Distributed databases

- I largely follow Silberschatz (not the latest edition) while adding info from Elmasri-Navathe, Connolly-Begg and (to a large extent) my own experience.

- A distributed database consists of loosely connected nodes in a network

- They do not share any physical components and thus end up in the class “share nothing” among distributed computer systems

- The database systems that run on each of the nodes are independent of the other DBSs

- What they share is the conceptual database model and data management while logically they are separate DBSs that co-operate by allowing transactions to touch more than one of the nodes.

- (Leslie Lamport) A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.

Distributed architectures – Shared nothing?

Usually referred to as “massively distributed” systems, not to be confused with “massively parallel computers”.

“Shared nothing” systems consists of massive numbers of complete computers, often connected with an internal network for operational speed-up.

The stored information is partitioned and spread over the computer systems involved and the real database is the union of all databases in the network.

Optimal when data are local (as when data are distributed in the same way as the enterprise). Else less efficient than “shared disk”.

Very large databases sometimes use “shared nothing” systems as it is possible to use “smart” strategies for replication in order to keep the system available and fast, largely without interruptions.

If one or more (depending on number of copies and distribution strategy) computers are down there are still enough data copies to answer queries and when repaired nodes become available they are quickly and efficiently updated from online nodes.

Databases on Shared Nothing systems range up to Petabyte in size (1 PB = 10^15 Byte)

Distributed architectures – Shared disk

Loosely connected architecture optimized for centralized applications with demands on high availability and very high performance.

Each CPU has its own primary memory but all CPUs share secondary memory (can access all secondary memories)

You avoid the shared primary memory bottleneck and you don’t have to use extra program overhead to manage physically partitioned data.

Data security is often upheld by using RAID technology.

Faster than this is not possible (unless you use supercomputers) but other demands than speed exist and may make you choose another architecture.

These are the systems normally referred to as cluster systems, even if the term irregularly is used for other distributed architectures . . .
### Distributed architectures – Shared memory

Shared memory is a tightly connected architecture where a number of processors share system memory.

Usually referred to as “symmetric multiprocessing” (SMP) and has become popular on a range of systems, from simple PCs up to large RISC systems and even to the largest systems.

Hard to beat in speed but scales only up to 128 processors. Beyond that the internal bus is a bottleneck (but the limit slowly moves upwards thanks to NUMA, NonUniform Memory Access, that takes care of memory access so that access competition ends and race conditions are eliminated).

Most operating systems supports at least more than one processor, Windows and MacOSX 16 (?), Linux 64 and Solaris UNIX 128.

### DDBMS – what is gained?

- Organisational conformity. Many organisations spread their operations to more than one location and there are great benefits in mapping the same structure onto the information infrastructure.

- Local autonomy. By mapping the organisational structure onto the database structure, each part of the organisation enjoy local access to data that is directly associated with local activities and only communicate with the rest of the database when necessary. You may also allow the local operation to have local control over its own data and let the overall system DBA be the overall co-ordinator.

- Increased availability. In centralized systems even the simplest error may render the system inaccessible. In a distributed system performance suffer but the system as a whole is still available.
DDBMS – what is gained . . .

• Increased reliability. Using a smart replication strategy, the system may be fully operational even if some nodes are inaccessible.

• Increased performance. Distribution and local “proximity” to data makes data access fast and at the same time you have a high degree of parallelism as computations may be shared among close-by nodes.

• Better economy. It is cheaper to add to an existing DBMS than to buy a new, bigger machine when growing out of the existing one.

• Modular growth. It is not only cheaper to grow. You can add a new node without halting the system (which is operational during upgrade).

DDBMS – drawbacks

• Complexity. DDBMS is more complex than a centralized system. In addition, security is often managed by data replication which increases the complexity of DB changes. E.g. an update must ensure that all copies of a data item are updated before being used again.

• Cost. Increased complexity increases cost.

• Security. It is harder to maintain security in a decentralized system. You must keep track of all copies of data items and the network traffic makes the system vulnerable. Also you are forced to have complicated updating strategies.

DDBMS – drawbacks . . .

• Integrity control. All copies of data objects must have correct values at every time that they are to be used or must copies be unavailable until corrected.

• Standardization. There is no fully accepted standard for distributed databases. Each DDBMS provider has his own opinion of how the system must work, which sometimes makes it hard to change to another product.

• Database design becomes more complicated as you may be forced to take fragmentation and replication into account already at schema design.

Homogeneous DDBMS

In a homogeneous distributed database

• all nodes have identical DBMSs

• all nodes have knowledge about all other nodes and accept co-operating with them

• all nodes partially give up their autonomy and allow co-operating nodes to make certain updates to schemas and software

• each node behaves towards clients as if it was a single DBS (= clients don’t notice that the DBS is in fact distributed)
Heterogenous DDBMS

In a heterogenous distributed database, sometimes referred to as a federated database

- may different nodes have different schemas and different software (DBMS)
- do the differences in schemas constitute a problem when querying the system
- do the differences in software constitute a problem in transaction management
- might nodes be unaware of other nodes
- might some nodes allow only limited co-operation in transactions

Distributed data storage

Most systems today are relational (RDBMS) or extensions to relational (ORDBMS) systems

Suppose that this is the case and that co-operation is maximized. Then

- the system will keep more than one copy of a single data object, spread over the system nodes. This is referred to as replication. Replication is used to increase speed and fault tolerance.
- tables will be divided into partitions or fragments, that will reside on different nodes. This is referred to as fragmentation.
- replication and fragmentation may be combined so that the system keeps more than one copy of a fragment spread among the nodes.

Data replication

- A relation or a fragment that is redundantly stored on more than one node is replicated.
- If a relation has copies on each node in a system it is fully or totally replicated
- A database is fully redundant if each node has a complete copy of the database
- A DDBMS may have different strategies for different DBs

More on data replication

- Advantages:
  - Availability – a missing node means performance loss but not data loss
  - Parallelism – a query may be split into subqueries that execute in parallel
  - Proximity (reduced network traffic) – smart strategy for copy location may render local queries in many cases
- Deficiencies:
  - Update cost increases as all copies must be updated
  - Complicated concurrency control – consistency not simple to obtain. All strategies have shortcomings
Fragmentation

- Fragmentation means partition of relations. If table $t$ is divided into fragments $t_1, t_2, \ldots, t_n$, then the fragments must contain enough data to allow the recreation of $t$.

- Horizontal fragmentation – each row of $t$ exist in at least one of $t_1, t_2, \ldots, t_n$ so that $\bigcup_{i=1}^{n} t_i = t$

- Vertical fragmentation – the schema for $t$ is divided into subschemas so that $\bigotimes_{j=1}^{n} t_j = t$

Fragmentation – advantages?

- Horizontal fragmentation
  - enables parallel processing of table fragments
  - enables locality – by placing redundant tuples on the nodes where their use is most frequent

- Vertical fragmentation
  - enables locality – by placing redundant parts of tuples where their use is most frequent
  - enables rapid join operations through the use of pseudo keys
  - enables parallel processing of entire tables

- Vertical and horizontal fragmentation can be used together and fragments may be fragmented

Data transparency

- Users must not have to be concerned about
  - how data is fragmented,
  - how data object copies are spread over the DDBS or
  - where specific data objects are stored

- The distributed database system shall behave towards the client as if it was a centralized system

Transaction management

- Transactions may include objects from more than one node
  - Each node maintains a local transaction manager that
    - keeps one or more log files for local recovery and
    - manages local concurrency (in reality participates in \ldots as it communicates with other transaction managers)

- Each node has a transaction coordinator that
  - starts distributed transactions that are initialized on the local node,
  - distributes partial transactions (subtransactions) to suitable nodes and
  - coordinates commit/abort to all the subtransactions that have been started by the transaction that initialized on the node
Transaction management...

What can go wrong in a DDBMS?

- Loss of a node (server crash)
- Message loss (communication problem)
- Network failure (might be necessary with redirection in systems with many communication links)
- Partitioning (stupid name . . . ) meaning that a network is split in two without communication between the two networks
- Generally difficult for the surroundings to distinguish between server crash and partitioning

Commit

- A commit protocol is used to guarantee ACID properties (Atomicity, Consistency, Isolation, Durability) in DDBMS to ensure that
  - a distributed transaction must either commit on all involved nodes or abort on all nodes
  - no transaction may commit on one node and abort on another
- 2-phase commit is the most frequently used but
- 3-phase commit also exists, is more complicated and demands more resources but avoids, by splitting the second phase in 2-phase commit, minor problems that might occur in the 2-phase commit protocol
  I have seen it used on one installation (which has reverted to 2-phase commit)

Commit...

- To ensure proper processing of the commit protocol then nodes that encounter an error must stop immediately and take no action (not even send an error message). Usually referred to as the “fail-stop model”
- Note that the commit protocol has nothing to do with the locking protocol where the denotation “2-phase” also exist.
- The transaction coordinator starts the commit algorithm at the end of a transaction and involve all nodes where participation in the transaction resulted in an update on the node.
Commit in the large

2-phase:
- phase 1 Decide whether commit is possible
- phase 2 Perform commit or abort

3-phase:
- phase 1 Decide whether commit is possible
- phase 2 Prepare and report (OK or abort)
- phase 3 Perform commit or abort

Thus, in 3-phase commit local decisions are collected to help the coordinator to make a final decision län koordinator ta ett slutgiltigt beslut.

Commit in the large, 2-phase/3-phase . . .

- $q_c \rightarrow w_c$ Send commit request to all nodes
- $q_i \rightarrow w_i$ Send OK from node $i$ to coord
- $q_i \rightarrow a_i$ Send abort from node $i$ to coord
- $w_c \rightarrow a_c$ abort from at least 1 node, send to all
- $w_i \rightarrow a_i$ received abort from coordinator
- $w_c \rightarrow c_c$ OK from all nodes – commit to all nodes
- $w_i \rightarrow c_i$ commit from coordinator

2-phase commit . . .

- Assume transaction $T$ on node $N_i$ with coordinator $K_i$
- $K_i$ tells all node transaction managers to prepare for commit (all nodes that update something in $T$)
  - $K_i$ writes a record in the log file, $<\text{prepare } T>$ and forces write to secondary memory (not always easy, different optimization strategies for write in different OS’es).
  - $K_i$ send “prepare $T$” to all nodes involved in $T$
- Each node that receives “prepare $T$” message verifies if commit of $T$ is locally possible
  - Force write of $<\text{ready } T>$ on log
  - Force out all updated records to secondary memory
  - Send $<\text{ready } T>$ to $K_i$
- and else
  - Force write of $<\text{abort } T>$ to log
  - send “abort $T$” to $K_i$
Node failure

All errors that a transaction may encounter (se slide 25) may also occur during commit:

If node $N_k$ was disconnected it will check its global log and if it finds

- \(<\text{commit } T>\) run: redo ($T$)
- \(<\text{abort } T>\) run: undo ($T$)
- \(<\text{ready } T>\) consult $K_i$ and if $T$ performed commit run: redo ($T$), $T$ performed abort run: undo ($T$)
- nothing then $N_k$ crashed early and $T$ must have aborted: run: undo ($T$)

Coordinator failure

Means that $K_i$ disappeared during execution of commit.

All transaction managers must agree and decision is taken in the following order

1. If any participating node has a $<\text{commit } T>$ then all must commit and otherwise
2. if any participating node has an $<\text{abort } T>$ then all must abort and else
3. if any participant does not have a $<\text{ready } T>$ then the coordinator failed before deciding so decide abort and finally
4. if all nodes have a $<\text{ready } T>$ but nothing more, then all nodes must wait for $K_i$ to recover prior to deciding.

If all nodes wait until “timeout” a blocking is assumed to have occurred. intråffat. Corresponds to a dead-lock in the locking protocol

Network partitioning

- If all participating nodes are in the same partition it’s OK to continue
- If not
  - Nodes in partitions where $K_i$ is missing run according to the coordinator failure protocol. It's OK (may mean “wait”).
  - Nodes in the same partition as $K_i$ assumes that those that do not answer have failed and run according to the usual commit protocol. (OK)

Transaction management . . .

Transactions run, apart from what has been noted above, according to the same protocol as centralised system but because of the distribution (or rather, the replication) you may need a slightly different ground for decisions regarding locking

- One lock manager in the system: All subtransactions send lock requests to the lock manager that issues a lock on the appropriate nodes (one extra message). One message to release the lock. Simple dead-lock detection – the one used in centralized systems is good enough but there are performance and vulnerability issues.
- Distributed lock manager: All nodes have a local lock manager and participate in building a global “conflict graph” / “precedence graph” / “serializability graph” as a kind of merge of all local conflict graphs. Complicated dead-lock detection and management.
- Primary copies: For each data object a decision is taken on which copy that is considered to represent the data object per se. All locks are issued on the primary copy. Vulnerable protocol. As far as I know none uses it any more.
Transaction management...

- **Majority protocol.** A certain number (often half) of the existing copies must be successfully locked to consider that a lock exists. Complicated to implement, complicated dead-lock detection and management.

- **Weighted protocol:** Shared lock is requested only on the local copy but exclusive lock is requested for all copies. Simple when reading but complicated for updates, tricky dead-lock detection.

- **Quorum Consensus:** Issue a weight on all nodes and let $S$ be the total weight for all nodes. Choose one quorum for read ($Q_r$) and one for write ($Q_w$) such that $\sum Q_r + \sum Q_w > S \land 2 \times \sum Q_w > S$

  Each read must lock enough copies that the sum of all weights $\geq Q_r$

  Each write must lock enough copies that the sum of all weights $\geq Q_w$

- **Time stamping:** Collect all time stamps for all copies of participating objects. Compare and issue a global time stamp and require that no local copy time stamp differ too much in time. If a local time stamp differs too much, exclude from update and correct later.


Transaction management...

Many DDBMS support a protocol called the “weak consistency protocol”. It means “lazy evaluation”.

All such systems require time stamps that are used a little like “better use before” stamps in the grocery store.

A transaction is considered to be successful if you successfully update all primary copies. Then the updates propagate to all copies “as needed” or “at once” (which is not always all that direct) or “in due time”.

The propagation after a successful transaction has nothing to do with the transaction but is managed by a stand-alone process.

This means the transaction may read any copy after checking the time stamp but that update are performed either on a primary copy or on a copy acting temporarily as primary.

This in turn means that the database as a whole may never reach a fully consistent state even if all local databases have local consistency as there might exist updates waiting to propagate.


Transaction management...

In distributed transaction management all queries must be transformed to (split into) a number of local queries and the answer to the query be a merge of all answers.

The strategy depends on course on the strategy for replication and fragmentation. In the worst case a sequence of unions and joins. But network traffic is the major cost.

Optimization requires additional parameters:

- **Locality:** Is there a local object or do we have to issue a distributed subquery? Is it cheaper to make a local copy? Are many of the objects on another node?

- **Cost:** for network traffic to move objects, to move the execution or to distribute execution?

- **Performance:** Is there a gain in allowing several nodes execute the same (sub-)query? (Is it possible to earn on using parallelity)

Instead of join (doesn’t matter what kind of join so I choose natural join) you may perform a semijoin ($\bowtie$ or $\bowtie$).

If you are to execute $r_1 \bowtie r_2$ where $r_1$ with schema $R_1$ resides on node $N_1$ and $r_2$ with schema $R_2$ resides on node $N_2$, then, instead, you perform $r_1 \bowtie r_2$:

1. calculate $s \leftarrow \Pi_{R_1 \cap R_2} (r_1)$ on $N_1$

2. send $s$ from $N_1$ to $N_2$

3. calculate $t \leftarrow s \bowtie r_2$ on $N_2$

4. send $t$ from $N_2$ to $N_1$

5. calculate $r_1 \bowtie t$ which is equivalent to $r_1 \bowtie r_2$

In reality statistics are used to calculate which of $r_1 \bowtie r_2$ and $r_1 \bowtie r_2$ is the best alternative depending on expected network traffic.
Some notes on security

- Authentication
  - Check if the user is who he/she claims to be (worth a closer look)
  - Check data correctness

- Authorization – who may do what and under what circumstances?

- Keep data confidential

- Data integrity

- Transaction confirmation

Authentication

Mainly, you generate a key from a password (or passphrase) in one of a million ways. Relies on calculations that are rapid in one direction but with a reverse that is intended to be impossible or at least too time consuming.

- DH (Diffie-Hellman): Generate numbers, \( X \) and \( Y \) from the password or one of them from a public key and calculate \( X^Y \). Or cut the password in two and consider the parts as \( X \) and \( Y \) and calculate \( X^Y \). Virtually impossible to find \( X \) and \( Y \) from the encrypted password.

- RSA: Find two primes departing from the password and multiply them. Impossible to factorize in reasonable time and very hard to find the password that generated the two primes.

- Secret/public keys
  - Secret keys are distributed from special key distribution centers like Kerberos
  - Public key often as certificates

The last three deal with cryptography. Data is encrypted with an extra crypto for confirmation that transfer was OK (digital signatures)

The second alternative – authorization – is managed by assigning roles when authenticated. It may be performed in many ways.

Authentication is common and can be quite a pain regarding all the systems on the market.

Authentication by back-end database

- Check if the collected info matches some info in the back-end database

- Such a database is referred to as a realm

- A realm may contain user names, encrypted passwords, roles . . .

- Organisation and maintenance depend on OS and type of technology LDAP, RDBMS, standard file, PAM, Windows AD
Authentication via web (Tomcat)

You collect information for login in a dialogue window that the browser provides. Not secure so use encryption (SSL)

Tomcat can manage several realms,

- Standard file (default)
- RDBMS (via JDBC)
- LDAP (Lightweight Directory Access Protocol)
- Loads of other technologies via LDAP (e.g. PAM, Plugable Authentication Modules, de facto standard on Linux/Unix)

Default in Tomcat

Find the Tomcat installation directory and in its configuration subdirectory (config)

config/tomcat-users.xml

Unencrypted it is simple to set up but completely insecure so use only locally for testing or use e.g. Apache as a proxy. Fairly simple to configure for maximum security.

```xml
<?xml version='1.0'?>
<tomcat-users>
    <role rolename="manager"/>
    <role rolename="employee"/>
    <role rolename="admin"/>
    <user username="serafim" password="srfmpwd"
          roles="manager,employee"/>
</tomcat-users>
```

Default in Tomcat...

Then tell the web container that you use basic authentication

In web.xml for each application

```xml
<web-app>
    ...  
    <security-constraint>...</security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>realm-name</realm-name>
    </login-config>
    ...  
</web-app>
```

Default in Tomcat...

Then add the URLs that ought to have access control enabled

```xml
<web-app>
    ...  
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>WRCollection</web-resource-name>
            <url-pattern>/loadpricelist</url-pattern>
            <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>admin</role-name>
        </auth-constraint>
    </security-constraint>
</web-app>
```
Tomcat ... access controlled URLs (SSL)

```
<user-data-constraint>
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name></realm-name>
</login-config>
...