Distributed Databases

Centralised vs. Distributed Databases

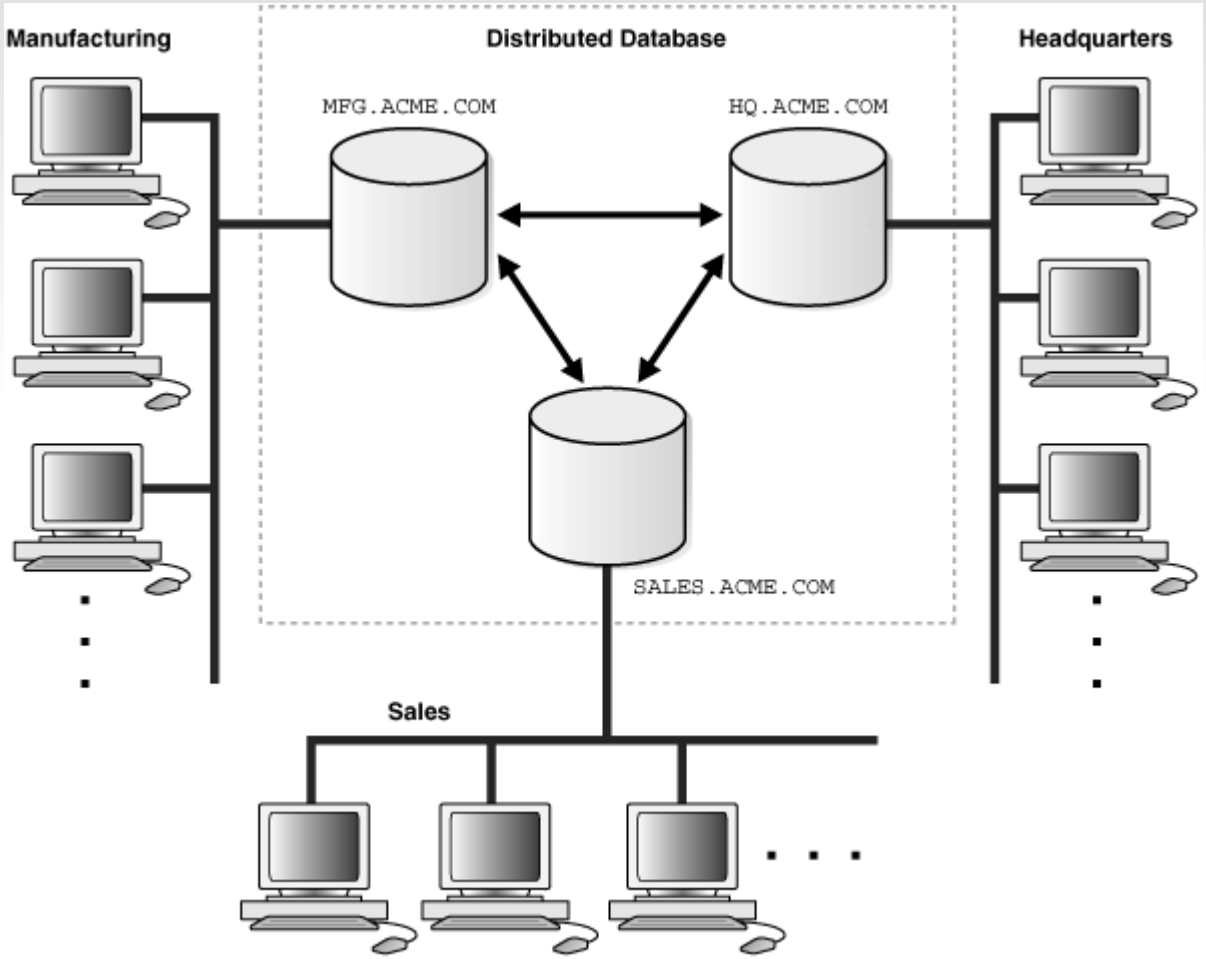
What is a Distributed Database?

- The DB is stored on several computers, interconnected through a network.
- Each site can process **local transactions** involving data only on that site.
  - Each site has a local transaction manager
  - Each site has a transaction coordinator
- Some sites may get involved in **global transactions**, which access data from several sites.
  - Local transaction managers communicate to achieve this.

Examples

- University of the West Indies
  - Bursary - Financial information
  - Personnel - Staff information
  - Registry - Student information
- Multinational
  - HQ in Kingston
  - Manufacturing in Trinidad
  - Warehouse in Miami
  - European HQ in London
  - Each site keeps data on local employees, as well as data relevant to the operation of the site.

Distributed Database Architecture Example



Why Distributed Databases I

- More natural for representing many real world organizations.
- Local autonomy
  - Local organization is fully responsible for accuracy and safety of its own portion of DB.
- Improved performance
  - Speed up of query processing
    - Possibility of parallel processing
  - Local processing
    - If data must be accessed often at a site, store it locally.

Why Distributed Databases II

- Improved availability/reliability
  - DB can continue to function even if some subsystems are down if we replicate data or hardware.
- Security
  - Avoid destruction of DB by replicating vital data.
- Incremental growth
  - Increasing size of DB
  - Increasing operations
    - Example: Include R&D facility.

What do we want from a distributed database?

- **Allow users to use the system as if it is not distributed.**
- Leads to following objectives:
  - No reliance on central *master* site.
    - Would be a bottleneck.
    - If down, whole system is down.
  - Continuous Operation
    - Enforces reliability and availability
  - Distributed Query Processing
    - A query may require accessing data at a no. of sites.
    - Same data may be available at a no. of sites.

What do we want from a distributed database?

- Distributed Transaction Management
  - Same copy of a data item may be a a number of sites.
- Transparency
- Local Autonomy

Transparency

- *Hiding* the distributed design from the user.
  - Network or distribution transparency
  - Replication transparency
  - Fragmentation transparency
  - Hardware and OS transparency
  - Language and DBMS transparency
    - Applies mostly to multidatabase systems.

Autonomy

- All operations at a site are controlled by that site.
- Types of autonomy
  - Design
    - Individual DBs can use data models and transaction management techniques that they prefer.
  - Communication
    - Individual DBs can decide which information they want to make accessible to other sites
  - Execution
    - Individual DBs can decide how to execute transactions submitted to them.

Issues and Problems I

- Distributed database design
  - How should DB and applications be placed across sites.
  - How should data dictionary be placed across sites.
  - Replication and partitioning
- Distributed query processing
  - How to break down a query into series of data manipulation operations.

Issues and Problems II

- Distributed concurrency control
  - Gets worse if data are replicated
- Distributed deadlock control
- Reliability
  - If site becomes inaccessible, how do you ensure that DBs at other sites remain consistent and up-to-date?
- Heterogeneous databases
  - Sharing pre-existing DBs with different conceptual schemas.
  - Also called multidatabase systems

Replication and Fragmentation

- Replication:
  - Should we maintain several identical copies of (part of) the DB, with each replica stored at a different site?
- Fragmentation
  - Should we partition a table into separate parts, each stored at a different site?
  - If we do, how should we partition the table?

Replication

- Advantages
  - Increased availability
    - If one site goes down, the data may be available from elsewhere.
  - Increased parallelism
    - More likely that the data needed is at the site that poses the query
      - Minimizes movement of data between sites.
    - We may send parts of the same query to different sites.

Replication

- Disadvantages
  - Increased storage space
  - Increased overhead on update.
    - Must ensure that all copies are consistent.

Fragmentation

- Why fragment? Why not simply store different complete tables at different sites?
- Two reasons:
  - Applications usually access only a subset of a relation.
    - Minimizes movement of data between sites.
  - Fragmentation gives possibility of increased parallelism.
- In fragmentation, make sure that we do not lose information.

Types of Fragmentation

- Horizontal fragmentation
  - Assign different tuples to each fragment (e.g., through a selection in the sense of relational algebra).
- Vertical fragmentation
  - Assign different attributes to each fragment.
  - Must ensure re-constructability.
- Mixed Fragmentation
  - Mixture of the two.

Problems in Fragmentation

- Increased response time if an application needs to access more than one fragment.
- Especially in vertical fragmentation, ensuring data integrity may become more difficult.
- Allocation: where to place the various fragments, and whether to replicate it.

Allocation

- Allocation concerns where to store each fragment and whether to replicate it.
- Possibilities:
  - Partitioned DB
    - Fragmentation, no replication
  - Partially replicated DB
    - Fragmentation, with each fragment stored at more than one site.
  - Fully replicated DB
    - DB is replicated in full at each site.

Evaluation of Partitioned DB

- Query processing is moderately difficult, and can be time consuming.
- Updating of DB is easy.
- Directory management is moderately difficult
- Concurrency control is easy, since can be done at one site.
- Reliability is very low.
- Requires least amount of space.

Evaluation of Partially Replicated DBs

- Query processing is moderately difficult, and can be time consuming.
- Updating of DB is moderately difficult.
- Directory management is moderately difficult.
- Concurrency control is hard.
- Reliability is high.
- Moderate amount of space.

Evaluation of Fully Replicated DBs

- Query processing is easy, and can be done quickly.
- Updating of DB is easy but time-consuming.
- Directory management is easy but time consuming.
- Concurrency control is moderately hard.
- Reliability is very high.
- Requires a lot of space.

Query Optimization

- Each DBMS has a query processor.
- The query processor takes a high level query (e.g., in SQL) and translates it into a set of relational algebraic expressions.
- Since this can be done in a number of different ways, query processor must choose the best one. This is called query optimization.

Example of query optimization

- Consider tables:
  - Empl(Eno, Ename, Title)
  - Job(Eno, JobNo, Resp, Dur)
- Query:

```
SELECT Ename
FROM Empl, Job
WHERE Empl.Eno = Job.Eno
  AND Resp = 'Manager';
```

- Two strategies

$$\pi_{ename}(\sigma_{Resp = \text{'Manager'}} (E \bowtie J))$$

$$\pi_{ename}(E \bowtie \sigma_{Resp = \text{'Manager'}}(J))$$

Distributed Query Processing/Optimization

- Query processing/optimization is more difficult in distributed DBMS
  - Require both global optimization and local optimization
  - A query may require data from more than one site, and communication has a cost.
  - It may be possible to perform some sub-queries in parallel.
  - Cost no longer dependent on only number of tuples accessed.

Example

Site 1: $E_1 = \sigma_{ENO \leq \text{'E3'}}(E)$

Site 2: $E_2 = \sigma_{ENO > \text{'E3'}}(E)$

Site 3: $J_1 = \sigma_{ENO \leq \text{'E3'}}(J)$

Site 4: $J_2 = \sigma_{ENO > \text{'E3'}}(J)$

Results are expected at site 5.

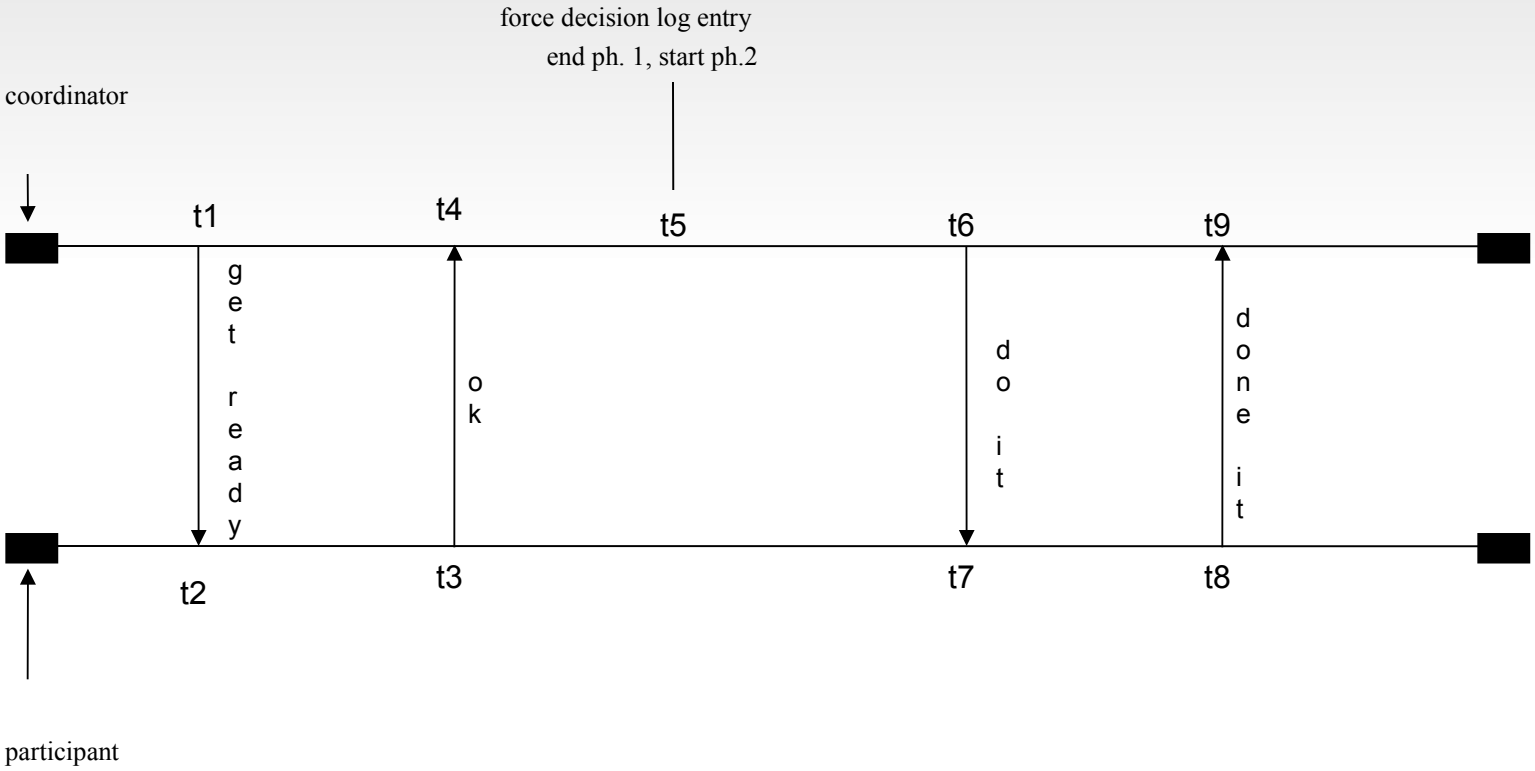
Different Possibilities

- Strategy 1:
  - Send everything to Site 5 and perform the original query there.
- Strategy 2:
  - Do selections at sites 3 and 4
  - Send results to sites 1 and 2
  - Perform join and projections
  - Send result to site 5.
- Strategy 2 might seem better but if communication to site 5 is cheap/fast, then 1 may be better.

Distributed Transaction Management

- Ensuring ACID properties of global transactions is more complex.
- Atomicity:
  - A single transaction may involve data at many sites.
  - All sites must commit or rollback in unison.
  - Two Phase Commit can be used to enforce this.

Two-Phase Commit



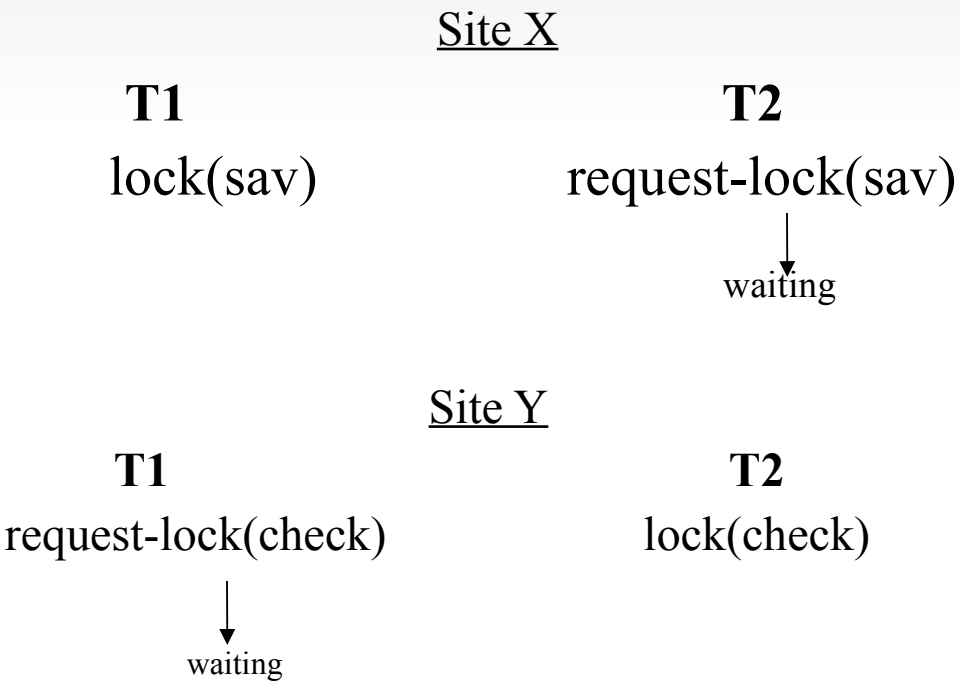
Concurrency Control

- More complex if there are multiple copies of the same data item.
- All copies of the data item must be updated (or none).
- To update an item, a lock on all copies must be granted.
- To request a lock on a copy of a data item a message must be sent to each of the sites with that copy.
  - These messages come with an overhead

- Possible to have Global Deadlocks.
  - High communication overhead to deal with this.

Global Deadlock

- E.g. Assume that data item *sav* is stored at site X and data item *check* at site Y. Both T1 and T2 need both.



Data Dictionaries

- A data dictionary in a non-distributed system contains so-called *meta-information*, information about the data.
- Examples: structure of tables, data types of attributes.
- In distributed DBMS's, data dictionary must also say where the fragments can be found.

Design of Global Data Dictionaries

- Should we have a single global directory, or a set of local ones?
- Should be maintain the data dictionary globally or in a distributed fashion?
- Should we replicate the data dictionary?

Multidatabase Systems

- Pre-existing databases form the components of multidatabase systems.
- Databases are heterogeneous.
  - Varying degrees of heterogeneity.
  - E.g. may be using different data models, dbms's, query languages, naming conventions.
- Provide logical data integration without requiring physical data integration.
  - Must address the heterogeneity issues.
  - Even more difficult to design than distributed systems.
  - Wrapper-Mediator approach is commonly used.