Annals of the University of Craiova, Mathematics and Computer Science Series
Volume 38(3), 2011, Pages 76–83
ISSN: 1223-6934, Online 2246-9958

Fragmentation and Data Allocation in the Distributed Environments

Nicoleta - Magdalena Iacob (Ciobanu)

Abstract. The distributed data processing is an effective way to improve reliability, availability and performance of a database system. In this paper we will concentrate on data allocation problem with the aim to assure an optimal distribution of data in the process of the distributed database design in correlation with data fragmentation. Efficient allocation of fragments requires a balance between costs (storage, processing and transmission of data), performance (especially response time) and data distribution restrictions. The allocation of fragments is closely related to the replication of data from distributed databases. In addition, we analyzed the cost of fragmentation and replication.

2010 Mathematics Subject Classification. Primary 68P05; Secondary 68P15, 68N17.
Key words and phrases. distributed databases, fragmentation design, allocation design, strategies, methods, cost analysis.

1. Distributed databases

Definition 1.1. *Let K be a collection of data* $K_i = \{A_{i_1}, ..., A_{i_k}\}$, $k = 1, 2, ...$ *which consists of a set of attributes (fields, columns) and has an associated set of data (lines, tuples, records). The database:* $BD = \{K_1, ..., K_i, ...\}$, $i \in I$, $I = 1, 2, ..., n$ *is a collection of structured data.*

Definition 1.2. *Let C be a set of computers:* $C = \{C_1, ..., C_j\}$, $j \in J$, $J = 1, 2, ..., m$ *then DDB is a distributed database and* $P_1, P_2, ..., P_j$ *the parties that compose it. Therefore* $DDB = \{P_1 \cup P_2 \cup ... \cup P_j\}$, *with* $\{P_1 \cap ... \cap P_j\} = \emptyset$, *where* P_j *is a corresponding part of the computer* C_j, $P_j \in \{K_1, ..., K_i, ...\}$, $i \in I$, *with* $2 \leq j \leq m$, *i.e.* P_j *is a subset of the DDB database.*

Definition 1.3. *A distributed database system consists of a collection of sites, connected together via some kind of communications network, in which:*

a. Each site is a full database system site in its own right, but

b. The sites have agreed to work together so that a user at any site can access data anywhere in the network exactly as if the data were all stored at the user's own site.

Definition 1.4. *It follows that a distributed database (DDB) is really a kind of virtual database, whose component parts are physically stored in a number of distinct "real" databases at a number of distinct sites (in effect. it is the logical union of those database) [2].*

Received July 05, 2011. Revision received August 25, 2011.
This work was partially supported by the strategic grant POSDRU/88/1.5/S/52826, Project ID52826 (2009), co-financed by the European Social Fund - Investing in People, within the Sectoral Operational Programme Human Resources Development 2007-2013.

Definition 1.5. *A distributed database management system (DDBMS) is a software system that manages a distributed database while making the distribution transparent to the user [3]. Distribution is normally discussed solely in terms of the fragmentation and replication of data. A data fragment constitutes some subset of the original database. A data replicate constitutes some copy of the whole or part of the original database [1].*

2. The design of a distributed database

The design of a distributed computer system involves making decisions on the placement of data and programs across the sites of a computer network. In the case of distributed DBMSs, the distribution of applications involves two things: the distribution of the distributed DBMS software and the distribution of the application programs that run on it [5].

The distributed nature of the data involves new further data: data fragmentation, partitioning, replication and fragments allocation as well as replicas on different sites. The design of DDB can be done, as in the case of centralized database, using the top-down and/or bottom-up approach.

Top-down design shown schematically in Figure 1, aims to assure an optimal distribution of data and is used in the distributed database design phase.

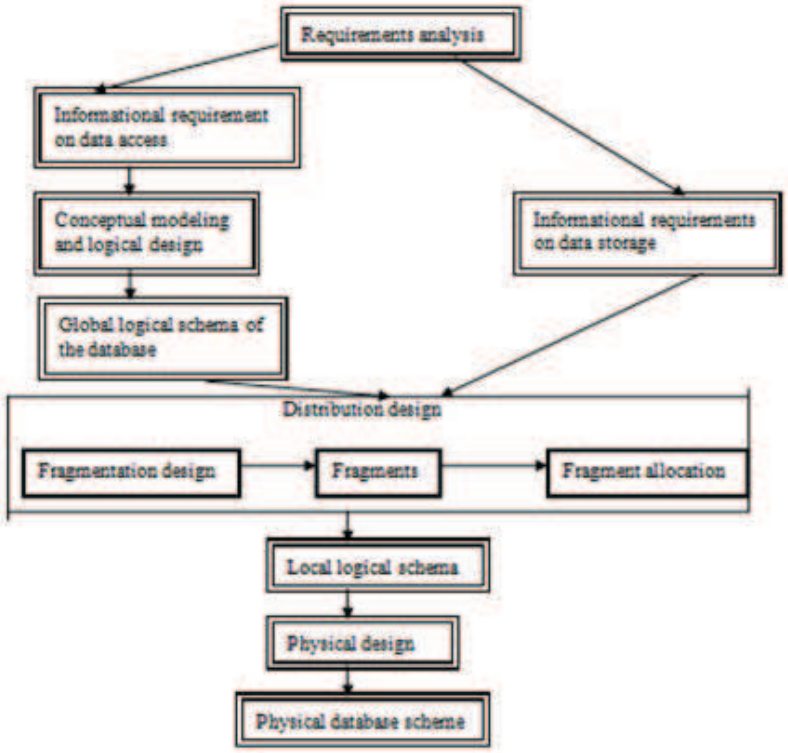

FIGURE 1. Top-down design of distributed databases

The starting point in a top-down design approach is to identify and analyze the requirements of the system which will use the distributed database architecture and

identify the data storage requirements and the data access mode. Based on these requirements first will be designed the global scheme of DDB independent of physical constraints regarding distribution and implementation. The next step is the data distribution design, which will show the fragmentation type of DDB and also allocation of fragments on nodes. The design process will end with the local database design for each node.

The design techniques for global scheme of DDB and local database are similar with those used in a centralized database scenario. Specific to DDB are aspects related to the fragmentation design and fragment allocation scheme to network nodes.

The two activities, fragmentation and data allocation are addressed independently, fragmentation results representing the input in data allocation phase. Both activities have the same inputs, the difference between them being the fact that fragmentation starts from global relationships, while the data allocation takes into account fragmentation results. Both take into account the data access requirements, but each of them ignore the way in which requirements are comprised by the other design decisions. The fragments and their allocation on sites aim to achieve, as much as possible, access to the local data references.

2.1. Fragmentation design.

Definition 2.1. ***Fragmentation.*** *The system partitions the relation into several fragments, and stores each fragment at a different site.*

If a distributed database B is partitioned this means that the parties $P_1, P_2, ..., P_j$, $i \in I$, $I = 1, 2, ..., n$ *which compose her form disjoint subsets:* $B = \{P_1 \cup P_2 \cup ... \cup P_j\}$, *with* $\{P_1 \cap ... \cap P_j\} = \emptyset$.

The fragmentation is the partitioning of a global relation R into fragments R_1, $R_2, ..., R_i$, *containing enough information to reconstruct the original relation R.*

There are three basic rules that should be looked at during the fragmentation, which ensure that the database does not have semantic changes during fragmentation, i.e. ensure consistency of the database:

- **Completeness**. If relation R is decomposed into fragments $R_1, R_2, ..., R_n$, each data item that can be found in R must appear in at least one fragment.
- **Reconstruction**. It must be possible to define a relational operation that will reconstruct R from the fragments. Reconstruction for horizontal fragmentation is *Union* operation and *Join* for vertical.
- **Disjointness**. If data item d_i appears in fragment R_i, then it should not appear in any other fragment.

Exception: vertical fragmentation, where primary key attributes must be repeated to allow reconstruction. In the case of horizontal fragmentation, data item is a tuple; for vertical fragmentation, data item is an attribute.

2.1.1. *Fragmentation strategies.* Consider a relation with scheme R. The fragmentation of R consist of determining the number of fragments (subscheme) R_i obtained by applying an algebraic relation on R (as operations on relations which show the logical properties of data). In this context, the fragmentation of data collection can be done in two ways:

a) Horizontal. The horizontal fragmentation of a relation R is the subdivision of its tuples into subsets called fragments; the fragmentation is correct if each tuple of R is mapped into at least one tuple of the fragments (completeness condition). An additional disjointness condition, requiring that each tuple of R be mapped into

exactly one tuple of one of the fragments, is often introduced in distributed database systems in order to control the existence of duplication explicitly at the fragment level (by having multiple copies of the same fragment). The resulted fragments R_i have the same scheme structure as well as collection R, but differ by the data they contain and are resulted by applying a selection on R.

Selection $o_p(R)$ - defines a relation that contains only those tuples of R that satisfy the specified condition (predicate p): $o_p(\prod a_1, ..., a_n(R))$. A horizontal fragment can be obtained by applying a restriction: $R_i = \eth_{condi}(R)$. So we can rebuild the original relation by union as follows: $R = R_1 \cup R_2 \cup ... \cup R_k$.

For example:

$R_1 = \sigma$ type='House'(PropertyForSale)

$R_2 = \sigma$type='Flat'(PropertyForSale)

There are two versions of horizontal partitioning:

- **primary horizontal fragmentation** of a relation is achieved through the use of predicates defined on that relation which restricts the tuples of the relation.
- **derived horizontal fragmentation** is realized by using predicates that are defined on other relations.

b) Vertical. It divides the relation vertically by columns. The resulted fragments R_i contain only part from the collection structure R. It keeps only certain attributes at certain site and they contain the primary key of the relation R to ensure that the restore is possible and are resulted from the application of a projection operation of relational algebra: $\prod a_1, ..., a_n(R)$, where $a_1, ..., a_n$ are attributes of the relation R. The fragmentation is correct if each attribute of the relation is mapped into at least one attribute of the fragments; moreover, it must be possible to reconstruct the original relation by joining the fragments together $R = R_1 \bigotimes R_2 \bigotimes ... \bigotimes R_n$; in other words, the fragments must be a lossless join decomposition of the relation.

For example:

$R_1 = \prod$staffNo, position, salary(Staff)

$R_2 = \prod$staffNo, firstName, lastName, branchNo(Staff)

c) Sometimes, only vertical or horizontal fragmentation of a database scheme is insufficient to distribute adequately data for some applications. Instead it can be useful to implement **mixed or hybrid fragmentation**. A mixed fragment from a relation consists of a horizontal fragment that is vertically fragmented, or a vertical fragment that is horizontally fragmented. A mixed fragmentation is defined using selection and projection operations of relational algebra: $\sigma_p(\prod a_1, ..., a_n(R))$ or $\prod a_1, ..., a_n(\sigma_p(R))$.

The comparison of fragmentation strategies is showed in table 1.

For each fragment of a relation R:

- Condition **C** = True (all tuples are selected).
- List (**L** = ATTRS(R)) = True (all attributes are included in the list).

	Vertical fragmentation	**Horizontal fragmentation**	**Mixed fragmentation**
C	True (all tuples)	False (not all tuples)	False (not all tuples)
L	False (not all columns)	True (all columns)	False (not all columns)

Table 1: Comparison of fragmentation strategies

The advantages of the fragmentation:

- *Usage* - applications work with views rather than entire relations;
- *Efficiency* - data is stored close to the place where it is mostly frequently used;
- *Parallelism* - with fragments are the unit of distribution, a transaction can be divided into several subqueries that operate on fragments;

- *Security* - data not required by local applications is not restored, and consequently not available to unauthorized users;
- *Performance* of global applications that require data from several fragments located at different sites many be slower.

2.2. Allocation design. A database is named distributed if any of its tables are stored at different sites; one or more of its tables are replicated and their copies are stored at different sites; one or more of its tables are fragmented and the fragments are stored at different sites; and so on. In general, a database is distributed if not all of its data is localized at a single site [6].

The problem of fragment allocation can be treated as a problem of placement optimization for each data fragment. Efficient distribution of fragments requires a balance between costs (storage, processing and transmission of data), performance (especially response time) and data distribution restrictions.

Give:

- a set of m fragments $F = \{F_1, F_2, ..., F_m\}$ exploited by a set of k applications $Q = \{Q_1, Q_2, ..., Q_k\}$ on a set of p sites $S = \{S_1, S_2, ..., S_p\}$.

Determine:

- $Det(F, S) = \begin{cases} 1, & if\ F\ is\ alocated \\ 0, & if\ otherwise \end{cases}$

  The allocation problem involves finding the "optimal" distribution of F to S.

To minimize:

- storage cost + communication cost + local processing cost

Subject to:

- response time constraints
- availability constraints
- network topology
- security restrictions

The allocation of fragments is closely related to the replication of data from DDB. In the data allocation phase the designer must decide whether DDB fragments will be replicated and also their degree of replication.

Definition 2.2. ***Replication****. The system maintains several identical replicas (copies) of the relation, and stores each replica at a different site. The alternative to replication is to store only one copy of relation r.*

A database is replicated if the entire database or a portion of it (a table, some tables, one or more fragments, etc.) is copied and the copies are stored at different sites. The problem with having more than one copy of a database is to maintain the *mutual consistency* of the copies-ensuring that all copies have identical schema and data content. Assuming replicas are mutually consistent, replication improves availability since a transaction can read any of the copies [4]. In addition, replication provides more reliability, minimizes the chance of total data loss, and greatly improves disaster recoveryIn addition, replication provides more reliability, minimizes the chance of total data loss, and greatly improves disaster recovery.

Fragmentation and replication can be combined: A relation can be partitioned into several fragments and there may be several replicas of each fragment [7].

2.2.1. *Methods of data allocation.* In designing data allocation, the following rule must be respected: the data must be placed closer to the location in which will be used. In practice, the methods used for data allocation are:

- **Method of determination of non redundant allocation** (called the best choices method) implies evaluating each possible allocation and choosing a single node for each fragment. The method eliminates the possibility of placing a fragment at a particular node if a fragment is already assigned to that node.
- **The redundant allocation method** on all profitable nodes implies the selection of all nodes for which the benefit of allocating a copy of the fragment is greater than the cost of allocation. Redundant final allocation is recommended when data retrieval frequency is higher than the frequency of updating data. Data replication is more convenient for systems that admit temporary inconsistent data.
- **Progressive replication method** implies initial implementation of a non redundant solution and then progressively introduces the replicated copies from the most profitable node. The benefits are calculated by taking into consideration all query and update operations.

3. Cost analysis

Assume the set of sites is $S = \{S_1, S_2, ..., S_p\}$. Let p be the total number of sites, N_{Rel} be the total number of relations, m be the total number of fragments, N_{frag} be the cardinality of the fragmented relation, n be the number of fragments of the fragmented relation, N_{rep} be the cardinality of each other ($N_{Rel} - 1$) replicated relations, N_{join} be the cardinality of the joined relations, k be the number of attributes in both fragmented and replicated relations, K_{join} be the number of attributes after joining the relations from any site, K_p be the number of attributes to be projected, CT_{comp} be cost per tuple comparison, CT_{conc} be the cost per tuple concatenation, $T_{cost-attr}$ be transmission cost per attribute, TC_R be the replication transmission costs, CP_{p-attr} be the cost per projected attribute.

3.1. Cost analysis of fragmentation and replication (CAFR). The CAFR algorithm requires one of the relations referenced by a query to be fragmented and other relations to be replicated at the sites that have a fragment of the fragmented relation. The query is decomposed into the same number of subqueries as the number of sites and each subquery is processed at one of these sites.

The cost for one of the relation to be fragmented across m sites is:

$$TC_{O-F} = N_{frag} * k * T_{cost-attr} * n. \tag{1}$$

The cost for all other ($N_{Rel} - 1$) relations to be replicated across n sites is:

$$TC_R = (N_{rep} * k * T_{cost-attr}) * (N_{Rel} - 1). \tag{2}$$

The local processing costs of these sites are (union cost for fragmented relation and natural join cost):

$$LPC = \sum_{j=1}^{r} Cost[\bigcup_{i=1}^{n}(F)_i]_j + [(N_{frag} * 1) + N_{rep} * (N_{Rel} - 1) * CT_{comp} + N_{join} * CT_{conc}] * p. \tag{3}$$

It follows that, the total cost for CAFR strategy is:

$$TC_{CAFR} = TC_{O-F} + TC_R + LPC. \tag{4}$$

The CAFR permits parallel processing of a query and are not applicable for processing distributed queries, in which all the non fragmented relations are referenced by a query.

3.2. Cost analysis of partition and replication (CAPR). The CAPR algorithm works as follows. For a given query, the minimum response time is estimated if all referenced data is transferred to and processed at only one of the sites. Next, for each referenced relation and each copy of the relation, the response time is estimated if the copy of the relation is partitioned and distributed to a subset of S and all the other relations are replicated at the sites where they are needed. A choice of processing sites and sizes of fragments for the selected copy of the chosen relation are determined by CAPR so as to minimize the response time. Finally, the strategy which gives the minimum response time among all the copies of all the referenced relations is chosen.

The union cost:

$$UC = \sum_{i=1}^{m} [k_{join} * N_{join} * CT_{conc}]_j. \tag{5}$$

If all the relations are transferred to any of the sites, to project and union the relations then the cost is:

$$TC_{PROJ-UNI} = \sum_{I=1}^{NTR} [N_{rep} * K_P * CP_{p-attr}]_I + UC. \tag{6}$$

3.3. Comparison of cost analysis. CAFR or CAPR requires substantial data transfer and preparation before a query can be processed in parallel at different sites. Performing local reductions before data transfer can reduce the data replication cost and the join processing cost. However, local reduction takes time and it delays data transfer. It is not always true that local reduction will reduce the response time for the processing of a given query. But comparing equations (6) and (4) it is concluded that $TC_{PROJ-UNI} > TC_{CAFR}$; as it is always true that sending all relations directly to the assembly site, where all joins are performed, is unfavorable due to its high transmission overhead and little exploitation of parallelism. This case is the worst case depending on the given input query (wherein there are no conditions or predicates in the where clause of the given query). The fragment and replicate strategy (CAFR) permits parallel processing of a query. CAFR are not applicable for processing distributed queries; in which all the non fragmented relations are referenced by a query. In case of CAFR strategy, if no relation referenced by a query is fragmented, it is necessary to decide which relation will be partitioned into fragments; which copy of the relation should be used; how the relation will be partitioned; and where the fragments will be sent for processing. To resolve this problem, CAPR (Partition and Replicate strategy) is presented. But CAPR algorithm favors joins involving small numbers of relations, and improvement decreases as the number of relations involved in joins increases. Improvement of CAPR over single site processing depends heavily on how fast a relation can be partitioned. If the implementation of relation partitioning is inefficient, CAPR actually gives worse response time than single site processing.

4. Conclusion

The objective of a data allocation algorithm is to determine an assignment of fragments at different sites in order to minimize the total data transfer cost involved in executing a set of queries. This is equivalent to minimization of the average query execution time, which has a primary importance in a wide area of distributed applications. The fragmentation in a distributed database management system increases the level of concurrency and therefore system throughput for query processing. Distributed databases have appeared as a necessity, because they improve availability and reliability of data and assure high performance in data processing by allowing parallel processing of queries, but also reduce processing costs.

References

[1] P. Beynon-Davies, *Database systems* (3rd ed.), New York: Palgrave-Macmillan, 2004.
[2] C.J. Date, *An introduction to Database Systems* (8th ed.), Addison Wesley, 2003.
[3] R. Elmasri and S. Navathe, *Fundamentals of database systems* (4th ed.), Boston: Addison-Wesley, 2004.
[4] N.M. Iacob (Ciobanu), The use of distributed databases in e-learning systems, *Procedia Social and Behavioral Sciences Journal* **15** (2011), 3rd World Conference on Educational Sciences - WCES 2011, Bahcesehir University, Istanbul - Turkey, 03-07 February 2011, 2673–2677.
[5] M.T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems* (3th ed.), New York: Springer, 2011.
[6] S. Rahimi and F.S. Haug, *Distributed database management systems: A Practical Approach*, IEEE, Computer Society, Hoboken, N. J: Wiley, 2010.
[7] A. Silberschatz, H.F. Korth and S. Sudarshan, *Database System Concepts* (6th ed.), McGraw-Hill, 2010.

(Nicoleta - Magdalena Iacob (Ciobanu)) University of Pitesti, Faculty of Mathematics and Computer Science, Computer Science Department, Targu din Vale Street, No.1, 110040 Pitesti, Romania
E-mail address: nicoleta.iacob_2007@yahoo.com