

8. Distributed DBMSs.

- [Syllabus](#)
 - [End of Page](#)
-

Databases and Distributed Database Management Systems - *DBMSs* - had their origin in large organisations' needs for centrally-controlled information management, and the software and associated administrative procedures were developed for that environment. Later, with the advent of small business machines and particularly PCs, single-user DBMSs were widely adopted to provide reliable and simple information-processing facilities for individuals or small working groups. There is now tendency to link machines together in networks, intended to give the advantages of local processing while maintaining overall control over system integrity and security. Developments in database technology obviously reflect this trend.

A DBMS like Oracle can for instance be run under Unix in *CLIENT-SERVER* mode. Using the *TCP/IP* protocol, an Oracle application running on a workstation can communicate with Oracle on a server, the tasks being shared between the two processors - the server handles updating and retrieval; the client application handles screen management, user data entry, and report generation. When implemented this should provide better performance.

Note that a DBMS which supports a fully relational interface is important for the success of this approach, as for the fully distributed databases to be discussed later. Using a relational language, interactions between the server and the client involve retrieving sets of records, which puts less load on the network than single-record transactions. Database servers are sometimes referred to as *SQL ENGINES* in that their only interaction with client machines is through basic SQL commands to accept and produce data. Standard SQL provides a common language with which completely different software products can communicate.

In a true distributed database, the data itself is located on more than one machine. There are various possible approaches, depending on the needs of the application and the degree of emphasis placed on central control versus local autonomy. In general, organisations may wish to:

- reduce data communications costs by putting data at the location where it is most often used,
- aggregate information from different sources,
- provide a more robust system (e.g. when one node goes down the others continue working),
- build in extra security by maintaining copies of the database at different sites.

Distributed systems are not always designed from scratch - they may evolve from traditional systems as organisational needs become apparent. One possibility is that a complete central database is maintained and updated in the normal way, but that local copies (in whole or part) are sent periodically to remote sites, to be used for fast and cheap retrieval. Any local updates have no effect on the central database. The implication here is that consistency between all copies of the database at all times is not crucial - it may for instance be enough to send new data to node sites overnight when networks are less busy.

Alternatively, distributed database development may involve the linking together of previously separate systems, perhaps running on different machine architectures with different software packages. A possible scenario is that individual sites manage and update their own databases for standard operational applications, but that information is collected and aggregated for higher-level decision support functions. In this case there is no single location where the whole database is stored; it is genuinely split over two or more sites. Once again, however, total consistency may not be looked for - local databases are kept up to date and there is periodical transmission of data back to the centre. To manage a system like this a product such as Oracle's SQL*Net is required. This enables, by the use of SQL drivers provided by the host RDBMS (e.g. Oracle ODBC drivers for Access), data stored on, say, an Access DBMS to be interrogated / updated by an Oracle DBMS or vice-versa. Note that under these circumstances it is essential that the appropriate driver is capable of generating standard SQL; SQL is the universal database language used for communicating between different RDBMSs.

A third possibility is that the database is designed from the start to be distributed, and that all nodes in the network may in principle query and update the database at any location. Codd has specified a set of criteria to characterise a genuinely distributed system; these are not in fact satisfied by any actual *DDBMS* commercially available today but, as with the 12 "commandments" about relational systems, they provide a framework for explanation and evaluation.

8.1. Local autonomy.

8.2. No reliance on a central site.

These points concern the overall control mechanism within a DDB, and in particular the location of data dictionary and system catalogue material. How much is it necessary for each site to know about data held elsewhere? In a local area network it is feasible for any node to broadcast a query to all the others, and await response from the one with the relevant information. In wide area networks, communications costs become more significant, and it is necessary to decide how and where to place information to determine the routing of queries. This point will be explored further after the next three rules have been explained.

8.3. Data fragmentation.

8.4. Transparency of location

8.5. Replication

The requirement is that the user of a DDB need not know how the data is partitioned, where any part of it is stored, or how many copies exist, as the system will be intelligent enough to present it as a seamless whole. No current general-purpose DDBMS can achieve this, although it is always possible to write code for particular applications which hides lower-level details from end-users. Decisions about fragmentation, location and replication are very important for the design of a DDB, and are now discussed in more detail.

A relational database is partitioned by first dividing it into a number of *FRAGMENTS*. In theory a fragment may be a complete table, or any *HORIZONTAL / VERTICAL* subset of it which can be described in terms of relational select and project - in other words groups of records or groups of fields. Choice of fragments will be based on expectations about likely usage.

1. *HORIZONTAL FRAGMENTATION* might depend on geographical divisions within an organisation so that, e.g. payroll or customer records are held in the location where they are most likely to be created and accessed. It should partition tables into discrete groups, based either directly on field values or indirectly on joins with another horizontally fragmented table (derived fragmentation). It should not result in missing records or overlaps!
2. *VERTICAL fragmentation* might depend on functional divisions within an organisation, so that, e.g. the site normally dealing with taxation has the relevant fields from all employee-records. There must in this case be some overlap - at least the primary key of vertically-fragmented tables will be repeated, and the designer may define clusters of fields to eliminate the potential need for many cross-site joins.

The fragments are now allocated to individual sites, on the basis of where they are most likely to be needed. Decisions are based on the *COST* and the *BENEFIT* of having a data fragment at a particular site, where:

- The *BENEFIT* relates to the number of times it will be needed to answer queries,
- The *COST* relates to the number of times it will be changed as a result of a transaction at another site.

The site with the best *COST BENEFIT RATIO* will be selected as the location for the fragment.

The designer may choose to replicate the data, i.e. keep several copies of each fragment in different locations. This provides extra security, and flexibility in that there is more than one way to answer the same question. However it increases the potential update cost and in practice it has been found that the benefits of holding more than two or three replicated copies will not generally outweigh the cost. At this stage the question may arise as to whether total consistency between copies is always necessary - such a requirement will place a

particularly heavy load on the transaction management software.

The final design stage involves the *MAPPING* of global database fragments to tables in local databases. It is important to adopt a naming system which allows unambiguous reference to sections of the global database, while retaining users' freedom to select their own local names. Global names will generally incorporate site names (which must be unique), and in some systems may have a more complex structure. In IBM's experimental DDBMS (R*) every database unit has a name identifying:

1. CREATOR_NAME,
2. CREATOR_SITE,
3. LOCAL_NAME,
4. BIRTH_SITE.

where BIRTH_SITE is the name of the site where the data was originally created. This name is guaranteed never to change, and will normally be mapped to local names by way of *SQL CREATE SYNONYM* clauses. It provides a convenient mechanism for actually finding database fragments, as will be described shortly.

The next important design decision is about where to locate the *SYSTEM CATALOGUE*. Query processing in a DDB may require access to the following information:

- GLOBAL SCHEMA
- FRAGMENTATION SCHEMA
- ALLOCATION SCHEMA
- LOCAL MAPPINGS
- AUTHORISATION RULES
- ACCESS METHODS
- DATABASE STATISTICS

Note that this information is not static - in principle changes may occur in any of the above categories and in particular database fragments may over time migrate from one site to another as patterns of access evolve. With a truly distribution-independent DBMS any alterations should be invisible to existing applications.

In principle it is possible to adopt one of the following strategies for holding the system catalogue. Each choice has advantages and disadvantages.

- Hold one copy only, in a central site. This is the simplest solution to manage, since there is no redundancy and a single point of control. The disadvantage is that the central site acts as a bottleneck - if the catalogue there becomes unavailable for any reason the rest of the network is also out of action. It is the solution adopted in practice by many current organisations but it violates Codd's criteria for categorisation as a full *DDBMS*.
- Replicate copies of the complete catalogue over all sites. This allows any site to carry out every stage of query processing, even down to generating and optimising query plans. Total replication produces a high overhead, particularly if changes to any part of the catalogue must be propagated throughout the network. Some systems operate a *CACHEING* mechanism whereby sites hold and use versions of the catalogue which are not guaranteed to be up-to-date, but may allow some queries to be processed without access to the latest version.

Another compromise is to replicate only part of the catalogue, e.g. the RDBMS INGRES arranges that all sites hold items (a), (b), and (c) - i.e. CREATOR_SITE, LOCAL_NAME and BIRTH_SITE - from the list given above. Any site knows where to direct queries, but the task of generating query plans is delegated to the site where the data is held. This may prove a barrier to global distributed query optimisation.

- Maintain only local catalogues. This solution does provide complete site autonomy but may give rise to extensive network traffic, since all sites must be interrogated for every query to see if they are holding relevant information. While perhaps tolerable on a small system using a local area network, this solution cannot be adopted in systems with high communication costs. However, using a convention where local names are mapped onto global names via synonyms, it is possible to ensure that any data element is accessible in at most two moves. For example, the R* system mentioned above holds complete catalogues for all database elements at both their birth-site and their current site, if these are

different. Query processing now involves the following actions:

- convert from local synonym to global name,
- identify the birth-site and interrogate it for the data,
- the birth-site will either return the data or, if it has migrated elsewhere, will know its current location, and inform the query site accordingly,
- the query site can now interrogate the site where the data is currently stored.

8.6. Continuous operation

It should not be necessary to halt normal use of the database while it is re-organised, archived, etc. It may be easier to achieve this in a distributed rather than centralised environment, since processing nodes may be able to substitute for one another where necessary.

8.7. Distributed query processing

The DDBMS should be capable of gathering and presenting data from more than one site to answer a single query. In theory a distributed system can handle queries more quickly than a centralised one, by exploiting parallelism and reducing disc contention; in practice the main delays (and costs) will be imposed by the communications network. Routing algorithms must take many factors into account to determine the location and ordering of operations. Communications costs for each link in the network are relevant, as also are variable processing capabilities and loadings for different nodes, and (where data fragments are replicated) trade-offs between cost and currency. If some nodes are updated less frequently than others there may be a choice between querying the local out-of-date copy very cheaply and getting a more up-to-date answer by accessing a distant location.

The ability to do query optimisation is essential in this context - the main objective being to minimise the quantity of data to be moved around. As with single-site databases, one must consider both generalised operations on internal query representations, and the exploitation of information about the current state of the database. A few examples follow.

1. Operations on the query tree should have the effect of executing the less expensive operation first, preferably at a local site to reduce the total quantity of data to be moved. Distributed query processing often requires the use of UNION operations to put together disjoint horizontal fragments of the same table - these are obviously expensive, and should (like join operations) be postponed as late as possible. A good strategy is to carry out all the REDUCER operations locally and union together the final results. This applies not only to select and project, but also to the aggregation functions. Note however that:

```
COUNT( UNION( frag1, frag2, frag3) )
```

must be implemented as

```
SUM( COUNT( frag1 ), COUNT( frag2 ), COUNT( frag3 ) ),
```

and

```
AVERAGE( UNION ( frag1, frag2, frag3 ) )
```

must be implemented as:

```
SUM( SUM( frag1 ), SUM( frag2 ), SUM( frag3 ) )
```

```
SUM( COUNT( frag1 ), COUNT( frag2 ), COUNT( frag3 ) )
```

- Given that the original fragmentation predicates were based on expected access requirements (e.g. records partitioned on a location field), frequently-used queries should need to access only a subset of the tables to which they refer. In the absence of fragmentation transparency, such queries can of course be directed to particular sites. By contrast any DDBMS with full distribution independence should be able to detect whether

some branches of the query tree will by definition produce null results, and eliminate them before execution. In the general case this will require a theorem-proving capability in the query optimiser, i.e. the ability to detect contradictions between the original fragmentation predicates and those specified by the current query.

- A knowledge of database statistics is necessary when deciding how to move data around the network. A cross-site join requires at least one table to be transferred to another site, and for a complex query it may also be necessary to carry intermediate results between sites. Relevant statistics about each table are:

- CARDINALITY (no. of records),
- the size of each record,
- the number of distinct values in join or select fields.

The last point is relevant in estimating the size of intermediate results. For example in a table of 10,000 records, containing a field with 1000 distinct values uniformly distributed, the number of records selected by a condition on that field will be 10 X the number of values in the select range. It is especially useful to estimate the cardinality of join results. In the worst case (where every record matches every other) this will be $M \times N$, M and N being the cardinality of the original tables. In practice the upper bound is generally much lower and can be estimated on the number of distinct values in the two join fields. To take a common occurrence, if we have a primary \rightarrow foreign key join between tables R and S , the upper bound of the result is the cardinality of table S .

A common strategy for reducing the cost of cross-site joins is to introduce a preliminary *SEMI-JOIN* operation. Suppose we have:

```
Emp(Empno,ename,manager,Deptno) at site New York
Dept(Deptno,dname,location, etc) at site London
```

```
Join condition: Dept.deptno = Emp.deptno.
```

We may perform the join as follows:

```
New York: Project on Emp.deptno. Send the result to London
London  : Join the results from New York with Dept.deptno;
           select only matching records.
```

```
Send the matching records from London back to New York to
participate in a full join.
```

This method produces economies if the join is intended to make a selection from Dept, and not simply to link together corresponding records from both tables. However, unless an index exists for Dept.deptno it still involves a full sort on Dept.

An alternative method sends a long bit vector as a filter so that Dept need not be sorted before the semi-join. The vector is formed by applying a hash function to every value of Emp.deptno, each time setting the appropriate bit to 1. It is then sent to London, and the same hash function applied to each value of Dept.deptno in turn - those which match a 1-bit are selected. The relevant records are sent to New York and the full join carried out as before. Since hashing functions always produce synonyms some records will be selected unnecessarily, but experiments using IBM's R* system showed that this method overall gives better performance than normal semi-join.

8.8. Distributed transaction management.

A true DDBMS must be able to handle update transactions which involve changes to data at more than one site, while maintaining overall consistency. As in a centralised system a transaction is defined as series of updates which form a logical unit - they must be carried out in total or not at all. If at any stage it proves impossible to complete a transaction, the effect of all updates made so far must be cancelled. A DBMS capable of transaction handling must allow the start of a transaction to be identified (e.g. Oracle SQL SAVEPOINT), and the use of COMMIT or ROLLBACK commands to make a transaction permanent or to

abort it.

In a distributed system there is an extra complication in that two or more independent processors have to be synchronised. To maintain consistency all relevant sites should be able to carry out their part in the transaction immediately, but in practice there may be a need to relax this requirement, e.g. where a site is currently inoperative its updates may be stored temporarily and executed later. An additional problem arises if fragments of the database have been replicated and all copies must be kept up to date. A compromise strategy is to maintain a master copy for each fragment which is updated immediately, but to propagate other copies in a non time-critical manner.

Handling multi-site transactions requires the use of the *TWO-PHASE COMMIT* protocol. The process involves a *CO-ORDINATOR* site (probably the one where the transaction was initiated) and one or more *PARTICIPATING* or *CO-OPERATING* sites. Note that the kind of discipline involved here may conflict with Codd's original demand for complete site autonomy!

An outline of the algorithm is as follows:

- The coordinator issues details of the transaction to participating sites and asks if they are able to execute it. It then awaits answers from them all.
- Each site reports whether it is ready to perform the transaction or not.
- If all sites agree, the coordinator issues instructions to commit. If any site has refused, or if all answers have not been received within a given time, the coordinator issues instructions to rollback.
- All sites commit or rollback the transaction accordingly.

The process involves extensive use of transaction logs, so that any combination of conditions can be detected and corrected.

The second important issue in this context is *LOCKING*, since there is by definition concurrency and possibly contention for resources between different elements of the system. Provided that the transactions run at each site follow the normal locking discipline, the complete transaction will not produce the lost update problem. In fact, locking over several sites forces unnecessary serial orderings on transactions, reducing overall efficiency by preventing some harmless concurrent executions.

A possible alternative method involves synchronisation via a *TIMESTAMP* mechanism, whereby a unique serial number is assigned to each read or write transaction within the entire database. For each data item, the numbers of the latest relevant read and write time-stamps is recorded. An attempt to read or write using a time-stamp LOWER than the last one recorded for any item causes the whole transaction to be aborted. Clearly this method has substantial overheads, but it has the advantage of avoiding the *GLOBAL DEADLOCKS* which are a potential problem in distributed systems, where no one site has an overview of the whole process so as to detect when different transactions are in contention for the same resources. To attempt to apply global control here would violate the requirement for site autonomy.

8.9. Hardware

8.10. Machine

8.11. Network

8.12. DBMS

The requirement is that a DDBMS should be able to run over multiple hardware and software platforms. The important issue is that all nodes must use a common network protocol, and have a common language for accessing databases, e.g. standard SQL.

-
- [Syllabus](#)
 - Previous: [Transaction Control](#)
 - Next: [Fourth Generation Environments](#)

- [Top of page.](#)

Mike Gatford and Tony Valsamidis

November 1998

tony@is.city.ac.uk