPU. It may be stored in multiple computers located in the same physical location, or may be dispersed over a network of interconnected computers."

We can see the basic concept, but the frontier is not clear. This definition only approach data distribution and it is also too ambiguous, because it does not specify what a storage device is or what a CPU is or even what you consider as "attached". This definition allows us to consider Microsoft Access, OpenOffice Base and SQLlite as distributed databases, because although all of them are single-file databases, I can store a single file over different devices on different computers (combining, for example, RAID-0+iSCSI). It also allow me to consider 4 oracle 11g sharing information and processing over dblinks and also sharing information with other databases (as IBM DB2, Microsoft SQL Server, etc.) as a centralized database if all of them are installed on the same computer.

Now let's see Spanish version of the Wikipedia. Here someone has defined a distributed database as

"...a set of multiple logically related databases those are distributed on different logical spaces and interconnected by a communication network. Those databases have the capacity of perform autonomous processing, which allow to perform local or distributed operations. A Distributed database management system is a system where multiple database sites are linked by a network system in such a way that a single user on any site can access data on any network location as is they were accessed locally."

This definition is closer to the typical definition you can find on literature. In theory is a good definition, in practice it fails on one aspect. Many of the current distributed database systems, or those we can catalogue as distributed ones, verify the first and second sentences, but the last is not verified at all. Fully transparent distribution is clean and perfect in theory for a theoretical distributed database management system. But reality is dirty and imperfect and transparent data access (this means reading and/or writing) is not always possible. So we decide to be practical and leave distribution transparency as a desirable quality but not as a requirement.

But this is a workshop on distributed and replicated databases. But if we cannot define it clearly, what is this workshop about? The answer is that, although some global concepts are defined and treated, this workshop ant its hands-on-lab is focused on those systems that are constituted by multiple, logically related and interconnected relational databases, each one with autonomous processing capacity and being able to perform local and distributed operations. Note that we focused only on relational (and object-relational) databases. By logically related we means that it can exist some kind of relationships (being transparent or not for the user) among the objects of the database's logical layer on different databases. In relational databases logical layer usually refers to relational objects' layer. For example, the logical layer in Oracle consists on tablespaces, tables, clusters, indexes, views, stored procedures, triggers and sequences. We are going to avoid programmable objects (stored procedures and triggers) intentionally. Even more, we are going to consider only relationships among tables and views. We, also intentionally, have introduced ambiguity in how databases are physically interconnected; this means how these databases shared information or communicate among them.

Now let's start introducing our concepts.

Distribution

On databases there exist two types of distribution, based on the nature of what is being distributed. Traditionally we found:

- Data distribution. Appears when data is stored on different databases. In this workshop by data we means tables' and views' rows and columns.
- Process distribution. Occurs when processing is distributed among different databases. In our case we focused on process involving tables and views manipulation. This means process derived from DML sentences (select-insert-update-delete), and the mechanisms responsible of maintaining data integrity and transaction support.

Depending on the concurrence of Data and Process distribution in a system, we can define some types of systems.

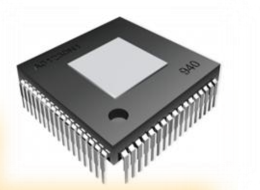
	No Process Distribution (single-site process)	**Process Distribution (multiple-site process)**
No Data Distribution (single-site data)	Centralized database	Parallel database
Data Distribution (multiple-site data)	Sometimes referred as homogeneous distributed database	Also known as fully heterogeneous distributed database

We must be careful with the names homogeneous and heterogeneous distributed databases. Here homogeneous and heterogeneous refer to processing and the type of system that can be implemented. Single-site process/Multiple-site data is defined as homogeneous distribution because all the processing is done by one database (the single process site). Multiple-site process/Multiple-site data can include systems where many different databases can share data and cooperate in the processing, even the type of these databases can be different (here appears the term heterogeneous).

Especially interesting are parallel databases and fully heterogeneous distributed databases, because both terms have also their own internal classification.

Parallel databases are systems where two or more database management systems work with the same database, exactly the same. Attending to how parallel databases share this database, or more exactly what type or resource they share, we can found the following classification:

- Share memory. If the different database management systems share main memory. This means that they are running on the same machine, usually a multiprocessor system sharing all or at least certain quantity of main memory, or they are running on different machines with main memory regions synchronized (commonly using specific protocols and hardware).



- Share disk. If the different database management systems share disk. This means that they are running on different machines sharing one or more disks.

- Share nothing. If the different databases management systems do not share either memory or disk.

For example, Oracle Parallel Server supports all three different types (share memory, share disk and share nothing).

Depending on how the shared resource is implemented by the hardware infrastructure, we can also talk about tightly coupled or loosely coupled. A multiprocessor machine where different processor can access the same memory module is a tightly coupled architecture. If two different machines use a communication bus or network to maintain two memory regions synchronized (like Digital's Memory Channel) we talk about loosely coupled architecture.

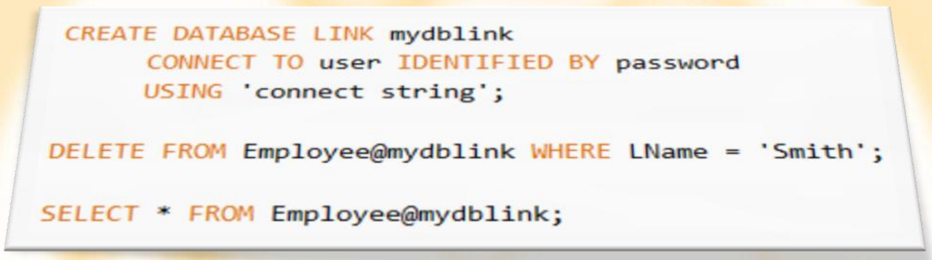
On the other hand, heterogeneous distributed databases can be classified based on the nature of the processing capabilities of each processing site or node. If each node is fully functional and completely autonomous the system is called a Federated Database Management System. If the constituting nodes are not completely autonomous, having limited processing capabilities, it is also called Non-Federated or simply a Multiple Database Management System.

Is there anybody out there?

Yes, as the song in The Wall (the masterpiece of Pink Floyd), the first step to introduce distribution in a database is to let it know that there exists other data sources out there, and the second step is to make the access to this external data possible. We only focus on access external data in other databases, although many databases are ready to access data from different types of sources.

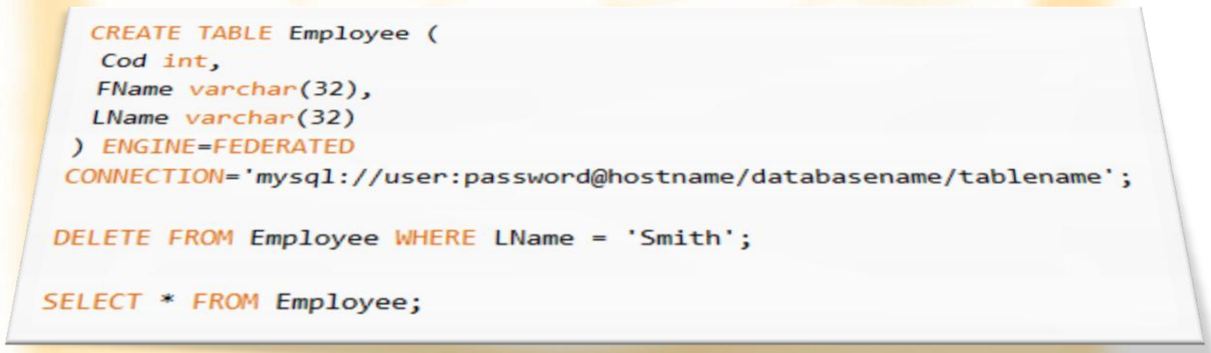
There not exists a single way to do this, but most databases introduces the concept of a reference to another database. Sometimes the reference or link is directly a usable object, being able to access practically any object available on this referred database. Other times it is not possible to link a database and the concept only allows the user to refer a particular object in the external database.

For example, Oracle uses the object Database Link to allow a user to access external data.



```
CREATE DATABASE LINK mydblink
      CONNECT TO user IDENTIFIED BY password
      USING 'connect string';

DELETE FROM Employee@mydblink WHERE LName = 'Smith';

SELECT * FROM Employee@mydblink;
```

My SQL can use the Federated engine, which gives you access to particular objects.



```
CREATE TABLE Employee (
 Cod int,
 FName varchar(32),
 LName varchar(32)
) ENGINE=FEDERATED
CONNECTION='mysql://user:password@hostname/databasename/tablename';

DELETE FROM Employee WHERE LName = 'Smith';

SELECT * FROM Employee;
```

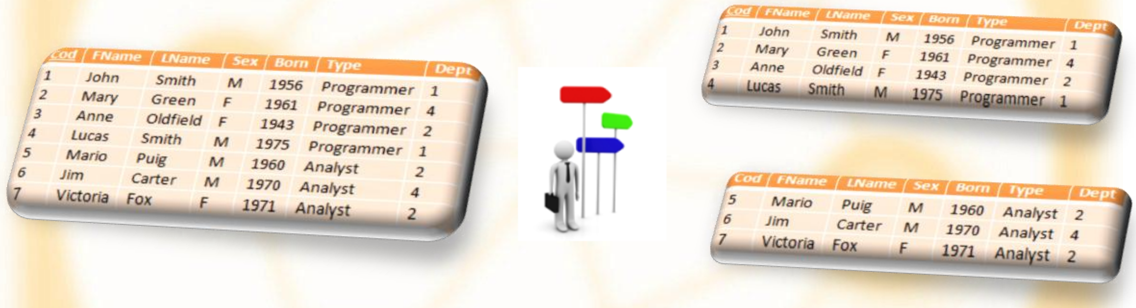
Both cases delete employees with last name 'Smith' and shows the resulting table. They apparently do de same. The only difference is that in Oracle everybody on remote database continues seeing Smith until the local transaction commits. But on MySQL everybody can see how smith disappears when local transaction performs the DELETE. Moreover if MySQL local transaction performs a rollback Smith continues deleted on remote table.

The answer: Oracle integrates the two commit phase protocol as its transactional model, MySQL Federated engine does not; of course, MySQL can perform two-phase commit using XA transactions (a specification of the X/Open group). This remember us that when using external references we must seriously consider that we are in a distributed environment, and if we does not use a transactional model specifically designed to cope with distribution we can easily lost the ACID property (Atomicity, Consistency, Isolation and Durability) in our operations.

Fragmentation

Typically data distribution limited to table contents is also called data or table fragmentation. There exist different kinds of fragmentations:

- Horizontal. It is present when different rows of a single table can be stored on different databases.
- Vertical. It is present when different attributes of the table are stored on different databases.
- Mixed. Occurs when we combine horizontal and vertical fragmentation.
- Derived. Also known as Horizontal Derived, it occurs when Horizontal fragmentation in a table is based on the Horizontal fragmentation or a parent table. It can be considered as a subclass of Horizontal fragmentation.



Cod	FName	LName	Sex	Born	Type	Dept
1	John	Smith	M	1956	Programmer	1
2	Mary	Green	F	1961	Programmer	4
3	Anne	Oldfield	F	1943	Programmer	2
4	Lucas	Smith	M	1975	Programmer	1
5	Mario	Puig	M	1960	Analyst	2
6	Jim	Carter	M	1970	Analyst	4
7	Victoria	Fox	F	1971	Analyst	2

Cod	FName	LName	Sex	Born	Type	Dept
1	John	Smith	M	1956	Programmer	1
2	Mary	Green	F	1961	Programmer	4
3	Anne	Oldfield	F	1943	Programmer	2
4	Lucas	Smith	M	1975	Programmer	1

Cod	FName	LName	Sex	Born	Type	Dept
5	Mario	Puig	M	1960	Analyst	2
6	Jim	Carter	M	1970	Analyst	4
7	Victoria	Fox	F	1971	Analyst	2

Horizontal fragmentation.- If we say that rows of a single table can be stored in different databases it means that two or more databases share the same table definition (same column count, same column order, same column names and compatible types), each one called a fragment. Each fragment stores a set of whole rows, and a row must be stored in, at least, one fragment.

Many databases sometimes offer a similar concept called table partitioning (sometimes indexes can also be partitioned) giving us a rich set of different types of partition techniques, based on keys, ranges, hash functions, lists, intervals, references and more. But we must note that, in many cases, table partitioning only works locally, not being able to separate different fragments on different databases. If this occurs then partitioning is not equals to fragmentation, but a particular local case this.

Fortunately, to implement fragmentation is not difficult. Complexity appears when we want to do maintain efficiency and ACID properties. It can be done transparently to some users when system includes the capability of programming view writing operations. Oracle allows this with the use of instead of triggers over views. PostgreSQL introduces a spectacular concept materialized in a SQL object called rule, allowing the interception modification operations not only on views, but also on tables, and even intercepting SELECT operations. As far as we known, on MySQL 5.5 and earlier versions there not exists a mechanism to allow user to programme how writings operations must be done using views. This does not mean that MySQL cannot exploit fragmentation; it only means that it cannot be done transparently to some users, just only inserting, updating or deleting on a table or view.

To ensure the regeneration of the whole table based on the fragments we need to consider two particularities.

- If the table has a key (no matter it is primary key or simple unique + not null).
- If one row can be stored in more than one fragment simultaneously, that is, there exists row replication.

Having this in mind we can regenerate the original table applying the following operators to all fragments. Remember that UNION removes duplicates and UNION ALL preserves duplicates.

	Key	**No Key**	
		No Duplicate Rows	**Duplicate Rows**
No Replication	UNION UNION ALL	UNION UNION ALL	UNION ALL
Replication	UNION	UNION	cannot regenerate

Remember that in the two cases where UNION/UNION ALL are indistinctly possible, that is, when we do not allow replication and the table has a key or does not have a key but duplicates never occurs, UNION ALL is preferable (UNION ALL is faster than UNION in practically all cases).

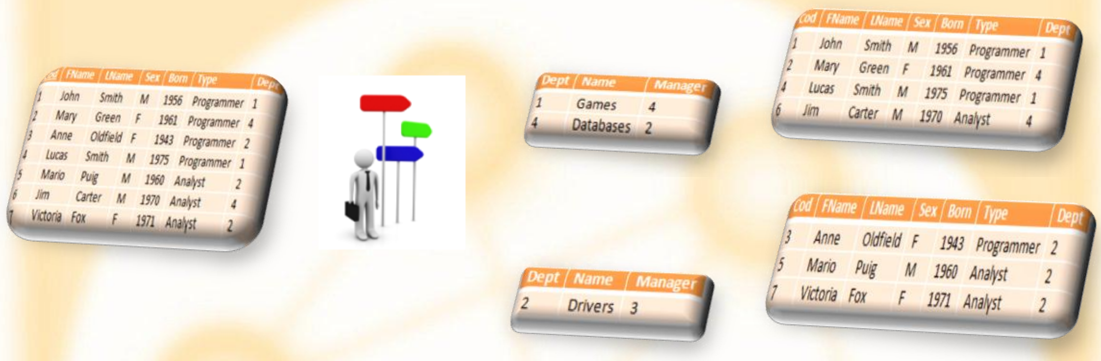
If we are in the worst scenario (no keys + possible duplicates rows + replication allowed) then horizontal fragmentation is not possible directly, because regeneration is not possible. The only way to obtain horizontal fragmentation here is to create a new table with the same definition than our original table plus a surrogate key (this new table can be horizontally fragmented with no problem). And redefine the original table as a view over the new table without the surrogate key.

If we consider the typical relational database constraints (primary key, foreign key, unique and check), then all valid constraints defined for the global table are also valid for fragments.

Another important aspect of the Horizontal replication is to determine the fragment or fragments that must store a row. Typically a predicate is associated to each fragment; if a row verifies the predicate of a particular fragment then this must store the row. The key factor is what we consider a valid fragment. Literature often uses the same kind of predicate or Boolean condition used in the selection operator of the relational algebra.

If the predicate is based exclusively on rows column values we can determine where a row is located if we known all these values. If any of these values is unknown, then we cannot determine the exact location and we need to perform a full scan among the candidate fragments.

But in real world Horizontal fragmentation is used with more complex predicates. One good example is the Horizontal Derived fragmentation. It is defined as a type of Horizontal fragmentation based on the fragmentation of a parent table. Parent table refers to the concept of mandatory one-to-many relation, being the one entity the parent table and the many entity the child table. On relational algebra this is translate to mandatory foreign keys, being the referred table the parent and the foreign key owner the child. On Horizontal Derived fragmentation a row is stored in the same fragments where the related row of the parent table is stored. This has some advantages; the main one is that it enables each fragment to regenerate one portion of the relationship among both tables, being able to regenerate the whole relation by a simple UNION of all this portions.



od	FName	LName	Sex	Born	Type	Dep
1	John	Smith	M	1956	Programmer	1
2	Mary	Green	F	1961	Programmer	4
3	Anne	Oldfield	F	1943	Programmer	2
4	Lucas	Smith	M	1975	Programmer	1
5	Mario	Puig	M	1960	Analyst	2
6	Jim	Carter	M	1970	Analyst	4
7	Victoria	Fox	F	1971	Analyst	2

Dept	Name	Manager
1	Games	4
4	Databases	2

Cod	FName	LName	Sex	Born	Type	Dept
1	John	Smith	M	1956	Programmer	1
2	Mary	Green	F	1961	Programmer	4
4	Lucas	Smith	M	1975	Programmer	1
6	Jim	Carter	M	1970	Analyst	4

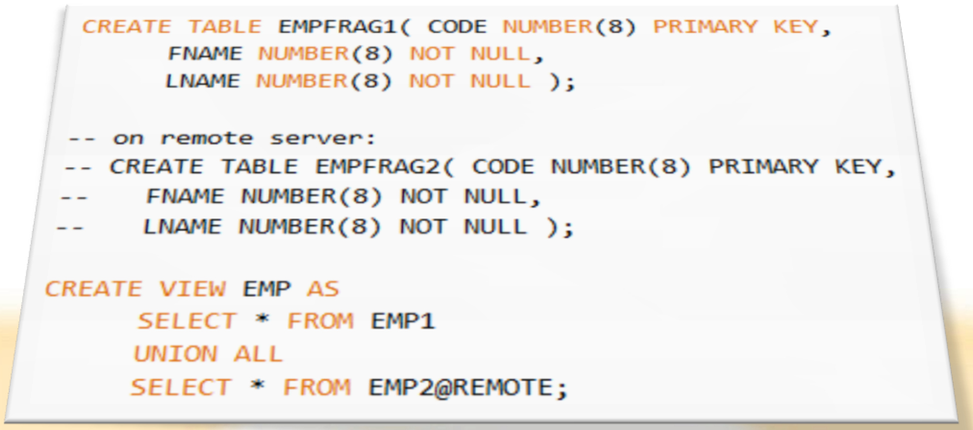
Dept	Name	Manager
2	Drivers	3

Cod	FName	LName	Sex	Born	Type	Dept
3	Anne	Oldfield	F	1943	Programmer	2
5	Mario	Puig	M	1960	Analyst	2
7	Victoria	Fox	F	1971	Analyst	2

More complex Horizontal fragmentations are possible and even useful. Horizontal Derived is based on the equality of a foreign key value in one table with its corresponding primary key value on the refereed table. But equality can be applied to two arbitrary attributes, not always a pair foreign-primary. This generalization is the base of table clusters, and its purpose is almost always the same: accelerate joins. This means that Horizontal Derived fragmentation can be generalized to store in the same location two fragments of different tables if they will be joined by a particular join operation. The only difference is that clusters can create many aggregations as different values has the cluster key; in Horizontal fragmentation we will need to use a hash function to limit the number of possible fragments.

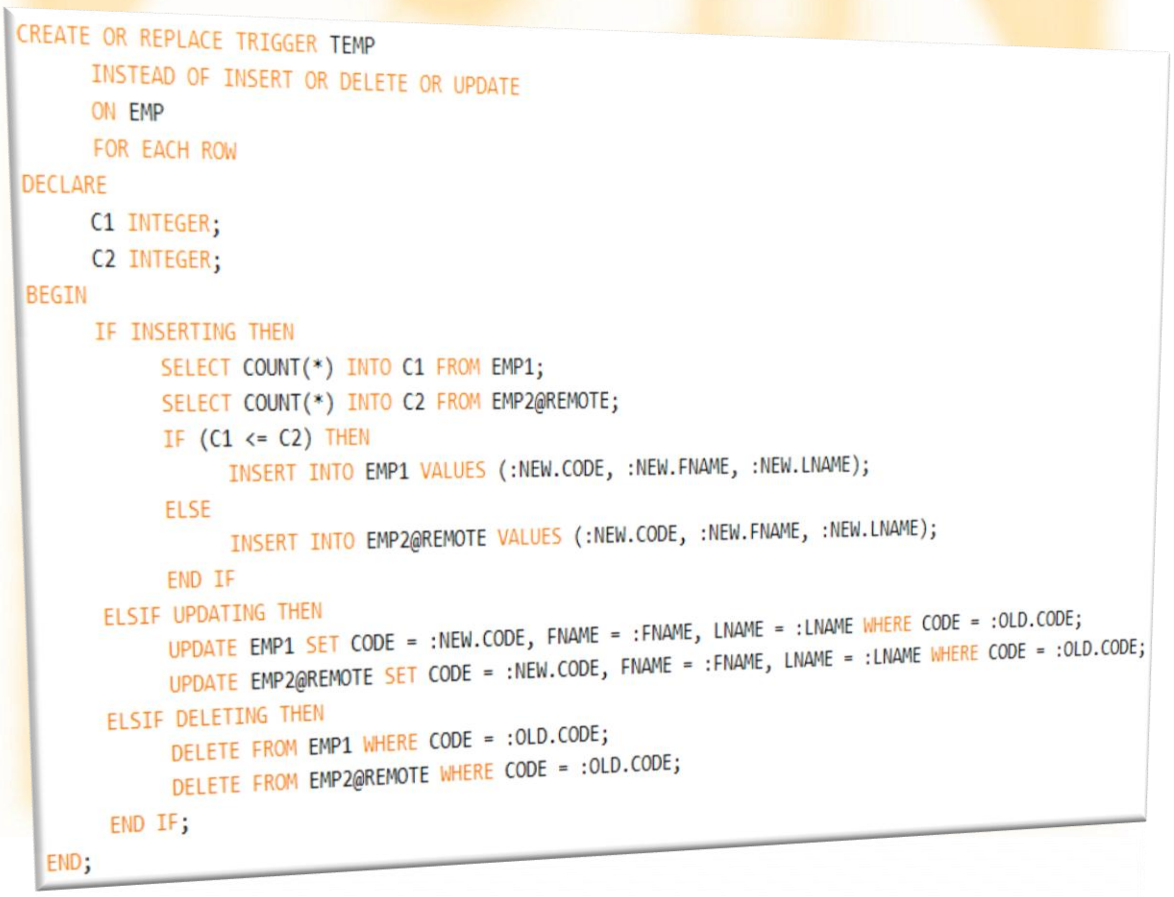
Moreover, Horizontal fragmentation can be further extended to use any arbitrary predicate. A frequently used technique in real systems to maintain uniformly sized fragments is to store a row in the less populated fragment, or use a round-robin based fragment assignation.

Let us see a simple and naïve example of how to force uniformly sizes fragments in Oracle (this example use a database link called REMOTE, previously create with CREATE DATABASE LINK sentence):



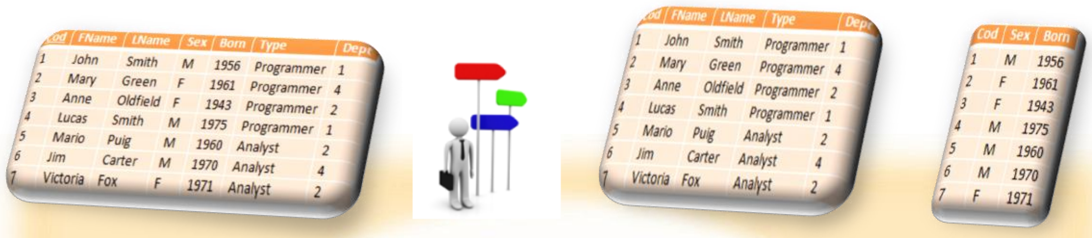
```
CREATE TABLE EMPFRAG1( CODE NUMBER(8) PRIMARY KEY,
       FNAME NUMBER(8) NOT NULL,
       LNAME NUMBER(8) NOT NULL );

-- on remote server:
-- CREATE TABLE EMPFRAG2( CODE NUMBER(8) PRIMARY KEY,
--     FNAME NUMBER(8) NOT NULL,
--     LNAME NUMBER(8) NOT NULL );

CREATE VIEW EMP AS
      SELECT * FROM EMP1
      UNION ALL
      SELECT * FROM EMP2@REMOTE;
```

And now the trigger:



```
CREATE OR REPLACE TRIGGER TEMP
      INSTEAD OF INSERT OR DELETE OR UPDATE
      ON EMP
      FOR EACH ROW
DECLARE
      C1 INTEGER;
      C2 INTEGER;
BEGIN
      IF INSERTING THEN
            SELECT COUNT(*) INTO C1 FROM EMP1;
            SELECT COUNT(*) INTO C2 FROM EMP2@REMOTE;
            IF (C1 <= C2) THEN
                  INSERT INTO EMP1 VALUES (:NEW.CODE, :NEW.FNAME, :NEW.LNAME);
            ELSE
                  INSERT INTO EMP2@REMOTE VALUES (:NEW.CODE, :NEW.FNAME, :NEW.LNAME);
            END IF
      ELSIF UPDATING THEN
            UPDATE EMP1 SET CODE = :NEW.CODE, FNAME = :FNAME, LNAME = :LNAME WHERE CODE = :OLD.CODE;
            UPDATE EMP2@REMOTE SET CODE = :NEW.CODE, FNAME = :FNAME, LNAME = :LNAME WHERE CODE = :OLD.CODE;
      ELSIF DELETING THEN
            DELETE FROM EMP1 WHERE CODE = :OLD.CODE;
            DELETE FROM EMP2@REMOTE WHERE CODE = :OLD.CODE;
      END IF;
END;
```

Of course, it is much more elegant not to try to update/delete both fragments, just only determining if it present using a previous select or determining if first update/delete has affected any row.



Cod	FName	LName	Sex	Born	Type	Dept
1	John	Smith	M	1956	Programmer	1
2	Mary	Green	F	1961	Programmer	4
3	Anne	Oldfield	F	1943	Programmer	2
4	Lucas	Smith	M	1975	Programmer	1
5	Mario	Puig	M	1960	Analyst	2
6	Jim	Carter	M	1970	Analyst	4
7	Victoria	Fox	F	1971	Analyst	2

Cod	FName	LName	Type	Dept
1	John	Smith	Programmer	1
2	Mary	Green	Programmer	4
3	Anne	Oldfield	Programmer	2
4	Lucas	Smith	Programmer	1
5	Mario	Puig	Analyst	2
6	Jim	Carter	Analyst	4
7	Victoria	Fox	Analyst	2

Cod	Sex	Born
1	M	1956
2	F	1961
3	F	1943
4	M	1975
5	M	1960
6	M	1970
7	F	1971

Vertical Fragmentation.- The capacity to store some attributes of the same table in different fragments is not an original idea of distributed databases. It is a frequent technique even in centralized systems, for example to separate the storage area of huge attributes (Large Objects like images, documents, videos, etc.) from the main table storage area that maintains "small" attributes.

The purpose of vertical fragmentation is to break the table vertically and treat each resulting fragment as a new table. How implement fragments and how to regenerate can be discussed.

Visually, in vertical fragmentation, each fragment contains a subset of the columns set, and each fragment will contain the same number of rows. The problem is how to regenerate the original table in this approach. As we are thinking visually the solution is to maintain the same table order for all replicas and use this order to regenerate the original table. This is a problem in most database servers, because maintain arbitrary orders inside a table storage is not a frequent feature.

The most frequent approach is to include on each fragment the table key, this means that tables with no keys cannot be directly vertically fragmented. But as in horizontal fragmentation this can be solved by adding a surrogate key. If we include the table key on each replica then the regeneration is easy, we only have to join all the fragments using the replicated key. A projection operator can be needed to regenerate the original column order, especially when a vertical fragment includes non consecutive columns of the original tables.

We can see that vertical fragmentation introduces a kind of replication: one table key is replicated. But we only consider vertical fragmentation with replication when one or more attributes, different from that key, are introduced in two or more fragments.

Mixed fragmentation is the simultaneous combination of vertical and horizontal fragmentation on the same table.

Replication

We have talked about some kind of replication when we described horizontal and vertical fragmentation. But now we address replication under the perspective of whole table replication or even whole database replication.

On databases, replication concept means to store something in at least two different locations. In this workshop we focused on the idea that a table that is stored simultaneously on two or more different databases or even two or more different databases that stores exactly the same objects.

So, the replication of a table on two or more databases means share the same table definition (same number of columns, same order, same column names and compatible types) and introduce a mechanism to maintain synchronized the table data among different replicas.

On real world it is common to replicate whole databases. Maintain replicated databases do not only means to synchronize data, but it also means to synchronize the data definition (the schemas). Traditionally a set of fully or partially replicated databases is called a database cluster.

Depending on who is authorized to introduce changes in data and/or schemas in the cluster, we can find two different types of architectures:

- Master/Slave. Only on replica, the master (sometimes called subscriber) is authorized to introduce changes, slaves only replicate this changes.

- Multimaster. Any member of the cluster can introduce changes. The cluster must propagate any changes made by any member to the rest of members. It must also prevent or resolve conflicts when two different members try to modify simultaneously the same data.

Of course, reality is much complex than this definition. It is possible to create mixed architectures:

- Clusters where there exists more than one master, but there also exists slaves.

- Master/Slave or Multimaster does not necessarily refer to the entire cluster, it can refer to a piece of data; in one cluster a database can be the master for one table and slave for another one.

This gives us an impressive variety of possible cluster configurations.

On the other hand, we attending on when a master notifies or propagates the changes to others we can find two types of systems:

- Synchronous. When the master notifies the changes before the finalization of the operation that produces the changes. The notification is considered as a part of the modification itself.

- Asynchronous. When the master notifies the changes after the finalization of the operation, being able to introduce more changes concurrently or even before the propagation of the previous modification.

Although there exist many ways to implement this type of propagation mechanisms, two basic types are commonly used:

- Log based. When we use the log file of a database to capture modifications and transfer them to the other members.
- Trigger based. Triggers are included in the master's table to notify the changes.

Log based are, by its nature, asynchronous mechanisms. Trigger based can implement both synchronous and asynchronous mechanism.

Now let's see some examples:

- Oracle: directly implements asynchronous master/slave using a log-based mechanism. Oracle database cluster: implements multimaster, synchronous using two phase commit and asynchronous using a deferred transaction queue.
- MySQL: implements multi-master, circular replication and master/slave. It implements synchronous a synchronous mechanism and an asynchronous (this last one available since MYSQL 5.1.6). Replication is implemented using three threads: binlog dump thread in the server and slave I/O and SQL threads in the slave.

- PostgreSQL: built-in master/slave asynchronous since version PostreSQL 9.0. Synchronous multimaster using PGCluster and pgpool-II, asynchronous master/slave using slony-I, Bucardo, Londiste, Mammoth and rubyrep. Bucardo and rubyrep also implements asynchronous multimaster.

When there is more than one master in a cluster, conflicts arise. What happen if two masters change or try to change the same data simultaneously? There are two possible solutions:

- Pessimistic techniques. When a mechanism try to avoid the apparition of a conflict it is called pessimistic. Commonly, when a master changes something it will try to lock or reserve in some manner the other master's replicas.

- Optimistic techniques. Also known as lazy replication, it does not try to avoid a conflict, but it try to repair the problem when conflict occurs.

Transactional models

We assume that you know what transactions are, what are their pros and cons, and how it can be implemented in traditional (centralized) databases. We assume that you also known the concepts of commit, rollback, save point, deadlock and lock.

Now we are going to introduce different transactional models.

Flat.- This is the simplest transaction model. A flat transaction has no internal structure, it only can start and finalize (with commit or rollback).

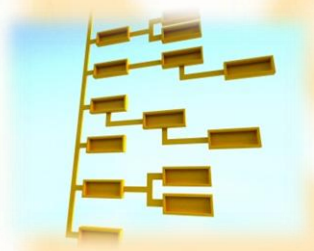
Begin	...	...	...	...	...	...	...	...	**End**

Flat with Savepoint.- Flat transaction is extended to allow the saving or partial changes, called savepoints. In this kind of transaction changes grows monotonically with successive savepoints. This is the transactional model Implemented by most database management systems.

Begin	...	...	SP1	...	...	SP2	...	...	**End**

Chained. This is an extension of the savepoint concept. A transaction is now seen as a sequence of simple transactions. Subsequent transactions can see the modifications made by committed chained transactions. Rollback affects only the active transaction. After a system crash chained transactions preserve their work, something that savepoints cannot do. Other relevant difference between savepoints and chained transactions is that the commit of a chained transaction releases their locks, improving concurrency efficiency.

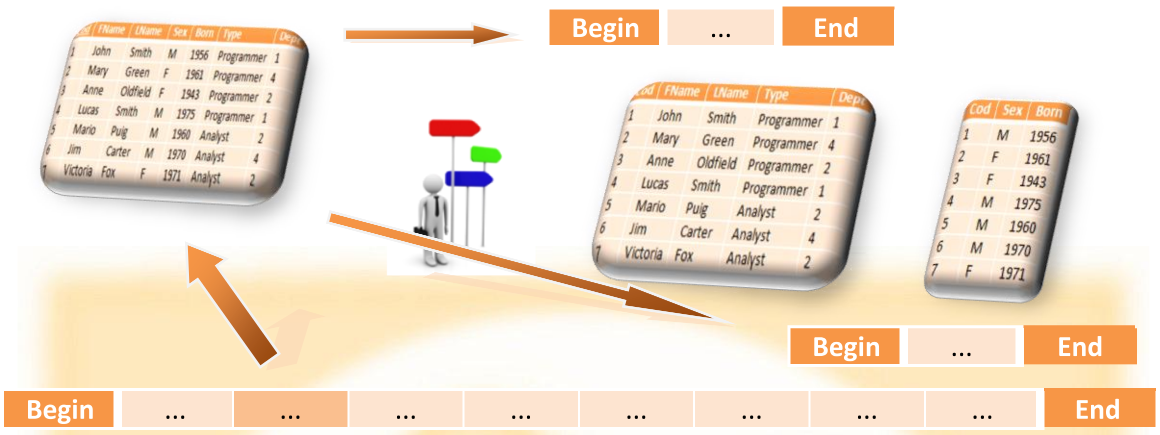
Begin	...	...	End/Begin	...	...	End/Begin	...	...	**End/Begin**



Nested.- Generalization of savepoints. Nested transactions have a hierarchical structure; internal transaction can also be nested transactions. Internal transactions can be rolled back without affecting the external transaction. Commited sub-transaction is not published until its parent transaction also commit, this means, nothing externally visible until root transaction commits.

Begin	...				...				**End**
		Begin	...	End		Begin	...	End	

Distributed.- Flat transactions that runs on distributed environments. When the transaction access data on other node (not locally) it needs to create a subtransaction in this remote node. If it access data on multiple nodes multiple subtransactions appears. This structure is similar to nested transaction, but the hierarchy is not controlled by the application, the topology of data distribution implicitly creates this hierarchy. Another important difference with nested transaction is that distributed subtransactions are tightly coupled with its parent. If a subtransaction commits or rolls back the main transaction also commits or rolls back.



Two phase commit.- A distributed algorithm designed to allow different nodes of a distributed database to reach a consensus on whether to confirm or to abort a distributed transaction. As its own name indicates it has two phases:

- Commit request phase. The transaction coordinator requests a commit of all participants. Depending if their respective operations have ended properly, each node will respond to the coordinator saying yes or no, voting to commit or to rollback.
- Commit phase. If everyone votes to commit, the coordinator will send a commit message to all participants. Each participant will commit its local part and confirms the coordinator that it has committed. When all participants have confirmed their commits, the coordinator completes the transaction. If someone vote to rollback the coordinator can decide performs the previous process sending a rollback message.

The main inconvenient is that this is a blocking algorithm. If coordinator performs the commit request phase but fails before commit phase starts, then all participants will be blocked (permanently if coordinator fails permanently).

If a participant fails during request phase (its receive the request, but fails before sending its vote), coordinator will be blocked, but in this case, if coordinator establishes a timeout limit, it could decide to abort the transaction and send an abort to the rest of participants.

Tree two phase commit.- A variant of the two phase commit. The coordinator is the root of the tree, when it requests the commit all its direct descendants also propagate the request down in the tree structure. When an intermediate node collects all the votes of its descendants decide its own vote and send it back to its parent. The commit phase proceeds in an equivalent way.

Dynamic two phase commit.- A variant of the tree two phase commit. There not exists a predefined coordinator on the tree. When a leaf node finishes its works it sends its agreement message (its vote). When an intermediate node receives all its agreements except the last one it sends its agreement to this last neighbour. The message will collide naturally in the last node, this will be the coordinator. Sometimes the messages collide in one edge; in this case one (any) of the two nodes connected to this edge is selected as a coordinator.

This variant increases the speed of the tree two phase commit.

Multi-level.- Based on nested transactions. Here a nested transaction can perform an early commit, called pre-commit, publishing data before the parent transaction commits, but also creating a compensating sub-transactions that will undone all the changes pre-commited by the commited subtransaction. If the main transaction finally wants rolls back, it only needs to commit the compensating transaction. This is a violation of the ACID principle.

Open-Nested.- Multi-level without any control. No matter if the parent transaction commits or rollbacks, subtransaction can be commited or rolled back independently. This means subtransactions are also top-level transactions.

Long-Lived.- When transaction exceeds the limit of what we consider a single database transaction, long-lived transaction appears. This type of transactions looks for atomic property grouping sequences of database transactions.

Sagas.- A chained model that introduces compensating transactions to be able to pre-commit its results. This is made by allow transactions under certain circumstances request access to resources being locked by other transactions, allowing owner transaction to force a pre-commit and releases the object in favour of the requesting transaction. Of course, requesting transaction cannot modify the object and must release in favour of the original owner when finished. A compensating transaction is used to return de database to a consistent state

Workflows.- This is not a type of transactional model, but it is a substitute of transactional model in certain scenarios. It uses workflows instead of transaction to maintain consistency. Workflow is not a traditional database term so it is difficult to define from a transactional perspective. With this in mind we admit this definition:

A workflow can be defined as an activity with open nested semantics that:

- Publish partial results
- Uses compensating actions on atomic and isolated sub-actions.
- Uses contingency actions on non-atomic sub-actions.

Real examples

This workshop has an important practical part. So we are going to introduce two real examples that will be used in the hands on lab.

As we have a limited space/time to introduce theory and practice, we cannot extend as much as we would like. So, among all possible topics that we can stress on the hands on lab, we have decided to select replication and the use of virtual machines to study this kind of systems. I have talked with my current students, with some old ones (now graduated and working with "real" databases) and with some colleagues about what topic would be the best choice (if a best choice exists). A distributed oriented transactional models hands on lab was considered too hard. Fragmentation was better accepted. But the most accepted topic among my students and colleagues was always replication. I don't know why but replication was always associated with raid systems, server racks and supercomputers, making replication attractive and a motivating scenario.

Although we can test replication on a centralized system (using a single databases or two database servers installed on a single machine), this is not a motivating approach for students.

So the use of virtual machines to test replication was our glamorous solution to motivate students to get into the world of distributed and replicated databases.

As we have justify on pre-workshop material, in the hands on lab we have decided to use MySQL Cluster and PgPool-II + PostgreSQL.

MySQL Cluster

MySQL implements cluster using NDB storage engine. NDB storage engine requires the use of two different services: the NDB Manager and the NDB services.

NDB Manager is the services that will coordinate the cluster nodes. Each node will have one MySQL server and one NDB Service.

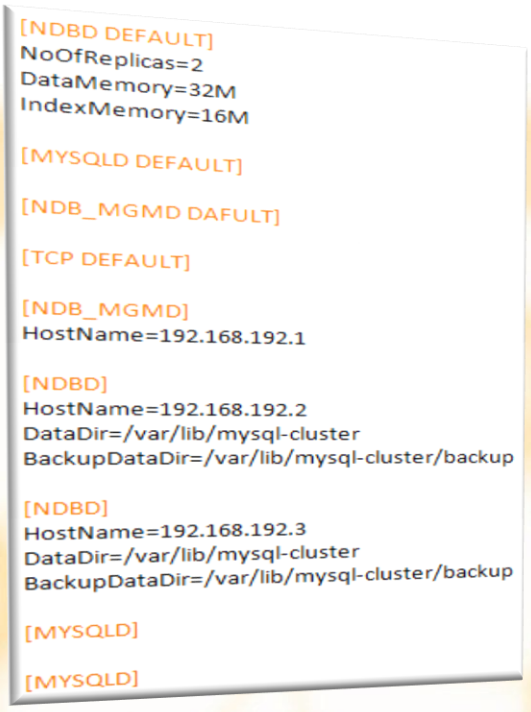
The NDB Manager Service has its configuration file, called ndb_mgmd.cnf in Debian (and located on /etc/mysql directory).

This file must define among others the following sections:

- NDBD DEFAULT.- This section configures the global values of the cluster.
- MYSQLD DEFAULT, NDB_MGMD DEFAULT, TCP DEFAULT.- This sections loads the default values for the cluster manager and network protocol.

- NDB_MGMD.- Configuration of the cluster manager.
- NDBD.- One section for each replica; configuration of each replica.
- MYSQLD.- One section for each replica; configuration of MySQL on each replica.

An example contents for this file is:



```
[NDBD DEFAULT]
NoOfReplicas=2
DataMemory=32M
IndexMemory=16M

[MYSQLD DEFAULT]

[NDB_MGMD DAFULT]

[TCP DEFAULT]

[NDB_MGMD]
HostName=192.168.192.1

[NDBD]
HostName=192.168.192.2
DataDir=/var/lib/mysql-cluster
BackupDataDir=/var/lib/mysql-cluster/backup

[NDBD]
HostName=192.168.192.3
DataDir=/var/lib/mysql-cluster
BackupDataDir=/var/lib/mysql-cluster/backup

[MYSQLD]

[MYSQLD]
```

This example declares that we will have 2 replicas, 32 MB of RAM will be used for data and 16 MB for indexes.

The host 192.168.192.1 as the manager server and two storage nodes: 192.168.192.2 and 192.168.192.3. On both the data directory and optionally backup data directory are declared.

MySQL servers on different nodes must be configured to use the NDB service and to known where NDB manager is located. This must be done by including

```
ndbcluster

ndb-connectstring = manager _server_ip
```

At the end of [mysqld] section in the MySQL configuration file, called my.cnf in Debian (located in /etc/mysql directory). It is also needed to include the following section in this configuration file:

```
[MYSQL_CLUSTER]

ndb-connectstring = manager_server_ip
```

With this configuration you only need to specify engine=ndbcluster when creating a table on any replica to have it automatically replicated on all storage nodes on the same database.

Remember to use a database that exists on all replicas.

PgPool-II + PostgreSQL

PgPool-II is an external replication layer. We will configure in our hands on lab a master-slave configuration. PgPool-II will act as the master, and two PostgreSQL on storage engines will act as servers.

PgPool-II configuration is file is called on debian pgpool.conf (located in /etc directory on PostgreSQL 8.3 version). This file defines many parameters; we are interested especially on three:

- listen_adresses = 'ip_addresses'. Determines on what IP PgPool-II must listen (remember that a system must have many network interfaces). Use '*' to accept requests from all interfaces.
- replication_mode = true. Command PgPool-II to enter in replication mode.
- load_balance_mode = true. This parameter determines if PgPool-II must balance the load on its PostgreSQL servers.

This file must also define the storage nodes, in our example:

```
backend_hostname0 = '192.168.192.2'

backend_port0 = 5432

backend_weight0 = 1

backend_hostname1 = '192.168.192.3'

backend_port1 = 5432

backend_weight1 = 1
```

We see that there exist three parameters per node: the hostname, the port and the weight ratio (a number that specify the proportion among server computation capabilities). This last parameter allows to mixes weak and strong servers, or it allows to liberate or to overloads specific nodes.

On each PostgreSQL server you must configure in its configuration file postgresql.conf (located on /etc/postgresql/8.3/main for PostgreSQL 8.3 version) the ip where postgreSQL server listen. Here again use listen_addresses='*' to indicate any IP device.

PostgreSQL requires that you define how users are authenticated. Here a configuration file called pg_hba.conf (located on /etc/postgresql/8.3/main for PostgreSQL 8.3 version). In our example we will allow users called postgres of any machine in our sample network to connect without authentication. This is done including the following line:

```
host    all     postgres      192.168.192.0/24      trust
```

This means trust postgres user when trying to access all databases (any) from an IP address in the range 192.168.192.0 to 192.168.192.255.

PostgreSQL Servers are not automatically replicated; both servers can work autonomously. To use replication you must work through PgPool-II server. On version 8.3 and under Debian it uses by default 5432 port. PgPool-II uses 5433.

So if you want to test replication you must connect to PgPool-II server (it will act as a master PostgreSQL server) and here create your own schemas. These schemas will be replicated on slave serves. All data modified will be immediately replicated.

Slaves can introduce their own information in replicated schemas, it will be only viewed locally, but it can produce cluster inconsistencies.