A Parallel Upgrade for Existing Relational Database Management Systems

Harald Kosch

[*]hkosch@lip.ens-lyon.fr

Matthieu Exbrayat

[†]exbrayat@lisiflory.insa-lyon.fr

Abstract

This papers presents a new and low cost approach of paralellising databases, using a parallel DBMS build on a network of workstations. The originality lays in the fact that it works with an existing commercialised DBMS. Coherency is left to the DBMS, and the prototype side only cares for optimal data distribution and query execution. This two-sides architecture allows us to introduce a new definition, the Common Criteria Unit, extending the classical ways of distributing relations, which goal is to introduce some basic intelligence into the distribution method.

Keywords

Parallelism, Networks of Workstations, Relational Databases, Data Distribution

[*]Laboratoire d'Informatique du Parallélisme. Ecole Normale Supérieure de Lyon

[†]Laboratoire d'Ingénierie des Systêmes d'Information. Institut National des Sciences Appliquées de Lyon

1 Introduction

Last five years have seen the arising of parallel techniques into Database Management Systems (DBMS). This is done to provide quick access to large and very large databases to many simultaneous users. Two domains are especially concerned : Online Transaction Processing (OLTP, i.e. : business databases) and Query Processing (QP, i.e. : data extraction). Both have specific needs. OLTP deals with fast and reliable updates, as long as it involves unduplicatable means, such as money, raw materials, or plane tickets. Query processing deals more with high bandwidth and wide storage. Speed of updates, and time in general, are less significant there, as the kind of information is either getting slowly out of date (e.g. : books list), or bringing few considerable incoherence (e.g. : statistical studies data).

Many studies have been driven through OLTP or QP goals. The main topics are data distributing [1, 2], Parallel Execution Plans, and duplicating strategies [3].

Most of those research have been driven through the assumption that parallel DBMS would run on Massively Parallel Machines (MPM). It must be noticed that such machines, even bringing an incomparable rise of performance, are still quite seldom, and represent a big investment. This brought hybrid architectures, such as workstation clusters, or network of workstations to come to front page of research. More than suggestive aspects, such as global cost, it can be considered that workstations are widely used by many companies.

It provides us with robust computing power in comparison to most parallel machines on the market. Moreover one can easily add new workstations to the existing system, while satisfying ???.

This allows us to believe that virtual parallelism between small- to middle-size computers is a promising domain of investigation. We propose in this paper to consider a different approach of parallel databases, based on an original architecture, which is hybrid in at least two ways : first, by using a network of workstation; secondly by recycling and integrating the existing DBMS.

In this paper we present in section 2 the specificities we had to face to use workstation networks, and how distributing techniques fit to them. Section 3 describes our BDMS's structure and abilities. Section 4 presents our implementation and gives an example of use.

2 Hardware-driven assumption

2.1 Networks of workstations

2.2 distributing techniques

3 Prototype description

3.1 Global architecture

Our prototype uses three kinds of machines :

- An existing DBMS (further : EDBMS). It contains the actual complete and coherent database.
- A server. This machine plays the role of the interface between requesters and data.
- Calculators. Those are machines, hardwarely speaking similar to the server. They handle a double function : storing the data in a distributed way, and extracting them in a parallel way.

The server receives users and applications requests.

A request interpreter analysis first those requests first examines them and acts differently depending on their meaning. Transactions (i.e. : updates, deletions, and

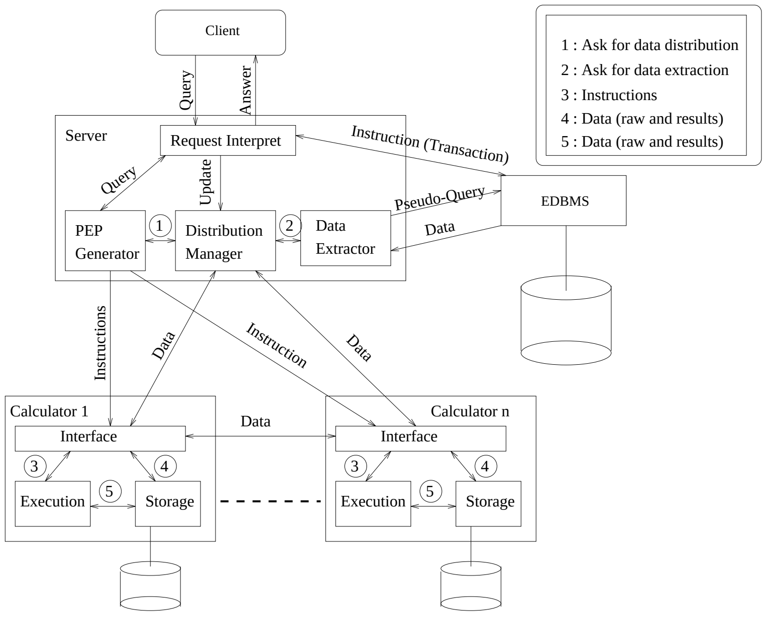

Figure 1: DBMS architecture

creations) are directly handed to the EDBMS. Queries are parallelized and executed on the calculators. This is done by specific modules, one generating the parallel execution plan and looking at is execution (Parallel Query Optimizer); another allowing to know how the data is distributed (Distribution Manager); a third communicating with the EDBMS to extract new data from it (Data Extractor).

Calculators both handle local storage and sub-query execution. To keep this two functions working with few interferences (allowing data reception while executing a sub query) they also need a local interface, in charge of information movements (i.e. : receiving data and instructions, and sending results).

The existing DBMS is kept apart of the parallel phase, except during the data distribution (see section 3.2).

It must be precised that the description given above could drive to any real hardware implementation, from MPM to networks of workstations, even using virtual processors [4]. The following section describes how the data are distributed, and how it best fits to networks of workstations.

3.2 Data strategies

3.2.1 Data unit

Our data distribution technique is derived from the block distribution methods used in GAMMA [5] and in the study of Seeger [6]. Our method is based on distribution functions decided and known by the distribution manager. Each of the *blocks* represents a set of tuples grouped by hashing or range partitioning.
In the case of hashing, we used a hash function (based on the value of one or more attributes of the tuple) to determine the site of storage. For the range partitioning (R.P.), the tuples are distributed according to the value of an attribute, using contiguous range domains.

There exists a third distribution technique, the round robin method, which makes a round distributing (of tuple or blocks) from one site to another. We do not applied it here, as it generates much network overhead and CPU costs on each concerned processor [6].

The most important point of the block method is the way, blocks are structured. As we are looking for a distribution allowing fast access and low network traffic, the basic idea was on one hand to keep together data which is highly supposed to be compared and on the other hand to distribute the data on the attribute, supposed to be most selected. This decision is based on spying the already processed queries and retaining the correlation between comparisons and data.

For this reason, we named these blocks **Common Criteria Units** (ccu). The meaning of the criteria can be used in several ways, as hashing or range partitionning criteria, but also as a semantic criteria (when allowed by the kind of relation e.g. ....). It must be precised here that the goal of our prototype is mainly to allow access to document databases. This means few relations with a huge amount of tuples and a limited number of join possibilities. This also signifies meaningfull relations (key words). Thus, the blocks size can be quite large, but the transfers are still limited.

The *ccu* structure expresses as: *ccuNumber; tuple**, where *ccuNumber* (the unique "key" number of the ccu), followed by the list of tuples. The mapping of the *ccus* to disks is determined by the *Distribution Manager*, using a B*-Tree structure.

3.2.2 Data distribution

Data is extracted from the EDBMS at launch time, relation by relation. Original data distribution is done by the DBMS manager, according to the most common selection attributes. .... These attributes commonly appear in applications (i.e. compiled queries). We establish a quick count of the number and frequency of queries (or transactions) using them. This is quite similar to distributed database study. Better pl can be realised later when comparing actual use rates (obtained by storing and analysing submitted queries and used ccus) with the theoretical ones.

Coherency of data is kept by updating the *uccs* immediatly after updating is processed in the EDBMS. Commit and abort informations are sent from the EDBMS to the request interpreter, which informs the client. In the case of commit, the *Distribution Manager* is asked by the *request interpreter* to update its *uccs*. To insure coherency during update phases, concerned ccus are locked in order to forbid their use. The update is done in two phases. First, the old ccus are sent to the Distribution Manager, which updates them. The new versions are then resent to their storage calculators to replace the older ones. A complete redistribution can be done after consequent updates. The old ccus are redistributed according to the new criterias, and then removed. This can also be done, wehen a new distribution is initiated.

Of course, updates and deletes could be very long without using the parallel function of our DBMS. Selecting the concerned tuple(s) can be done with a first selection of the concerned uccs, then with the selection of the matching tuples, and finally with the adequate treatment (on the EDBMS) of the latters.

3.3 Parallel query execution plans

Parallel query execution plans are based on the model of **DPL-graphs** [7] [1]. A *DPL-graph* not only describes the data flow when executing a specific query against a data base but also simulates aspects of communication, memory management and run-time constraints.

The nodes within a *DPL-graph* represent various operators which can be divided into three different categories: **basic**, **communication** and **control operators**:

- **Basic operators** are atomic operators working on relation partitions. They are part of the implementation of a relational operator. and work independently on each processor, holding a part of the implicated relations. Basic operators are graphically represented by circles whose inscription detail the functionality.

- **Communication operators** implement data redistribution. For their graphical representation boxes were chosen. The kind of repartition that is to be done is stated in their inscription.

- **Control operators** are used to control the query processing. They are represented by lozenges. Inscriptions describe the kind of control to be performed. Special annotations specify the execution parameters.

All operators are enriched with annotations depending on the characteristics of a parallel environment. Those annotations specify for example the method how the stored data is accessed and the kind of data dependency that exists between relational

[1] The term is derived from the **d**ata, **p**recedence and **l**oop dependencies

operations are applied to the nodes of the graph which represent these relational operations. Additionally, the differenttypes and degrees of parallelism, namely pipeline, task and data parallelism, can be referred to.

In order to develop the idea of *DPL-graphs*, a step-by-step approach leads from simple query processing trees to the more complex notion of *DPL-graphs*. The first step is to introduce low-level implementations into the existing model of query processing trees. The resulting graphs are called *D-graphs* (see figure 2 left scheme for an example). Unlike conventional processing query trees a *D-graph* not only displays the data flow during the execution of a given query, but also refers to aspects of the used hash-partition method, i.e. the **data dependency**. By adding annotations, different aspects of the applied parallelism during execution become visible and can therefore be related to the quality of the query execution process.

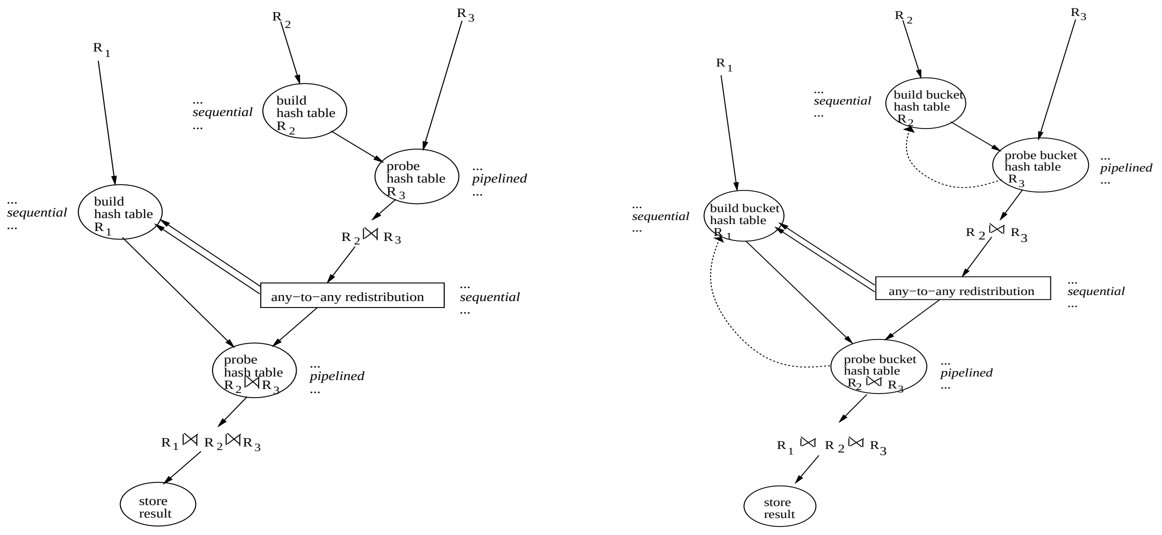

Figure 2: Left scheme: DPL-graph for $q = R_1 \bowtie R_2 \bowtie R_3$ showing the precedence dependecncy. Right scheme: Introducing the loop dependency for the example.

Interdependencies between operators that are not related to the data which is being processed, are not visible in simple query processinf trees yet used. Therefore our DPL-graphs also consider **precedence dependencies**. We speak of *precedence dependency* if an operator must be terminated before another operator can start working – although no data dependency is involved. This relationship between two operators is marked by a double edge (see figure 2, left scheme).

If relations cannot be held in main memory, they are split into buckets which have to be processed seperately. This way, a **loop dependency** is established between the operator that is providing the data bucket by bucket and the operator that is working on that data. The resulting graph is called *DPL-graph* (see figure 3.3, right scheme), because it is powerful enough to visualize *data dependency*, *precedence dependency* and *loop dependency.*

3.4 Query execution

The instruction messages consists of two parts; the first one is a tuple *(ccus, criteria)*, the second one specifies the result distribution (i.e. the number of ccu and the destination calculators).

The instructions are kept in the *calculator's* interface module and transmitted one by one to the *execution module*. Possible generated delays, when waiting for incoming uccs, are partially solved by allowing an instruction to execute only when the incomings ccus are arrived at the execution site. Instructions are then scanned, from the oldest to the last one arrived, and transmitted to the *Execution module* as soon as they can be executed. This solution allows to maintain a good global execution time between queries.

4 Parallel query optimization

We implemented yet the parallel query optimizer, it takes the optimized query plan be the EDBMS and generates a parallel query execution plan in the form of a *DPL-graph*, as explained in section 3.3.

Fig. 3 shows the architecture of our parallel optimizer. The *parallelizer modul*

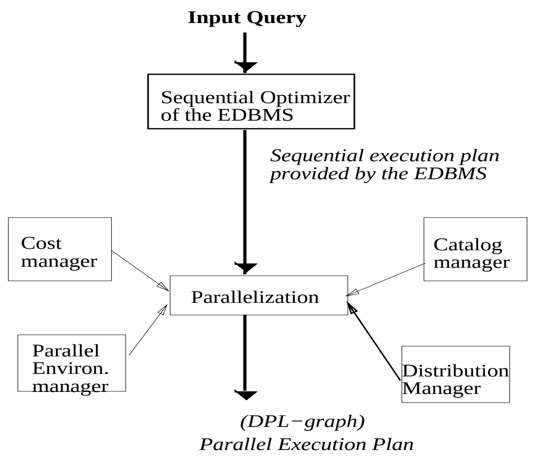

Figure 3: Optimizer architecture.

bases on *randomized search strategies* which is motivated by the increasing number of alternative execution plans in connection with the use of parallelism. The use of those randomized optimization algorithms seems to be the most promising solution to this problem [8].

The set of all possible parallel query execution plans (PEP) is regarded as a search space in which the optimal execution plan has to be found. Each PEP is being associated with a certain cost, obtained by a cost function applied to it. A *transformation* is a modification exercised upon a PEP to obtain another PEP. According to the studies of Lanzelotte [8], two particular transformations were chosen to exploit the whole search space : the *swap* and the *join exchange*. The *swap* transformation inverses the order of the input relations for one join, as the *join exchange* permutate two consecutive joins.

Such transformations are applied to the current PEP until no more cost improvements can be found.

The parallel optimizer architecture (see fig. 3) was constructed as an extensible architecture, in order to provide a robust approach to changing environments. All informations concerning the parallel execution and the relation specific data is kept out of the so called *Parallelizer* and put into special managers.

Two groups of manager can be distinguished, first the hardware specific ones :

- The *parallel environnement manager*, which collects all information of the available hardware configuration (e.g. the number of available calculators).
- The *cost manager*, which provides us with the cost constants (e.g. bandwith and latency of the communication network).

Second, the data specific manager:

- The *catalog manager* which contains static informations about the database relations (e.g. attribut and relation size).
- The *Distribution manager* which describes the distribution function.

5 Conclusion and future work

Il est beau le proto! On va interfacer avec une base documentaire sous Oracle. C'est marrant, ya peu de tables et elles sont assez grosses et puis les enregistrements ont plein de signification, on peut donc esperer un classement "intelligent".

We tend to develop a distributed B-Tree structure.

References

[1] D.J. DeWitt and J. Gray. Parallel database systems : the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[2] R. Gallersdörfer and M. Nicola. Improving Performance in Replicated Databases through Relaxed Coherency. In *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, 1995.

[3] D. Chamberlin and F. Schmuck. Dynamic Data Distribution (D3) in a Shared-Nothing Multiprocessor Data Store. In *Proceedings of the 18th VLDB Conference*, Vancouver, British Columbia, Canada, 1992.

[4] D. Schneider D.J. DeWitt J. Naughton and S. Seshardi. Pratical skew handling in parallel joins. In *Proceeding of the International Conference on Very Large Databases*, Vancouver, British Columbia, August 1992.

[5] D.J. DeWitt S. Ghandeharizadeh D. Schneider A. Bricker H.-I. Hsiao and R. Rasmussen. The GAMMA database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, June 1990.

[6] B. Seeger and P.-Å. Larson. Multi-Disk B-trees. In *Proceedings of the ACM SIGMOD International Conference of Managment of Data*, Miami Beach, USA, December 1991.

[7] L. Brunie and H. Kosch. DPL graphs - a powerful representation of parallel relational query execution plans. In LLNCS Springer, editor, *EUROPAR'96*, August 1996. accepted for publication.

[8] R.S.G. Lanzelotte P. Valduriez and M. Zaït. Industrial-Strength Parallel Query Optimization: Issues and Lessons. *Information Systems - An International Journal*, 1994.