Advances in Data Management
Distributed and Heterogeneous Databases

A.Poulovassilis

1 What is a distributed database system ?

A **distributed database system** (DDB system) consists of several databases stored at different sites of a computer network.

The data at each site is managed by a **database server** running some DBMS software.

These servers can cooperate in executing **global queries** and **global transactions** i.e. queries and transactions whose processing may require access to databases stored at multiple sites.

A significant cost factor in global query and transaction processing is the communications costs incurred by transmitting data between servers over the network.

An extra level of coordination is needed in order to guarantee the ACID properties of global transactions.

Data Fragmentation and Replication:

In distributed relational databases, a relation may be *fragmented* across multiple sites of the DDB system, for better query performance:

- fragments of relations can be stored at the sites where they are most frequently accessed;
- **intra-query parallelism** can be supported i.e. a single global query can be translated into multiple local subqueries that can be processed in parallel;

Fragmentation might be **horizontal** or **vertical**:

- Horizontal fragmentation splits a relation R into n disjoint subsets $R_1, R_2, \ldots, R_n$ such that
$$R = R_1 \cup R_2 \cup \ldots \cup R_n$$
- Vertical fragmentation splits a relation R into n projections $\pi_{atts_1}R, \pi_{atts_2}R, \ldots, \pi_{atts_n}R$, such that
$$R = \pi_{atts_1}R \bowtie \pi_{atts_2}R \bowtie \ldots \bowtie \pi_{atts_n}R$$
This is known as a **loss-less join decomposition** of R, and requires each set of attributes $atts_i$ to include a key (or superkey) of R.

Hybrid fragmentation is also possible i.e. applying both vertical & horizontal fragmentation.

Relations or fragments of fragmented relations can also be **replicated** on more than one site, with the aim of:

- faster processing of queries (using a local copy rather than a remote one)

- increased availability and reliability (if a site failure makes one copy unavailable, another copy may be available on a different site)

A disadvantage of replication is the added overhead in maintaining consistency of replicas after an update occurs to one of them.

Thus, decisions regarding whether on not to replicate data, and how many replicas to create, involve a trade-off between lower query costs and increased availability/reliability on the one hand, and increased update and replica synchronisation costs on the other.

There is a Global Catalog at each site of a relational DDB which holds information about the fragmentation, allocation and replication of relations in the DDB.

Autonomy and heterogeneity of DDB systems

DDB systems may be **homogeneous** or **heterogeneous**:

- Homogeneous DDB systems consist of local databases that are all managed by the same DBMS software.
- Heterogeneous DDB systems consist of local databases each of which may be managed by a different DBMS.

  Thus, the data model, query language and the transaction management protocol may be different for different local databases.

DDB systems may be **integrated** or **multi-database**:

- Integrated DDB systems provide one integrated view of the data to users.

  A single database administration authority decides what information is stored in each site of the DDB system, how this information is stored and accessed, and who is able to access it.
- Multidatabase DDB systems consist of a set of fully autonomous 'local' database systems.

  Additional middleware — the *Multi-Database Management System* (MDBMS), or *Mediator* — manages interaction with these local database systems, including providing global query processing and global transaction management capabilities.

  The MDBMS interacts with each local database system through an appropriate Wrapper, which provides information about the data and the query processing capabilities of the local database system.

  The local DBAs have complete authority over the information stored in their databases and what part of that information is made available to global queries and global transactions.

  One or more 'global' DBAs control access to the multi-database system, but must accept the access restrictions imposed by the local DBAs.

The general architecture of an integrated DDB is illustrated in Figure 1.

The general architecture of a multidatabase DDB is illustrated in Figure 2.

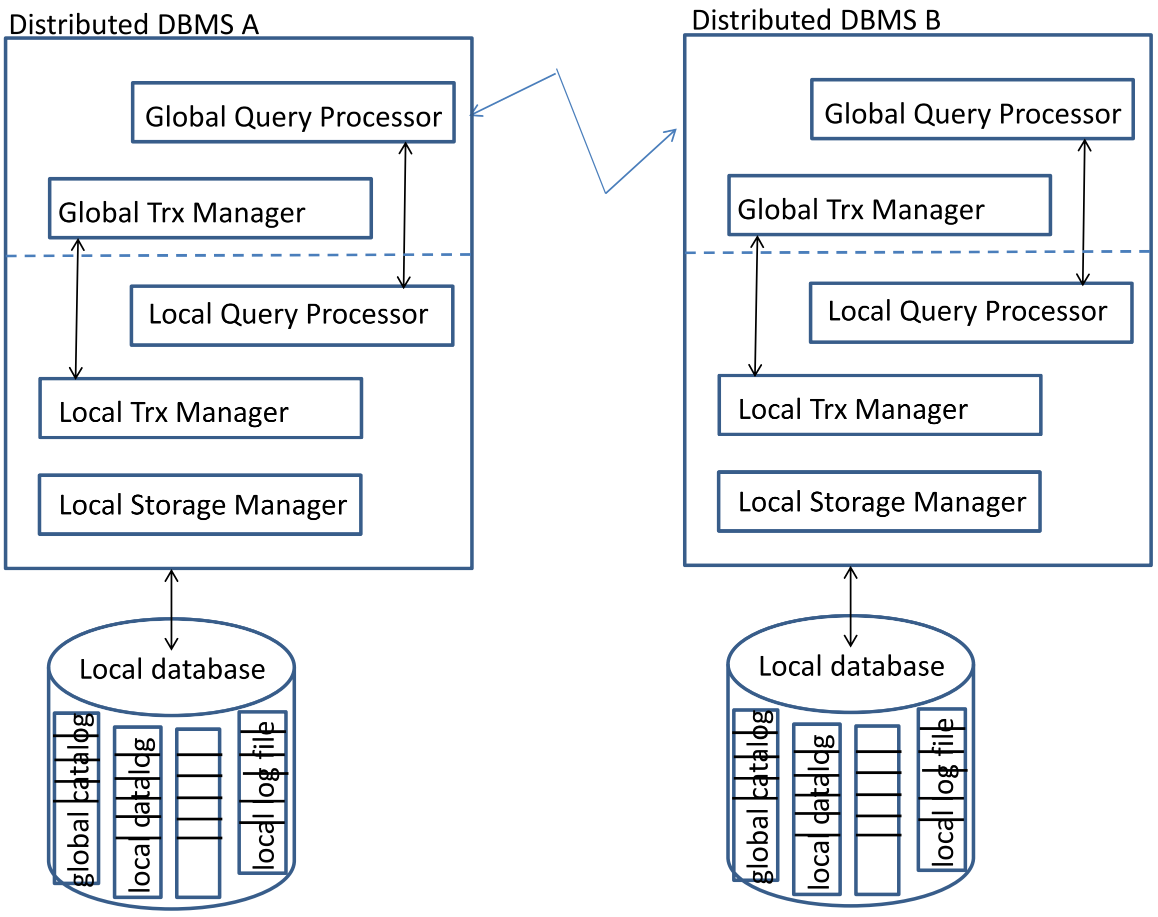

Figure 1: Integrated DDB Architecture

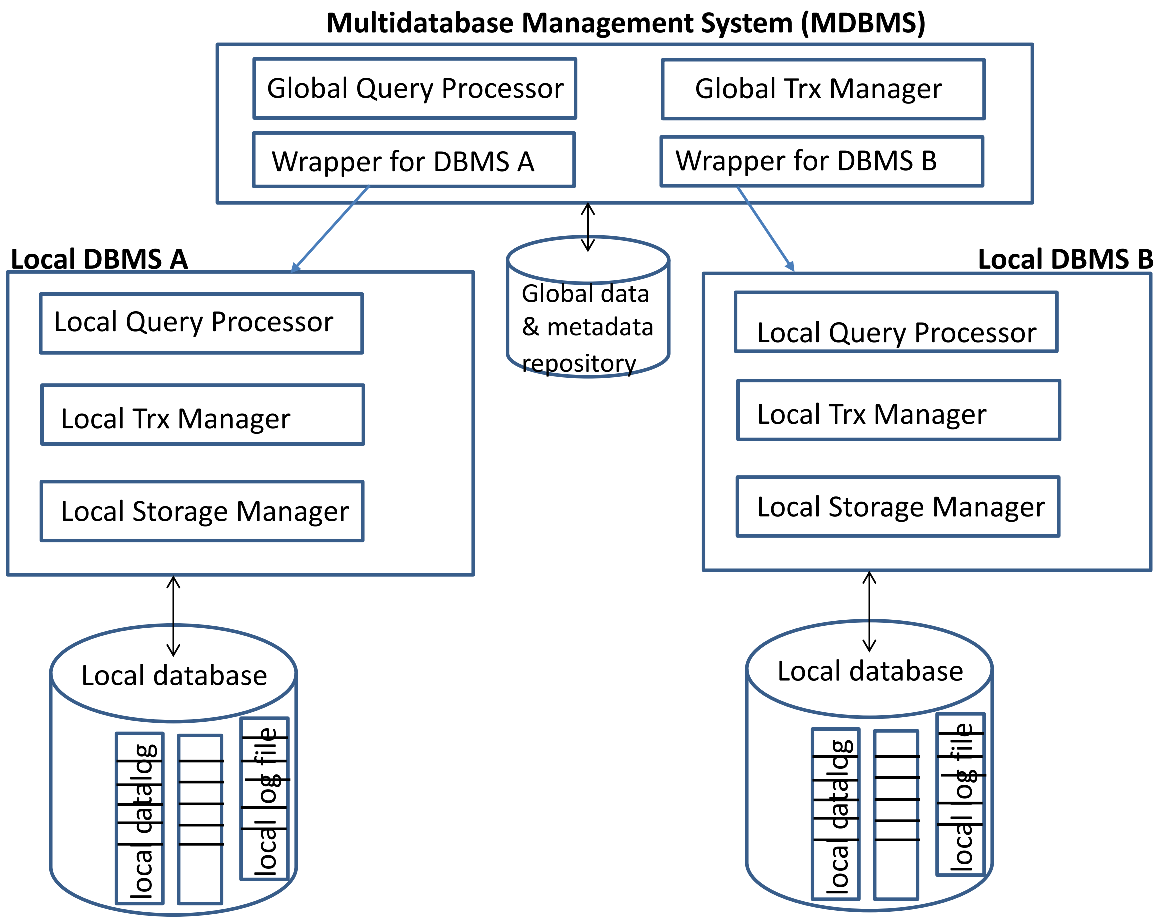

Figure 2: Multidatabase DDB Architecture

The most common combinations of the autonomy and heterogeneity properties for DDBs are:

1. integrated and homogeneous
2. multi-database and heterogeneous

so I will be focussing on these in this course.

For simplicity, I will use the term **homogeneous DDB** to mean ‘integrated, homogeneous DDB’ and I will use the term **heterogeneous DDB** to mean ‘multi-database, heterogeneous DDB’.

In homogeneous DDBs, the global schema is simply the union of all the local conceptual schemas i.e. the conceptual schemas at each site.

In heterogeneous DDBs, the local schemas are first translated into a **Common Data Model** (CDM) so that they can then be integrated into a global schema expressed in the CDM.

- It may be that not all of the data from a local database participates in a global schema — the local DBA has control over what part of a local conceptual schema is made accessible to global queries and transactions.
- There may be several global schemas, each formed by translating, transforming and integrating some parts of the local conceptual schemas; for example, different applications may require access to different parts of the information stored in the local databases.

2 Homogeneous DDB Systems

Key advances in homogeneous DDB systems include:

- **distributed database design**: techniques for determining how relations should be fragmented and allocated across sites on the network;
- **distributed query processing and optimisation**: techniques for processing and optimising queries running over networks, where communications costs are significant;
- **distributed transaction management**: extensions of concurrency control, commit and recovery protocols in order to guarantee the ACID properties of global transactions consisting of multiple local sub-transactions executing at different sites.

The first of these is beyond the scope of this course and we focus on the other two topics.

2.1 Distributed Query Processing

The purpose of distributed query processing is to process global queries i.e. queries expressed with respect to the global or external schemas of a DDB system.

The **local query processor** at each site is responsible for processing sub-queries of global queries that can be evaluated at that site.

A **global query processor** is needed at sites of the DDB system to which global queries can be submitted. This will optimise each global query, distribute sub-queries of the query to the appropriate local query processors, and collect the results of these sub-queries to return to the user.

In more detail, processing global queries consists of the following steps:

1. translating the query into a query tree;
2. replacing fragmented relations in this tree by their definition as unions/joins of their horizontal/vertical fragments;
3. simplifying the resulting tree using several heuristics (see below);
4. global query optimisation, resulting in the selection of a query plan; this consists of sub-queries each of which will be executed at one local site; the query plan is annotated with the data transmission that will occur between sites;
5. local processing of the local sub-queries; this may include further optimisation of the local sub-queries, based on local information about access paths and database statistics.

In **Step 3**, the simplifications that can be carried out in the case of horizontal partitioning include the following:

- eliminating fragments from the argument to a selection operation if they cannot contribute any tuples to the result of that selection;

  for example, suppose a table Employee(empID,site,salary,...) is horizontally fragmented into four fragments:
  $E_1 = \sigma_{site='A' \ AND \ salary<30000}\text{Employee}$
  $E_2 = \sigma_{site='A' \ AND \ salary>=30000}\text{Employee}$
  $E_3 = \sigma_{site='B' \ AND \ salary<30000}\text{Employee}$
  $E_4 = \sigma_{site='B' \ AND \ salary>=30000}\text{Employee}$

  then the query $\sigma_{salary<25000}\text{Employee}$ is replaced in Step 2 by

  $$\sigma_{salary<25000}(E_1 \cup E_2 \cup E_3 \cup E_4)$$

  which simplifies to

  $$\sigma_{salary<25000}(E_1 \cup E_3)$$

- distributing join operations over unions of fragments, and eliminating joins that can yield no tuples;

  for example, suppose a table WorksIn(empID,site,project,...) is horizontally fragmented into two fragments:
  $W_1 = \sigma_{site='A'}\text{WorksIn}$
  $W_2 = \sigma_{site='B'}\text{WorksIn}$

then the query $Employee \bowtie WorksIn$ is replaced in Step 2 by:

$$(E_1 \cup E_2 \cup E_3 \cup E_4) \bowtie (W_1 \cup W_2)$$

distributing the join over the unions of fragments gives:

$$(E_1 \bowtie W_1) \cup (E_2 \bowtie W_1) \cup (E_3 \bowtie W_1) \cup (E_4 \bowtie W_1) \cup (E_1 \bowtie W_2) \cup (E_2 \bowtie W_2) \cup (E_3 \bowtie W_2) \cup (E_4 \bowtie W_2)$$

and this simplifies to:

$$(E_1 \bowtie W_1) \cup (E_2 \bowtie W_1) \cup (E_3 \bowtie W_2) \cup (E_4 \bowtie W_2)$$

One simplification that can be carried out in Step 3 in the case of vertical partitioning is that fragments in the argument of a projection operation which have no non-key attributes in common with the projection attributes can be eliminated.

For example, if a table Projects(projNum,budget,location,projName) is vertically partitioned into two fragments:
$P_1 = \pi_{projNum,budget,location}$Projects
$P_2 = \pi_{projNum,projName}$Projects

then the query $\pi_{projNum,location}$Projects is replaced in Step 2 by:

$$\pi_{projNum,location}(P_1 \bowtie P_2)$$

which simplifies to:

$$\pi_{projNum,location} P_1$$

Step 4 (Query Optimisation) consists of generating a set of alternative query plans, estimating the cost of each plan, and selecting the cheapest plan.

It is carried out in much the same way as for centralised query optimisation, but now *communication costs* must also be taken into account when estimating the overall cost of a query plan.

Also, the *replication* of relations or fragments of relations is now a factor — there may be a choice of which replica to use within a given query plan, with different costs being associated with using different replicas.

Another difference from centralised query optimisation is that there may be a speed-up in query execution times due to the *parallel processing* of parts of the query at different sites.

Distributed Processing of Joins

Given the potential size of the results of join operations, the efficient processing of joins is a significant aspect of global query processing in distributed databases and a number of distributed join algorithms have been developed:

(i) Full-join method

The simplest method for computing $R \bowtie S$ at the site of S consists of shipping R to the site of S and doing the join there. This has a cost of

$$\textit{cost of reading } R \; + \; c \times \textit{pages}(R) \; + \; \textit{cost of computing } R \bowtie S \textit{ at site}(S)$$

where c is the cost of transmitting one page of data from the site of R to the site of S, and $pages(R)$ is the number of pages that R consists of.

If the result of this join were needed at a different site, then there would also be the additional cost of sending the result of the join from $site(S)$ to where it is needed.

(ii) Semi-join method

This is an alternative method for computing $R \bowtie S$ at the site of S and consists of the following steps:

(i) Compute $\pi_{R \cap S}(S)$ at the site of S, where $\pi_{R \cap S}$ denotes projection on the common attributes of R and S.

(ii) Ship $\pi_{R \cap S}(S)$ to the site of R.

(iii) Compute $R \ltimes S$ at the site of R, where $\ltimes$ is the **semi-join** operator, defined as follows:

$$R \ltimes S = R \bowtie \pi_{R \cap S}(S)$$

(iv) Ship $R \ltimes S$ to the site of S.

(v) Compute $R \bowtie S$ at the site of S, using the fact that

$$R \bowtie S = (R \ltimes S) \bowtie S$$

Example 1. Consider the following relations, stored at different sites:
$R = accounts(\underline{accno}, cname, balance)$
$S = customer(\underline{cname}, address, city, telno, creditRating)$.
Suppose we need to compute $R \bowtie S$ at the site of S.

Suppose also that

- *accounts* contains 100,000 tuples on 1,000 pages
- *customer* contains 50,000 tuples on 500 pages
- the *cname* field of S consumes 0.2 of each record of S

With the full join method we have a cost of

$$cost\ of\ reading\ R \ + \ c \times pages(R) \ + \ cost\ of\ computing\ R \bowtie S\ at\ site(S)$$

which is 1000 I/Os to read R, plus $(c \times 1000)$ to transmit R to the site of S, plus 1000 I/Os to save it there, plus $(3 \times (1000 + 500))$ I/Os (assuming a hash join) to perform the join. This gives a total cost of:

$$(c \times 1000) + 6500 \text{ I/Os}$$

With the semi-join method we have the cost of:

(i) Computing $\pi_{R \cap S}(S)$ at $site(S)$, i.e. 500 I/Os to scan S, generating 100 pages of just the *cname* values

(ii) Shipping $\pi_{R \cap S}(S)$ to *site*(R) i.e.

$$c \times 100$$

and saving it there i.e. 100 I/Os.

(iii) Computing $R \ltimes S$ at *site*(R) i.e. $3 \times (100 + 1000)$ I/Os, assuming a hash join

(iv) Shipping the result of $R \ltimes S$ to the site of S i.e.

$$c \times 1000$$

and saving it there i.e. 1000 I/Os.

(v) Computing $R \bowtie S$ at *site*(S) i.e. $3 \times (1000 + 500))$

This gives a total cost of

$$(c \times 1100) + 9400 \text{ I/Os}$$

So in this case the full join method is cheaper: we have gained nothing by using the semi-join method since all the tuples of R join with tuples of S.

Example 2. Let R be as above and let
$S = \sigma_{city='London'} customer$
Suppose again that we need to compute $R \bowtie S$ at the site of S.

Suppose also that there are 100 different cities in *customer*, that there is a uniform distribution of customers across cities, and a uniform distribution of accounts over customers. So S contains 500 tuples on 5 pages.

With the full join method we have a cost of

$$\textit{cost of reading } R \; + \; c \times \textit{pages}(R) \; + \; \textit{cost of computing } R \bowtie S \textit{ at site}(S)$$

which is 1000 I/Os + $(c \times 1000)$ + 1000 I/Os + $(3 \times (1000 + 5))$ I/Os = $(c \times 1000)$ + 5015 I/Os.

With the semi-join method we have the cost of:

(i) Computing $\pi_{R \cap S}(S)$ at *site*(S), i.e. 5 I/Os to scan S, generating 1 page of *cname* values

(ii) Shipping $\pi_{R \cap S}(S)$ to *site*(R) i.e.

$$c \times 1$$

plus 1 I/O to save it there.

(iii) Computing $R \ltimes S$ at *site*(R) i.e. $3 \times (1 + 1000)$ assuming a hash join

(iv) Shipping $R \ltimes S$ to the site of S i.e.

$$c \times 10$$

since, due to a uniform distribution of accounts over customers, $1/100^{th}$ of R will match the *cname* values sent to it from S.

Plus the cost of saving the result of $R \ltimes S$ at the site of S, 10 I/Os.

(v) Computing $R \bowtie S$ at *site*(S) i.e. $3 \times (10 + 5))$

The overall cost is thus $(c \times 11) + 3064$ I/Os.

So in this case the semi-join method is cheaper. This is because a significant number of tuples of R do not join with S and so are not sent to the site of S.

(iii) Bloom join method

This is similar to the semi-join method, as it too aims to reduce the amount of data being sent from the site of R to the site of S.

However, rather than doing a projection on S and sending the resulting data to the site of R, a bit-vector of a fixed size k is computed by hashing each tuple of S to the range $0..k-1$ (using the join attribute values). The i^{th} bit of the vector is set to 1 if some tuple of S hashes to i and is set to 0 otherwise.

Then, at the site of R, the tuples of R are also hashed to $0..k-1$ (using the same hash function and the join attribute values), and only those tuples of R whose hash value corresponds to a 1 in the bit-vector sent from S are retained for shiping to the site of S.

The cost of shipping the bit-vector from the site of S to the site of R is less than the cost of shipping the projection of S in the semi-join method.

However, the size of the subset of R that is sent back to the site of S is likely to be larger (since only approximate matching of tuples is taking place now), and so the shipping costs and join costs are likely to be higher than with the semi-join method.

2.2 Distributed Transaction Management

The purpose of distributed transaction management is to achieve the ACID properties for global transactions.

The **local transaction manager (LTM)** at each site is responsible for maintaining the ACID properties of sub-transactions of global transactions that are being executed at that site.

A **global transaction manager (GTM)** is also needed in order to distribute requests to, and coordinate the execution of, the various LTMs involved in the execution of each global transaction.

There will be one GTM at each site of the DDB system to which global transactions can be submitted.

Each GTM is responsible for guaranteeing the ACID properties of transactions that are submitted to it. In order to do this, it employs distributed versions of the concurrency control and recovery protocols used by centralised DBMS.

Distributed Concurrency Control

A commonly adopted approach is to use **distributed 2PL**, together with an atomic commitment protocol such as 2PC (see below) which ensures that all locks held by a global transaction are released at the same time.

In distributed 2PL, a GTM is responsible for coordinating global transactions that are submitted to it for execution. The GTM sends transactions' lock requests to the participating LTMs as needed. Local lock tables and waits for graphs are maintained at the LTMs.

If data items are replicated at multiple sites, a **ROWA** (Read One, Write All) locking protocol is commonly employed by GTMs:

- an R-lock on a data item is only placed on the copy of the data item that is being read by a local subtransaction;
- a W-lock is placed on *all* copies of a data item that is being written by some local subtransaction;
- since conflicts only involve W-locks, and a conflict only needs to be detected at one site for a global transaction to be prevented from executing incorrectly, it is sufficient to place an R-lock on just one copy of a data item being read and to place a W-lock all copies of a data item being written.

An alternative to ROWA is the **Majority Protocol**, which states that:

if a data item is replicated at n sites, then a lock request (whether R-lock or W-lock) must be sent to, and granted by, more than half of the n sites.

ROWA has the advantage of fewer overheads for Read operations compared to the majority protocol, making it advantageous for predominantly Read workloads.

The majority protocol has the advantage of lower overheads for Write operations compared with ROWA, making it advantageous for predominantly Write or mixed workloads. Also, it can be extended to deal with site or network failures, in contrast to ROWA which requires all sites holding copies of a data item that needs to be W-locked to be available.

Distributed Deadlocks

With distributed 2PL, a deadlock can occur between transactions executing at different sites.

To illustrate this, suppose the relation $accounts(\underline{accno}, cname, balance)$ is horizontally partitioned so that the rows for accounts 123 and 789 reside at different sites, under the management of different LTMs.

Suppose two global transactions are submitted for execution:

$$\begin{aligned} T_1 &= r_1[account\ 789], w_1[account\ 789], r_1[account\ 123], w_1[account\ 123] \\ T_2 &= r_2[account\ 123], r_2[account\ 789] \end{aligned}$$

Consider the following partial execution of transactions T_1 and T_2 under distributed 2PL:

$r_1[account\ 789], w_1[account\ 789], r_2[account\ 123], r_1[account\ 123]$

T_1 is unable to proceed since its next operation $w_1[account\ 123]$ is blocked waiting for T_2 to release the R lock obtained by $r_2[account\ 123]$.

T_2 is unable to proceed since its next operation $r_2[account\ 789]$ is blocked waiting for T_1 to release the W lock obtained by $w_1[account\ 789]$

In a centralised DB system, the 'waits-for' graph would contain a cycle, and either T_1 or T_2 would be rolled back.

In a DDB, the LTMs store their own local waits-for graphs, but also periodically exchange 'waits-for' information between each other, possibly at the instruction of the GTM.

In our distributed DB example, the transaction fragments of T_1 and T_2 executing at the site of account 123 would cause a waits-for arc $T_1 \rightarrow T_2$ which would eventually be transmitted to the site of account 789.

Similarly, the transaction fragments executing at the site of account 789 would cause a waits-for arc $T_2 \rightarrow T_1$ which would eventually be transmitted to the site of account 123.

Whichever site detects the deadlock first will notify the GTM, which will select one of the transactions to be aborted and restarted.

Distributed Commit

Once a global transaction has completed all its operations, the ACID properties require that it be made durable when it commits.

This means that the LTMs participating in the execution of the transaction must either all commit or all abort their sub-transactions.

The most common protocol for ensuring distributed atomic commitment is the **two-phase commit (2PC)** protocol. It involves two phases:

1. The GTM sends the message PREPARE to all the LTMs participating in the execution of the global transaction, informing them that the transaction should now commit.

   An LTM may reply READY if it is ready to commit, after first "forcing" (i.e. writing to persistent storage) a PREPARE record to its log. After that point it may not abort its sub-transaction, unless instructed to do so by the GTM.

   Alternatively, an LTM may reply REFUSE if it is unable to commit, after first forcing an ABORT record to its log. It can then abort its sub-transaction.

2. If the GTM receives READY from all LTMs it sends the message COMMIT to all LTMs, after first forcing a GLOBAL-COMMIT record to its log. All LTMs commit after receiving this message.

   If the GTM receives REFUSE from any LTM it transmits ROLLBACK to all LTMs, after first forcing an GLOBAL-ABORT record to its log. All LTMs rollback their sub-transactions on receiving this message.

   After committing or rolling back their sub-transactions the LTMs send an acknowledgement back to the GTM, which then writes an end-of-transaction record in its log.

A DDB system can suffer from the same types of failure as centralised systems (software/hardware faults, disk failures, site failures); but also, additionally, loss/corruption of messages, failure of communication links, or *partitioning* of the system into two or more disconnected subsystems.

There is therefore a need for a **termination protocol** to deal with situations where the 2PC protocol is not being obeyed by its participants.

There are three situations in 2PC where the GTM or an LTM may be waiting for a message, that need to be dealt with:

- An LTM is waiting for the PREPARE message from the GTM:

The LTM can unilaterally decide to abort its sub-transaction; and it will reply REFUSE if it is later contacted by the GTM or any other LTM.

- The GTM is waiting for the READY/REFUSE reply from an LTM:

  If the GTM does not receive a reply within a specified time period, it can decide to abort the transaction, sending ROLLBACK to all LTMs.

- An LTM which voted READY may be waiting for a ROLLBACK/COMMIT message from the GTM:

  It can try contacting the other LTMs to find out if any of them has either

  (i) already voted REFUSE, or

  (ii) received a ROLLBACK/COMMIT message.

  If it cannot get a reply from any LTM for which (i) or (ii) holds, then it is **blocked**. It is unable to either commit or abort its sub-transaction, and must retain all the locks associated with this sub-transaction while in this state of indecision.

  The LTM will persist in this state until enough failures are repaired to enable it to communicate with either the GTM or some other LTM for which (i) or (ii) holds.

Note 1: 2PC can be made non-blocking for non-total site failures (but only if there is not a network partitioning) by introducing a third phase which collects and distributes the result of the vote before sending out the COMMIT command. This is called the **three-phase commit (3PC)** protocol. Detailed discussion of 3PC is beyond the scope of this course, and can be found in the course textbooks. Because of its additional complexity and additional communications overhead, 3PC is not used in practice.

Note 2: To avoid blocking during distributed commit, *persistent messaging* techniques can be used instead, which rely on application-level exception-handling code to handle cases of delay/non-delivery of messages. Persistent messages are messages that are guaranteed to be delivered precisely once (regardless of failures) if the transaction sending the message commits; and not to be sent at all if the transaction aborts. Additional error-handling code is needed in the application in order to resolve situations where sub-transactions have aborted or in time-out situations. (For more details, see the discussion in Silberschatz et al. Section 19.4.3 — optional reading).

Replication Protocols

With the distributed locking and commit protocols described above, all data replicas are assumed to be updated as part of the same global transaction — this is known as **eager** or **synchronous** replication.

However, a commonly adopted approach in practice is for the DMBS to update just one 'primary' copy of a database object within the transaction itself, and to propagate updates to the rest of the copies afterwards — this is known as **lazy** or **asynchronous** replication.

The primary copy of a database object is Read-Write while the other copies are Read-Only.

Primary copies of database objects are termed 'master data' while non-primary copies are termed 'slave data'.

Locks are only held for those data replicas that are being accessed within the transaction.

The advantages of asynchronous replication are faster completion of transactions, e.g. if some of the slave data is temporarily not available, and lower concurrency control overheads.

The disadvantage is that slave data may not always be in synch with the primary copies, which may result in non-serialisable executions of global transactions reading/writing different versions of the same data item.

This may not be a problem for certain classes of applications, e.g. statistical analysis, decision support systems. But it may not be appropriate for some application classes, e.g. financial applications.

If each database object has just one primary copy, this is known as single-master replication. However, it is also possible for a DBMS to support multiple primary copies of a database objects, in which case this is known as multi-master replication.

Multi-master replication — where there are several primary copies supporting Write operations — may result in a disagreement due to concurrent updates by different transactions on different primary copies of the same database object. This situation requires the DBMS to apply a conflict-resolution policy in order to decide on a single up-to-date value for all the primary copies of the database object.

Availability

Failure of some site or some network link becomes more likely in large-scale distributed systems. DDB systems need to be able to detect such failures, dynamically reconfigure so as to continue their query and transaction processing, and recover when the failed site or link is repaired.

When failed links/sites recover, replicas that are "lagging behind" need to be automatically brought "up-to-date" before they can accept further Read or Write operations.

Several approaches to serialising distributed transactions in a fault-tolerant manner have been proposed, e.g.: using consensus protocols such as PAXOS; using synchronised clocks; workload partitioning; mini-transactions; linear transactions. For a discussion (optional reading) see
"Warp: Lightweight Multi-Key Transactions for Key-Value Stores", R. Escriva, B. Wong, E.G. Sirer, 2014. At `https://arxiv.org/pdf/1509.07815.pdf`

3 Heterogeneous DDB Systems

The purpose of Heterogeneous DDB systems is provide a single interface for users who need to access data from different sources, so that they don't have to be aware of details of the location, implementation or content of the data sources.

The main challenges in implementing Heterogeneous DDB systems lie in:

- **schema translation and integration**
- **global query processing and optimisation**
- **global transaction management**

We note that this kind of *virtual data integration* contrasts with the *materialised data integration* found in data warehousing architectures (see Appendix E).

The virtual integration of heterogeneous data sources is known as *Enterprise Information Integration* (EII) in the commercial sector, and more recently as *Data Virtualization.* Examples of such products are IBM's Infosphere Federation Server and TIBCO's Data Virtualization. Such products can typically access not only structured databases, but also XML files, flat files, 'big data' stores and web services.

3.1 Creating Global Schemas in Heterogeneous DDBs

There are two main steps

1. **schema translation**, of each local conceptual schema into the Common Data Model; and
2. **schema integration** into a global schema.

3.1.1 Schema translation

There are standard algorithms for automatically translating a schema expressed in one data model (that of a local conceptual schema) into an equivalent schema expressed in a different data mdel (that of a global schema).

Example 1. Suppose a heterogeneous database system has access to a relational database and an object-oriented (OO) database, and we wish to integrate them using the relational model as the CDM.

The relational database contains five tables and its schema, L_1, is as follows:

$Student(\underline{studentId}, name, address, \textbf{tutorId})$
$Staff(\underline{tutorId}, name, deptName)$
$Lecturer(\underline{lecturerId}, name, deptName)$
$Course(\underline{courseId}, name, programme)$
$Teaches(\underline{\textbf{lecturerId}}, \underline{\textbf{studentId}})$

Here, the underlined attributes are key attributes, and the bold-face attributes are foreign key attributes referencing the key attributes of the same name in other tables.

Since we are going to use the relational model as the CDM, the above local schema does not need to be translated.

The schema, L_2, of the other local database (the OO one) contains four classes *Department*, *Course*, *Enrollment* and $Staff$. Each *Department* object has attributes

$$\begin{aligned} \underline{deptName} &: string; \\ deptHead &: string; \end{aligned}$$

where underlining signifies a key attribute. Each *Course* object has attributes

$$\begin{aligned} \underline{courseId} &: integer; \\ courseName &: string; \\ units &: integer; \end{aligned}$$

Each *Enrollment* object has attributes

$$\begin{aligned} \underline{studentId} &: integer; \\ \underline{year} &: integer; \\ courseEnrollments &: set(Course) \end{aligned}$$

Each *Staff* object has attributes

$$\begin{aligned} \underline{staffId} &: integer; \\ name &: string; \\ worksIn &: Department; \\ teaches &: set(Course) \end{aligned}$$

Here are some general rules for translating an OO schema into a relational schema (N.B. these are a simplification of the full set of translation rules that would be required, e.g. they do not cover subclass information or structured attributes):

1. Each class C translates into a relation C^R.
2. If there is a set of attributes of C that uniquely identifies the instances of C, these are designated as the key attributes of C^R. Otherwise, add an attribute *C_id* to C^R which will serve at its key.
3. For each other attribute a of C:
   (a) if the type of a is another class, C' say, then a translates into a foreign key in C^R, referencing the key attribute(s) of C'^R;
   (b) if the type of a is $set(C')$ for some class C', then a is translates into a new relation whose attributes are the key attribute(s) of C^R and the key attribute(s) of C'^R.
   (c) otherwise a translates into an attribute of C^R.

Applying these rules, the information in the local OO schema L_2 translates into the following relational schema, C_2:

$Department(\underline{deptName}, deptHead)$
$Course(\underline{courseId}, courseName, units)$
$Enrollment(\underline{studentId}, \underline{year})$
$Staff(\underline{staffId}, name, \mathbf{deptName})$
$Teaches(\underline{\mathbf{staffId}}, \underline{\mathbf{courseId}})$
$CourseEnrollments(\underline{\mathbf{studentId}}, \underline{\mathbf{year}}, \underline{\mathbf{courseId}})$

3.1.2 Schema integration

One or more "export" schemas can be defined by the local DBA, i.e. subparts of the local schema that the local DBA is willing to make available to the multi-database and its potential set of users.

After being translated into the CDM, the set of export schemas contributing to a global schema can then be **integrated**.

There are three main steps involved in this integration:

1. **schema conforming** — making sure that the export schemas all represent the same information using the same schema constructs;

   since the local databases will typically have been designed by different people at different times to serve different application requirements, **conflicts** may exist between export schemas;

   thus, schema conforming requires **conflict detection** and **conflict resolution**, resulting in new versions of the export schemas which represent the same information using the same schema constructs;

2. **schema merging** — identifying common constructs between different export schemas, and producing a single integrated schema;

3. **schema improvement** — improving the quality of the integrated schema by removing any redundant information.

A series of **equivalence-preserving transformations** can be applied in steps 1 and 3:

During step 1, the transformations that are applied include:

- removing **synonyms** and **homonyms**:

  synonyms are schema constructs with different names but representing the same real-world concept;

  homonyms are schema constructs with the same name but representing different real-world concepts;

  we need to **rename** schema constructs so that constructs representing different real-world concepts have different names, and constructs representing the same real-world concept have the same name;

- removing **structural conflicts** e.g.

  – converting a relation $R(\underline{k_1}, \ldots, \underline{k_n}, a_1, \ldots, a_m)$ into m relations $R(\underline{k_1}, \ldots, \underline{k_n}, a_1)$, $\ldots$, $R(\underline{k_1}, \ldots, \underline{k_n}, a_m)$;

  – or vice versa;

  – using an attribute of a relation whose domain has n possible values to create n new relations without that attribute; for example, for a relation People(Name,DoB,Gender) where the attribute Gender has possible values M and F, we can create two relations Male(Name,DoB) and Female(Name,Dob);

  – or vice versa.

During step 3, we can

- add **implied** schema constructs, or
- remove **redundant** schema constructs.

Example 2. To illustrate, consider the two relational schemas from Example 1. Suppose that the two export schemas, ES_1 and ES_2, that we wish to integrate into a global schema are identical to those two schemas.

During schema conforming, the following steps take place:

- there is a structural conflict between ES_1 and ES_2 regarding departments, in that in ES_1 a department is represented by an attribute while in ES_2 it is represented by a relation;

  we can add to ES_1 a new relation $Department(\underline{deptName})$ together with new foreign key constraints on $deptName$ in $Lecturer$ and $Staff$;

- there is a naming conflict between $Lecturer$ and $Staff$ in ES_1 and $Staff$ in ES_2, in that $Staff$ in ES_2 represents the set of course lecturers as does $Lecturer$ in ES_1, whereas $Staff$ in ES_1 represents the set of students' tutors;

  thus, $Staff$ in ES_2 should be remaned to $Lecturer$;

- the attribute $Lecturer.staffId$ in ES_2 should be renamed to $lecturerId$, to conform with $Lecturer.lecturerId$ in ES_1; the attribute $Teachers.staffId$ in ES_2 should similarly be renamed to $lecturerId$;

- the attribute $Course.name$ in ES_1 should be renamed to $courseName$, to conform with $Course.courseName$ in ES_2;

- there is a naming conflict between the relations $Teaches$ in ES_1, where it represents students taught by lecturers, and $Teaches$ in ES_2 where it represents courses taught by lectures;

  one of these relations must be renamed to a different name — let's say $Teaches$ in ES_1 is renamed to $TeachesStudents$.

The schemas have now been conformed, and can be merged on their common relations and attributes, obtaining:

$Student(\underline{studentId}, name, address, \mathbf{tutorId})$
$Staff(\underline{tutorId}, name, \mathbf{deptName})$
$Lecturer(\underline{lecturerId}, name, \mathbf{deptName})$
$Course(\underline{courseId}, courseName, units, programme)$
$TeachesStudents(\underline{\mathbf{lecturerId}}, \underline{\mathbf{studentId}})$
$Department(\underline{deptName}, deptHead)$
$Enrollment(\underline{studentId}, \underline{year})$
$CourseEnrollments(\underline{\mathbf{studentId}}, \underline{\mathbf{year}}, \underline{\mathbf{courseId}})$
$Teaches(\underline{\mathbf{lecturerId}}, \underline{\mathbf{courseId}})$

Finally, during schema improvement:

- we can remove the redundant relation *Enrollment* since this information can be derived by projecting on the *studentId*, *year* attributes of *CourseEnrollments*; and
- we can remove the redundant relation *TeachesStudents* since this information can be derived by joining *Teaches* and *CourseEnrollments* over their *courseId* attribute.

The final integrated schema after all of the above transformations is as follows:

$Student(\underline{studentId}, name, address, \textbf{tutorId})$
$Staff(\underline{tutorId}, name, \textbf{deptName})$
$Lecturer(\underline{lecturerId}, name, \textbf{deptName})$
$Course(\underline{courseId}, courseName, units, programme)$
$Department(\underline{deptName}, deptHead)$
$CourseEnrollments(\underline{\textbf{studentId}}, \underline{\textbf{year}}, \underline{\textbf{courseId}})$
$Teaches(\underline{\textbf{lecturerId}}, \underline{\textbf{courseId}})$

Mappings

Generally in DI systems, several kinds of *mappings* are possible for expressing the semantic relationships between global schema constructs and local schema constructs:

- **global-as-view (GAV) mappings**, in which global schema constructs are defined as views over local schema constructs;
- **local-as-view (LAV) mappings**, in which local schema constructs are defined as views over global schema constructs;
- **globoal-local-as-view (GLAV) mappings**, which relate a view over the global schema with a view over the local schemas.

A special class of multi-database DDBs called *Federated DDBs* support just simple GAV mappings that *rename* a local schema construct into a new global schema construct, without any further transformation or merging of local schema constructs. So the global schema of a federated database is just a union of its constituent local schemas.

The AutoMed Heterogeneous Data Integration System

AutoMed is a heterogeneous data integration system developed at Birkbeck and Imperial Colleges — see `http://www.doc.ic.ac.uk/automed` — which supports multiple Common Data Models.

In AutoMed, the mappings relating global schema constructs to local schema constructs are generated automatically from a sequence of schema transformations, such as those of the example earlier (see Homework Reading - optional).

AutoMed is a characterised as a **both-as-view** data integration system, as its schema transformations can be used to generate both GAV and LAV mappings (and indeed also GLAV mappings).

3.2 Instance-level data conformance

So far I have ignored the possibility that different databases may have different representations for the same information.

For example, for sums of money different currencies, for weights and measures different units of measurement, for numeric amounts different precision etc.

This **representational heterogeneity** also has to be taken into account during schema integration and the necessary datatype conversion functionality needs to be supported in the mapping process.

I have also ignored the possibility that different local databases may contain conflicting data e.g.

- two Customer databases may contain different addresses for the same person;
- two Income databases may contain different incomes for the same person, etc.

A **conflict resolution policy** is needed for such situations, whereby a selection or aggregation criterion is applied to the conflicting data items. For example,

- one Customer database may be regarded as 'primary' and its information may override that of the other;
- the two incomes from the two Income databases may be added together to form the income data item in the global schema.

The logic for handling representational heterogeneity and conflict resolution can be included within the mapping rules that connect global and local schema constructs.

3.3 Global Query Processing

This is more complex in heterogeneous DDBs than in homogeneous DDBs, for a number of reasons:

(a) The extra query translation steps that are needed:

  (i) In Step 2 of Distributed Query Processing (see Section 2.1), a global query expressed on a global schema now needs to be translated into the constructs of the export schemas.

  There are several possible query translation approaches:
  (1) Global-As-View (GAV) — the most common — uses mappings that define each global schema construct as a view over export schema constructs;
  (2) Local-As-View (LAV) uses mappings that define each export schema construct as a view over global schema constructs;
  (3) Global-Local-As-View (GLAV) — the most general — uses mappings that related views defined over the global schema to views defined over the export schemas.

  (ii) In Step 5, local sub-queries expressed using the query language of the Common Data Model have to be translated into queries over on the local schemas expressed using the local query language.

(b) In Step 4, the cost of processing local queries is likely to be different on different local databases. This considerably complicates the task of devising a cost model upon which to base optimisation of the global query.

Moreover, the local cost models and local database statistics may not be available to the global query optimiser.

Thus, the global query optimiser has to rely more on **algebraic** query optimisation techniques e.g. splitting up complex selection conditions and performing selections as early as possible.

One technique that it can use to build up local cost information is to send **calibrating queries** to the local databases, e.g. so as to determine the size of a relation or the selectivity of a selection criterion or the speed of a communication link.

Another way to build up local query cost information is to monitor the actual execution of local sub-queries and record their execution times.

(c) The local databases will in general support different query languages and may have different query processing capabilities.

Thus, in Step 4, the global query processor needs to know (or to infer) the query processing capabilities of the local databases and only send them queries that they are able to process.

(d) This also means that some **post-processing** of local sub-queries may have to be undertaken by the global query processor in order to combine the results of the local sub-queries — this is an extra 6th step that needs to be added to the 5 steps for Distributed Query Processing in homogeneous DDBs listed in Section 2.1.

3.4 Global Transaction Management

Complications may arise when processing global transactions in heterogeneous DDBs, arising from the heterogeneity and autonomy of the local DBMSs:

- Different local DBMSs may support different concurrency control methods and different notions of serialisability.
- In order to preserve their autonomy, local DBMSs may not wish to expose their local lock tables or waits-for graphs, in which case global conflicts and deadlocks will not be directly detectable.
- Global transactions have the potential to be *long-running*, hence tying up local resources that are being devoted to maintaining the ACID properties of global sub-transactions, and thereby impacting on the performance of the local DBMSs on local transactions.
- It is possible that some local DBMSs may not export 2PC capabilities, so other mechanisms for obtaining global transaction consistency are needed for such sites e.g. the persistent messaging queues mentioned earlier, or the "sagas" model discussed below which based on the execution of compensating transactions.

These problems have led to much research into new approaches for supporting global transactions in heterogeneous database systems.

3.4.1 Alternative transaction models

One solution is to relax the ACID requirements by using **multilevel transaction** models. These allow transactions to consist of sub-transactions that are allowed to commit individually rather than as a whole.

Sagas are one example of such a transaction model:

Sagas consist of a sequence of local sub-transactions $t_1; t_2; \ldots; t_n$ such that for each t_i it is possible to define a **compensating transaction** t_i^{-1} that undoes its effects.

After any local sub-transaction commits, it releases its locks.

Thus, sagas relax the Isolation property since sagas can see the intermediate results of other concurrently executing sagas. This needs to be taken into account by applications programs.

If the overall saga later needs to be aborted, then for all committed sub-transactions their compensating transactions are executed (in reverse order). Thus, the Atomicity property is not relaxed.

If a saga does abort, it will be necessary to abort any other sagas that have read data that was updated by this saga. This may result in a **cascade of compensations**.

Homework Reading

1. Read the Appendices to these Notes.
2. (Optional) Oracle's Distributed Database Management facilities: Chapter 31 of the "Oracle Database Administrator Guide 12c" on "Distributed Database Concepts".
3. (Optional) Oracle's Data Replication facilities: Chapter 1 of "Oracle Advanced Replication 12c", "Introduction to Advanced Replication".
4. (Optional) The Notes on the AutoMed data integration toolkit.
5. (Optional) The Notes on AutoMed's IQL query language, in particular Section 2 (but you can skip 2.2) and Section 3.

Appendix A. Distribution Transparency in homogeneous DDBs

Ideally, a DDB should make the distribution of the data transparent to the user i.e. the distributed data should appear as a single database, and be accessible as if it were stored in a single database.

There are different levels of distribution transparency:

- **fragmentation transparency**: the user does not need to know how data is fragmented, perceives the distributed data as a single database, and uses global names in queries/updates
- **location transparency**: the user does not need to know where in the network the data fragments are located, but does need to know the names of the fragments; fragment names need to be used in queries/updates
- no transparency: the user needs to specify both the names and the locations of fragments in queries/updates

Globally unique names are necessary for identifying fragments and replicas stored in a DDB e.g. names of the form: *CreationSiteId.RelationName.FragmentId.CopyId*.

Having to use such names within queries would result in loss of distribution transparency, and a common solution is to set up aliases for them in each site's Global Catalog.

Appendix B. Physical database connectivity in heterogeneous DDBs

There are two main approaches to interconnecting different DBMSs — **connectors** and **standard APIs**

A connector allows one DBMS to exchange information with another DBMS, but different connector software is needed for each combination of different vendors' DBMSs and thus the approach does not scale very well in highly heterogeneous systems.

This has led to the emergence of standard Application Programming Interfaces (APIs) supported by DBMSs, such as JDBC and ODBC. This allows open, extensible heterogeneous DDB systems to be built, interconnecting any DBMS that supports the API.

Standard APIs are a more generic solution but, since they represent the lowest common denominator, they may allow less information to be shared than via special-purpose connectors.

Use of a standard API involves a driver manager and a driver (for each particular DBMS, Operating System, and Network protocol). Applications communicate with the driver manager to load a particular driver, and then submit requests to the driver.

More details on the different ways of accessing databases and Database APIs can be found in Chapter 15 of Lemahieu et al., Principles of Database Management, 2018.

Appendix C. Recovery from failures in DDBs

Failures At Local Transaction Managers:

Each LTM in the DDB can use standard techniques based on Redo/Undo logs to recover from system crashes by undoing the operations of unfinished local sub-transactions and redoing the operations of commited ones.

As in a centralised database system, this recovery process is executed each time an LTM is restarted after a crash.

However, there is now the extra complexity that other sites might need to be contacted during the recovery process to determine what action should be taken for particular global transactions:

In particular, if there is a PREPARE record written in an LTM's log for a transaction, but no subsequent ABORT or COMMIT record, then the LTM is 'in doubt' about the status of this transaction.

It therefore needs to contact the GTM to find out the result of the vote on the global transaction, so that it knows whether to rollback or commit its sub-transaction.

If the GTM is not contactable, the other LTMs will need to be contacted.

In the worse case, there may be no information available about the status of the global transaction. In this case, the LTM can proceed by reacquiring the write locks it had held for this transaction, so as not to block the rest of the local recovery process while information about the in-doubt transaction is awaited.

Failures At Global Transaction Managers:

A GTM may fail while coordinating the commitment of a global transaction.

If, when it recovers, there is a GLOBAL-COMMIT or GLOBAL-ABORT record in its log, it can notify the LTMs of this decision (it might or might not have already notified them before it crashed).

If the GTM has no such information in its log, it can either repeat the first phase of the protocol, sending out a PREPARE message, or it can decide to abort the transaction, sending out a ROLLBACK message.

Appendix D. Transaction Standards and Benchmarks

The **X/Open** model defines a set of standard protocols for implementing distributed transaction processing systems.

In particular, there is an X/Open interface defined between a transaction manager and a resource manager called the XA interface (transaction managers and resource managers in this context are analogous to our earlier terminology of GTMs and LTMs).

The XA interface includes functions such as:
open and *close* to connect/disconnect from a resource manager;
start and *end* to start/end a transaction;
rollback to rollback a transaction;
prepare to prepare for global commit of a transaction;
commit to commit a global transaction.

Thus, as well as implementing their own proprietory versions of 2PC, vendors can also supprt distributed transaction management functionality by providing implementations of the X/Open XA protocol.

The **Transaction Processing Performance Council** is a group of hardware and software vendors who since the 1990s have developed a set of benchmarks aiming to provide a common standard for measuring the performance of transaction processing systems — see `http://www.tpc.org/`.

See `http://www.tpc.org/information/benchmarks.asp` for details of the current set of active benchmarks, including:

- **TPC-C.** This benchmark is designed to measure the performance of traditional OLTP (online transaction processing) environments. TPC-C defines a mixture of read-only and update-intensive transactions for an order-processing application.
- **TPC-E.** A newer benchmark also targeting OLTP environments and simulating the workload of a brokerage firm. It has a more complex database schema, constraint checking requirements, and specifies computer configurations that are intended to be closer to ones that customers would use.
- **TPC-H.** This benchmark is designed to measure the perfomance of decision support environments where users can run complex ad-hoc queries, as well as updates, against a database system.

- **TPC-DS.** A newer benchmark also targeting decision support environments, including "big data" systems.

Appendix E. Materialised Data Integration: Data Warehouses

In the 1970s and 80s, databases were increasingly used to store data about organisations' day-to-day operations.

In such applications, transactions typically make small changes to the database and large volumes of such transactions need to be processed efficiently.

DBMSs were designed to perform well for such **On-Line Transaction Processing (OLTP)** applications.

From the 1990s, organisations have placed increasing focus on developing applications which need to access different sources of current and past data as a single, consistent resource.

The aim of this kind of application is to support high-level decision making, as opposed to day-to-day operation.

Such applications are known as **decision support systems** (DSS). **On-line analytical processing (OLAP)** and **data mining** are examples of DSS.

DSS queries are generally historical and statistical in nature, involving data that may cover time-scales of months or years. Such queries are typically too complex to run directly over the, typically distributed, primary data sources.

Hence the need for a **data warehouse** which integrates and centralises into a single database the necessary information to support DSS applications, and whose data storage, indexing and query processing functionalities are designed to support such applications.

DSS queries typically do not require the most up to date operational version of all the data. Thus, updates to the primary data sources do not have to be propagated to the data warehouse immediately.

Creating a data warehouse comprises three major activities: data extraction, data cleansing and transformation, and data loading (also known as ETL — extraction, transformation, loading)

The DW needs to be periodically refreshed in order to reflect updates in the primary data sources. This uses techniques for incremental view maintenance.

Out-of-date data also needs to be periodically purged from the DW onto archival media.

Comparison of DWs with Heterogeneous DDBs

DWs share several features with Heterogeneous DDBs and there has been much cross-fertilisation between these areas:

- the need for semantic integration of heterogeneous data sources;
- the possibility of erroneous and/or inconsistent data in the data sources;

- the need for query processing over the integrated resource.

There are also of course several key differences, which imply different challenges:

- in a DW, the integrated data is materialised whereas in a HetDB it is retrieved from the data sources (so the global schema is a virtual schema);
- in a DW, the data will not in general be consistent with the current data sources but with some version of them from the recent past;
- in a DW, query processing and transaction management is done centrally on the materialised data, whereas in a HetDB it is distributed over the data sources.

Appendix F. Peer-to-Peer Databases

In **peer-to-peer** (P2P) systems a number of autonomous servers, or *peers*, share their computing resources, data and services. Each peer can be both a provider and a consumer of services.

P2P systems have many potential advantages:

- scalability — by the addition of extra peers;
- self-organisation — no centralised control;
- availability — by replication of data and services;
- adaptability and fault-tolerance — robust to node and network failures; no single point of failure; self-repairing.

In keeping with the themes of this ADM course, I focus here on **data-sharing P2P systems** and the challenges that they pose in:

- query performance: finding the data relevant to a query and processing the query;
- transaction support: peers may fail, and may join or leave the network at any time;
- data exchange and integration, in the face of peers' autonomy and heterogeneity of peers' data.

Data-sharing P2P systems may be viewed as a generalisation of virtual data integration, as found in heterogeneous distributed databases, and materialised data integration, as found in data warehouses:

- semantic connections between different peers' datasets are expressed though mappings (GAV, LAV or GLAV mappings); there is no distinction between 'local' and 'global' schemas within these mappings, instead the mappings specify pairwise relationships between the datasets of pairs of peers;

- when answering a query that is submitted to some peer, P, the mappings can be used to extract additional relevant information from other peers that are directly or indirectly reachable from P via these pairwise mappings;
- when processing a data update submitted to a peer, P, the mappings can be used to propagate the change to other peers that have subscribed to be notified of updates to P.

Thus the overall data contained in the system is partially shared, integrated and replicated across the different peers; and query answering makes use of the pairwise mappings between peers.

Query performance

Queries can be submitted to any peer and are answered with respect to the data at this peer as well as the data at peers that it is physically and semantically connected with. There are three dimensions that effect query performance:

- network topology
- data placement
- message routing protocol

The **network topology** might be

- unstructured: peers connect to whomever they wish; or
- structured according to some protocol, e.g. in the HyperCup topology peers are connected into a n-dimensional hypercube.

Data placement defines how data and metadata is distributed. This might be:

- according to ownership: each peer stores only its own data and metadata;
- according to some distributed data placement algorithm, e.g. Distributed Hash Tables;
- according to the semantics of the data, e.g. in the Edutella P2P data-sharing system there is schema-based clustering of peers, with each group of peers being connected to one "superpeer" (see Reference 1).

The **message routing protocol** defines how messages are transmitted from peer to peer:

- it can take advantage of the network topology, the data placement strategy, and any routing indexes available at the peers;
- P2P networks that broadcast all queries to all peers do not scale: as the number of peers increases, so the message transmission costs increase exponentially.

Transaction support

Similar challenges as with heterogeneous distributed databases arise in P2P data-sharing systems:

- heterogeneity in the concurrency control and recovery methods supported by different peers;
- autonomy of peers may result in restricted access to the information required to coordinate global transaction execution;
- the computing resources available at peers may be limited, making long-running global transactions problematic;
- peers may not wish to cooperate in the coordination of global transactions;
- peers may disconnect at any time from the network, including during the execution of a global transaction in which they are participating.

It is therefore generally necessary to relax both the Atomicity and the Isolation properties of global transactions:

Subtransactions executing at different peers may be allowed to commit or abort independently of their parent transaction committing or aborting, and parent transactions may be able to commit even if some of their subtransactions have failed.

Subtransactions that have committed can be reversed, if necessary, by executing *compensating* subtransactions.

For coordinating the execution of compensating subtransactions, an *abort graph* can be maintained that describes the abort dependencies between parent transactions and their subtransactions.

The abort graph can be distributed amongst the peers that participate in any subtransaction of a top-level transaction. The graph can be extended dynamically at the start of each new subtransaction.

Example P2P data integration systems

Much research has been undertaken into *schema-based* peer-to-peer data integration, and example prototype systems are Edutella (L3S, U. Hannover), Piazza (U. Washington) and AutoMed (Birkbeck and Imperial) — see References 1-3.

References

1. W. Nejdl et al. EDUTELLA: a P2P networking infrastructure based on RDF. In Proc. 11th WWW Conference, pp. 604-615, 2002.

2. A.Y. Halevy et al. Schema mediation in peer data management systems. In Proc. 19th ICDE Conference, pp. 505-516, 2003.

3. P. McBrien and A. Poulovassilis. P2P Query Reformulation over Both-As-View Data Transformation Rules. In Proc. DBISP2P 2006, pp. 310-322.

4. K. Haller et al. Transctional peer-to-peer information processing: The AMOR approach. In Proc. 4th Int. Conf. on Mobile Data Management, Springer, pp. 356-362, 2003.

5. A. Bonifati et al. Distributed Databases and Peer-to-Peer Databases: Past and Present. ACM SIGMOD Record 37(1), pp. 5-11, 2008.