Parallel Evaluation of Relational Queries on a Network of Workstations

Lionel Brunie, Matthieu Exbrayat, André Flory

Laboratoire de l'Informatique du Parallélisme

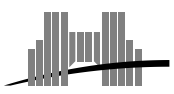
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON
n° 5668

Parallel Evaluation of Relational Queries on a Network of Workstations

Lionel Brunie
Matthieu Exbrayat
André Flory

mars 1999

Research Report N° RR1999-22



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

Parallel Evaluation of Relational Queries on a Network of Workstations

Lionel Brunie
Matthieu Exbrayat
André Flory

mars 1999

Abstract

In this paper we propose an innovative approach to handle "read-most" data bases. This approach is based on a parallel extension, called *parallel relational query evaluator*, working over a network of workstations, in a coupled mode with a sequential Database Management System (DBMS). We present a detailed architecture of the parallel query evaluator and focus on the management of data during executions and transmissions, especially through macro-pipelining. We then present Enkidu, the prototype that has been build according to our concepts. We finally expose a set of measurements, conducted over Enkidu, highlighting both the specific performances of macro-pipelining and the global ones of Enkidu.

Keywords: Relational Database, query parallelism, network of workstations, macro-pipelining.

Résumé

Ce document propose une approche innovante de gestion des bases de données "majoritairement en lecture", c'est-à-dire pour lesquelles l'accès en lecture est très nettement dominant vis-à-vis de l'accès en écriture. L'approche proposée se fonde sur l'usage d'une extension parallèle, appelée *évaluateur relationnel parallèle*, fonctionnant sur un réseau de stations, en couplage avec un Système de Gestion de Bases de Données (SGBD) séquentiel. Nous présentons ici l'architecture détaillée de cet évaluateur parallèle. Nous insistons particulièrement sur la gestion des données durant les transferts et les interrogations, en étudiant notamment l'utilisation du macro-pipelining. Nous introduisons ensuite notre implémentation de l'évaluateur parallèle, le prototype Enkidu. Nous présentons enfin divers tests et mesures réalisés sur le prototype Enkidu et mettant en avant, non seulement les performances globales du prototype, mais également celles plus spécifiques liées au macro-pipelining.

Mots-clés: Base de données relationnelle, parallélisation de requêtes, réseau de stations, macro-pipelining.

Parallel Evaluation of Relational Queries on a Network of Workstations

Lionel Brunie* LISI, INSA Lyon
Matthieu Exbrayat† LIP, ENS Lyon
André Flory* LISI, INSA Lyon

Abstract

In this paper we propose an innovative approach to handle "read-most" data bases. This approach is based on a parallel extension, called *parallel relational query evaluator*, working over a network of workstations, in a coupled mode with a sequential Database Management System (DBMS). We present a detailed architecture of the parallel query evaluator and focus on the management of data during executions and transmissions, especially through macro-pipelining. We then present Enkidu, the prototype that as been build according to our concepts. We finally expose a set of measurements, conducted over Enkidu, highlighting both the specific performances of macro-pipelining and the global ones of Enkidu.

1 Introduction

Many modern database applications, such as decision support, document retrieval or medical databases have to face huge amounts of data. An attractive solution to handle such constraints consists in using parallel DBMSs (PDBMSs), which offer performance and extensibility. However the diffusion of PDBMSs has been strongly limited, due to the cost of parallel machines and of parallel DBMS software. In order to limit the implementation cost, vendors and researchers have recently proposed to port the existing parallel DBMS on networks of workstations [1, 13]. Unfortunately, all of those systems remain expensive because of the complexity of their software. In particular, PDBMSs include complex transactional facilities.

One is allowed to wonder if complete and complex parallel DBMSs are necessary to handle the many applications in which data is mostly accessed in a "read-only" mode. For instance, decision support applications usually concern archive data accessed in an off-line fashion. In such cases, the use of coherency and update functionalities are useless. In the same way, digital libraries (e.g. medical image databases, document library, multimedia archives) are basically used in a read-only way (i.e. modifications occur at very low rates, e.g. once a day or once a week).

*Laboratoire d'Ingénierie des Systèmes d'Information, Institut National des Sciences Appliquées de Lyon, 20 Avenue A. Einstein, F-69621 Villeurbanne Cedex, France. Email: {Lionel.Brunie,Andre.Flory}@insa-lyon.fr

†Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 46 Allée d'Italie, F-69364 Lyon Cedex 07, France. Email: Matthieu.Exbrayat@ens-lyon.fr. The LIP Laboratory is jointly supported by ENS Lyon, CNRS and INRIA (UMR 5668).

In this context, in order to be able to handle the growing amount of data an queries we propose to couple a pre-existing sequential relational DBMS with a parallel query evaluator, which runs on a network of workstations.

Thus, in this paper we propose the software architecture of such a parallel extension. Attention is particularly focused on the management of communications by introducing macro-pipelining heuristics.

A prototype implementing these concepts, called Enkidu, has been developed. Experiments combining Enkidu and Oracle 7 have been run, showing both a linear speed up for Enkidu, and the efficiency of the coupling architecture

In section 2 we present the detailed architecture of our system. In section 3 we describe the structure of the Enkidu prototype. We then present and discuss our experiments in section 4. Related work is commented in section 5. Finally, we summarize this paper and investigate future work in section 6.

2 Architecture of a coupled parallel relational query evaluator

The goal of a coupled query evaluator consists in i) extracting data from one or several pre-existing relational DBMS, ii) distributing this data on a network of workstations and iii) using this distribution in order to process parallel relational queries. In this section we first present a global overview of our proposal. Then we introduce and discuss redistribution techniques. We finally propose a macro-pipelining strategy specifically adapted for processing queries network of workstations.

2.1 Architecture

2.1.1 General overview

The overall architecture [10] can be divided into two main components (see fig. 1): the *server* and the *calculators*. The server is the access point to the whole system for both administrator and users. All tasks are submitted to and treated by it. This *server* is connected to several *calculators* which are in charge of storing and processing redistributed data. The next two sections describe these components in details.

2.1.2 The server module

The server module (see figure 2) consists of eight components allowing the distribution of data (circuit A), the collection of calculators' load information (circuit B) and the parallel query execution (circuit C).

Data distribution is done by the administrator. This latter connects itself to the system through the *interface*. His demand of distribution is then transmitted to the *redistribution manager* (A1), which contacts the DBMS in order to extract the requested data. Extracted data is then transmitted to the *communication* module (A2), which sends it to the calculators. The *redistribution manager* also indicates the redistribution parameters to the *parallel execution optimizer* (A3).

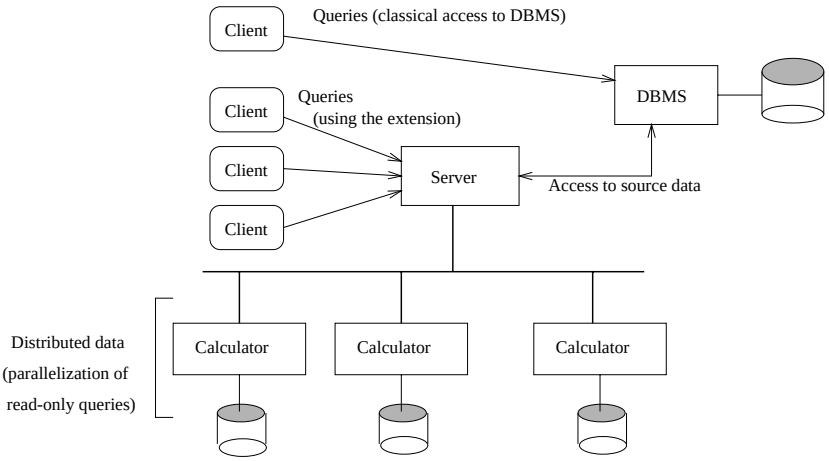


Figure 1: General overview

Processor load information is regularly returned by each calculator in order to allow the implementation of load balancing procedures. It is transmitted to a *load manager* (B1), which in turn transmits it to the *parallel execution optimizer* (B2). Distribution and load information is used by this latter in order to find the best suited location for each operation.

Query execution is triggered by submitting a SQL query through the *interface*. This query is then transformed in an internal format by the *SQL analyzer* (C1). The raw execution plan obtained is then improved by the *parallel execution optimizer* (C2), which produces an optimized parallel execution plan (PEP). This plan (C3) consists of basic (elementary) operators connected by flows of data and pre- and post-conditions, e.g. scheduling decisions [7]. The *parallel execution manager* analyses the PEP so that each calculator only receives the operators which takes place on it (C4). The *parallel execution manager* receives processing information during the execution, indicating, for instance, the end of each operator (C5). Resulting tuples are grouped and stored by the *result manager* (C6), and then returned to the user (C7).

2.1.3 The calculator module

The calculator module consists of five components (see figure 3). Here again we have the *communication* module, which allows each calculator to exchange data and messages with both the server and the other calculators. Incoming data is transmitted to the *storage* module which stores it.

Incoming instructions are placed in a queue. Then they go through a *scheduling* module, and are presented to the *computation* module as needed. Intermediate results that will be used locally are transmitted to the *storage* module, while other results are sent to the other calculators (intermediate results) or to the server (final results). Execution messages are also sent to the server at the end of each operator.

Finally, administration messages (suppression of a relation, shutdown, etc.) can be received and treated, with the possibility to send acknowledgment messages back.

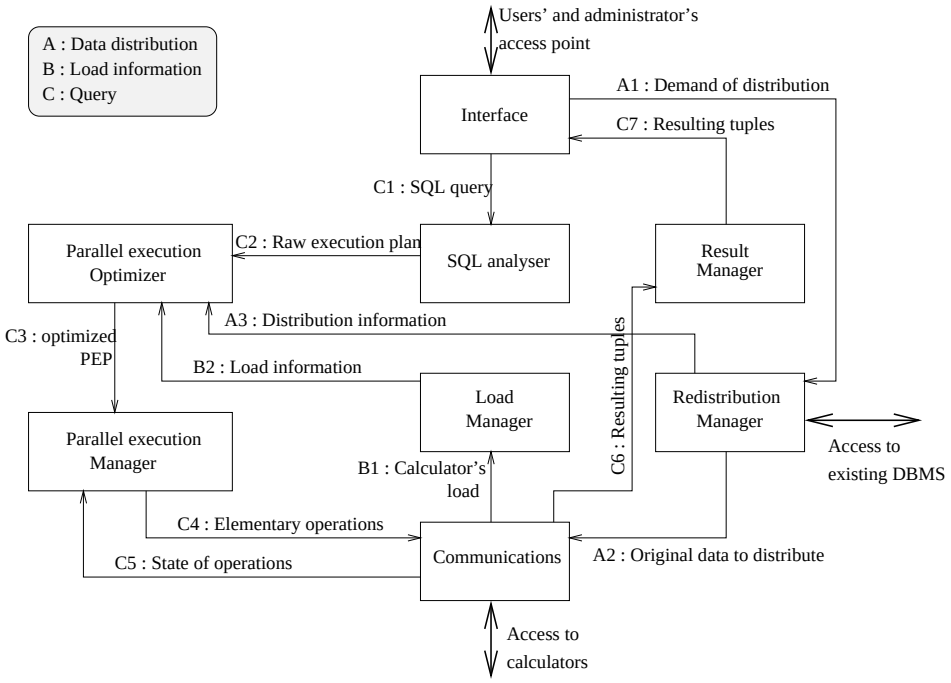


Figure 2: Architecture of the server module

In order to optimize the use of each calculator, the computation module has been multi-threaded. Several computation threads are working on different operators, with a given priority for each. This priority is determine according the precedence of query and of operators. When the thread with the highest priority is waiting for new data (in case of pipelined operators), secondary priority threads can start working. By this way, no waiting delay is lost. Thread switching is limited by using a coarse grain of treatment: tuples are grouped in packets. Once the treatment of a packet is started, the whole packet is treated.

Data flows between storage and computations are not reals flows (i.e. pipes). We use some queuing structures in which packets are referenced. To be more precise, each computation thread has its own set of queues (2 queues). When a thread receives a new operator to compute, it contacts the storage manager (see figure 4 and 5), which in turn referenced the data packets in the queues (e.g. first queue for the building relation, second queue for the probe relation in case of a hash join).

2.2 Redistribution

The coupled evaluator allows a partial distribution both amongst tables and inside tables. This functionality is driven by two main constraints. First, the evaluator is oriented toward read-only querying. Thus, the querying domain can be (at least partially) pre-defined, and extracted data can be limited to the concerned one. Second, the system is supposed to work on a (possibly not dedicated) network of workstations. According to this point of view, some space limits can appear, in which case a choice amongst distributed data must

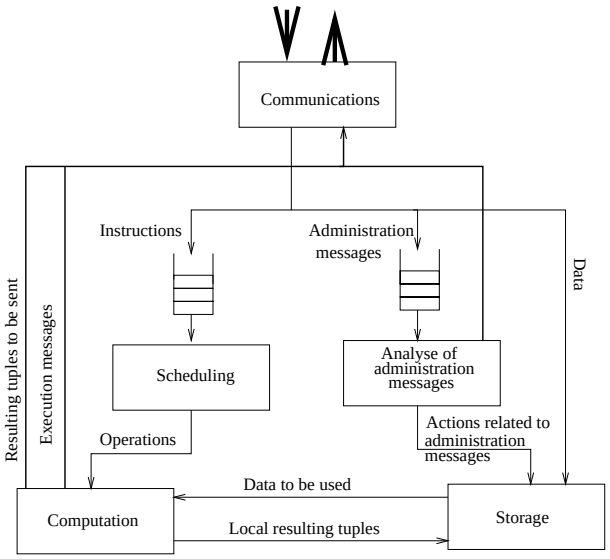


Figure 3: Architecture of the calculator module

be done [11]. Third, in our target applications (e.g. digital libraries), as in most database applications, most queries concern a restricted set of data (usually called "hot data"). So, as the parallel evaluator will be coupled with a sequential DBMS, it would not make sense to duplicate the whole database. Only hot data has to be extracted from the sequential DBMS and distributed over the network.

Such limitations offer two different gains. First, the space utilization is optimized, and second, the extraction delay is limited. Indeed, extraction delays can be a limiting factor in case of a brief use.

Once a distribution is done, we must have a glance to the frequency of the updating procedure. As far as the evaluator is mostly concerned with off-line applications, updates can be delayed according to a given frequency (e.g. once a day), which offers a sufficient "freshness" of data according to the application. Then a second problem appears. The refreshing delay can be limited by updating only the new or modified tuples, if update tracing is allowed by the DBMS.

2.3 Introducing macro-pipelining

Context

Macro-pipelining has been used for some years in parallel applications, mostly in scientific computation [8]. Macro-pipelining consists in sending coarse grains of data between a producer and the corresponding consumer, which in the context of databases corresponds to sending several tuples at the same time between two successive operations. It appears that this technical issue has been poorly studied for shared-nothing systems. We can find some buffered pipelining in hierarchical systems [4], or some tuple-by-tuple pipelining in shared-nothing architectures [9, 14], but no explicit macro-pipelining.
However, a tuple-by-tuple inter-node pipelining brings very high communication overheads, while a macro-pipelining would bring very limited ones, especially

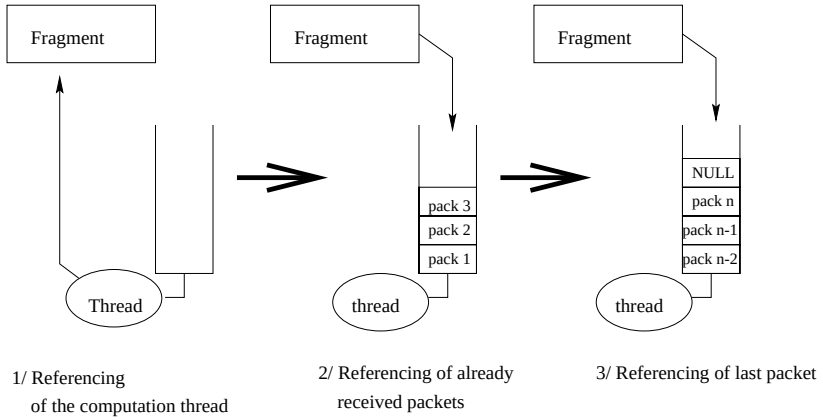


Figure 4: Data packets referencing in queues

on a network of workstations.

Determination of a satisfactory packet size

A satisfactory packet size can be determined through transmission tests, but remains specific to a given configuration. For instance, figures 6 and 7 express the transmission time of one byte with respect to several sizes of packets over a 10base-T Ethernet LAN. Considering this case, we can assume that packets should be at least 5 kilobytes long.

Considering this minimal physical size, the next step consists in proposing a satisfactory effective size depending whether two operations are pipelined or not.

In lack of pipelining, the packets size can be as large as needed. Anyway, middle-sized packets limit the transfer time of the last packet, and by this way lower the total latency between two consecutive synchronized operators (in this case, we do not use pipelining but communication / computation overlapping).

In case some pipelining appears,the first step consists in obtaining an approximation of the number of tuples produced by a join. In case we do not have any statistics, except the cardinality $\|R_1\|$ and $\|R_2\|$ of both relations, we can express the number of tuples N_p produced by the join as a combination of the minimal cardinality and a join factor α as:

$$N_p = \alpha.min(\|R_1\|, \|R_2\|) \tag{1}$$

This estimation is a relatively optimistic one, but it appears to be sufficient in the case of a low skew. Going a step forward, we must introduce the fact that both consumer and producer operators can be parallelized over several nodes. Let p the number of producers and d the number of consumers. If we assume that the distribution is balanced enough, the number of tuples N_{pp} produced by each producer can be estimated as:

$$N_{pp} = \alpha.min(\frac{\|R_1\|}{p}, \frac{\|R_2\|}{p}) = \frac{\alpha}{p}.min(\|R_1\|, \|R_2\|) \tag{2}$$

Thus, if the distribution over consumers is also balanced, the number of tuples N_d received by each consumer is:

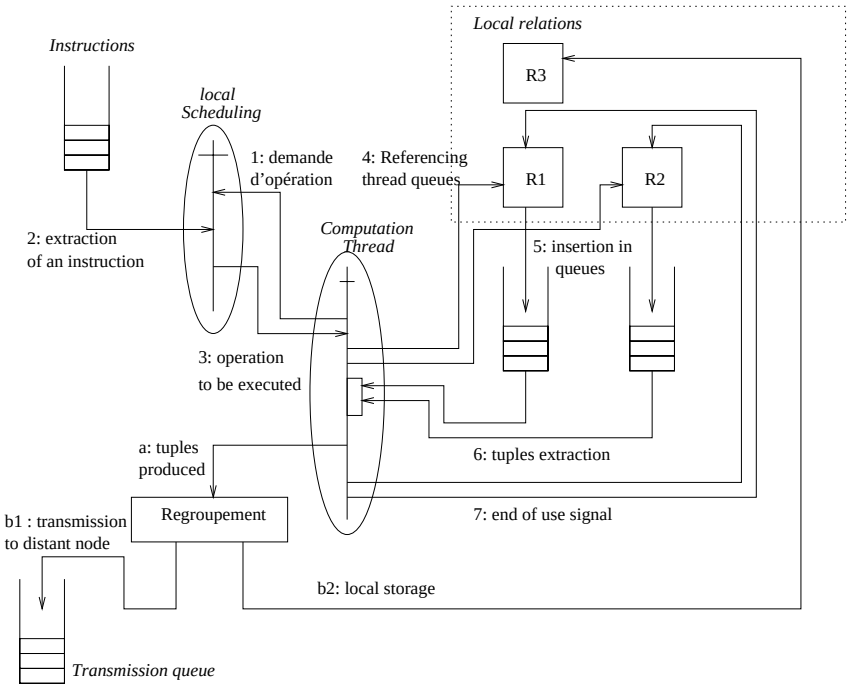


Figure 5: Execution of an operator

$$N_d = \frac{\frac{\alpha}{p}.min(\|R_1\|, \|R_2\|)}{d} = \frac{\alpha}{d.p}.min(\|R_1\|, \|R_2\|) \quad (3)$$

Let S_t the size of the transmitted tuples, then the average volume of data transmitted V_d is:

$$V_d = S_t.\frac{\alpha}{d.p}.min(\|R_1\|, \|R_2\|) \quad (4)$$

We can notice that V_d is the packet size to be used if we plan to send only one packet. Finally, one must consider the properties of the network. Let Min_S the physical minimal size depending on the properties of the network (for instance, 5 kilobytes). The expression of packet size is then given by:

$$S_{rl} = max(\frac{S_t.min(\|R_1\|, \|R_2\|)}{d.p}, Min_S) \quad (5)$$

Thus, Min_S will be used for pipelining, while S_{rl} will be used for synchronized operators.

Threshold

Using fixed size packets appears not to be pertinent in most cases of pipelining. Indeed, an irregular production rate implies an irregular consuming rate, and by the way a slow-down over the whole query. For this reason, we propose to use a flushing threshold, in order to improve the fluidity of the pipelining data flow. This threshold consists in limiting the delay between consecutive packets by allowing the system to send partially-filled packets. The main points consist

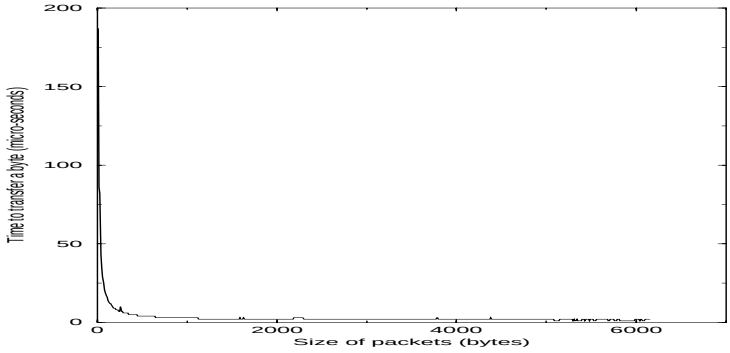


Figure 6: Transfer time (small packets)

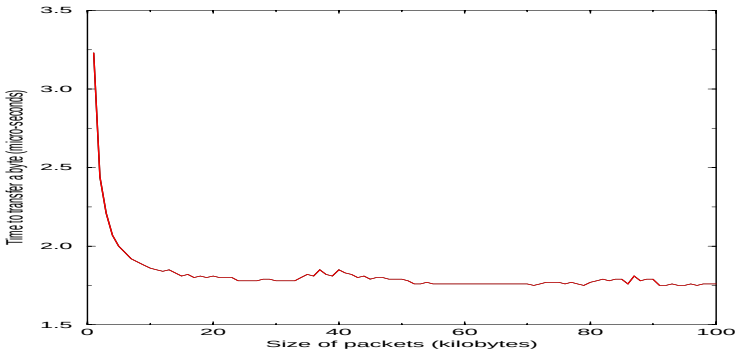


Figure 7: Transfer time (bigger packets)

in i) defining the threshold unit and ii) proposing a satisfying threshold. Concerning the threshold unit, we propose to use the number of incoming packets already treated by the operator since the last flush (i.e. the last data transfer). More precisely, as far as classical join strategies (i.e. no full parallelism [15]) are used, only packets belonging to the probe relation are taken into account.

Using the number of tuples as threshold unit would be more accurate, but this would imply more accounting work. On another side, using packets as accounting unit requires dividing the probing relation into packets. This seems relatively easy, as all probing relations are intermediate results. As it can be seen in figure 8, even if source relations are monolithic (i.e. are not divided into packets), the first operator of a pipeline chain will produce packets, and the second one will then be able to apply a threshold.

ßo we will define the threshold as follows: *If n packets of the probing relation have been treated, then all resulting packets in progress are flushed.*

We could argue that a restricted packet size would be a good alternative to threshold, but this would have a very different meaning. Introducing a threshold guarantees that data is sent at least at a given rate (of packets), but that these packets reach an economic size, while reducing the packet size would raise the transmission cost.

Estimation of the threshold

Estimating the threshold and estimating the packet size can be done the same way. Going back to equation 4, we can deduce that the average number of packets Nl_d transmitted to each consumer is:

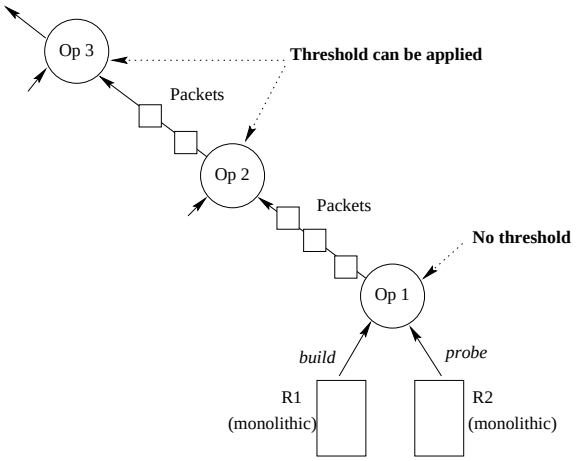


Figure 8: Thresholds and source relations

$$Nl_d = \frac{S_t.\frac{\alpha.min(\|R_1\|,\|R_2\|)}{d.p}}{Min_S} = \frac{S_t.\alpha.min(\|R_1\|,\|R_2\|)}{d.p.Min_S} \tag{6}$$

As explained above, thresholds are used during the probing phase of joins. Thus, they depend on the number of packets N_{probe} of the probing relation. The average number r of "probing" packets used to produce a result packet is then given by:

$$r = \frac{N_{probe}}{Nl_d} = \frac{N_{probe}.d.p.Min_T}{T_t.\alpha.min(\|R_1\|,\|R_2\|)} \tag{7}$$

This equation expresses the average rate of production. The threshold T must then be close to this value:

$$T = \beta.r = \beta.\frac{N_{probe}}{Nl_d} = \frac{\beta}{\alpha}.\frac{N_{probe}.d.p.Min_S}{S_t.min(\|R_1\|,\|R_2\|)} \tag{8}$$

The ratio $\frac{\beta}{\alpha}$ gives the fraction of the average production rate that is used for the threshold. Empirically, through our tests (low skew) it appeared that the best values for $\frac{\beta}{\alpha}$ were between 0.5 (twice the average production) and 1 (the average production).

Our tests revealed that skewed data can suppress the advantage of using a threshold. To be more precise, we could say that, in our context, there is a "good" and a "bad" skew. The "good" skew appears when a few tuples of the building relation match the ones of a packet of the probing one. In this case the production is limited, and the threshold must be used in order to send the few tuples produced. On the contrary, the "bad" skew occurs when the production rate is high. In this case the threshold interferes with the "natural" transmission of full packets.

These interferences come from the fact that the threshold forces a useless flush of packet. To prevent this side effect, we flush a packet if and only if no packet has been naturally sent to the corresponding consumer node since the last flush.

As it will be shown in the next section, using this limitation brought the performance to remain satisfactory in presence of a relatively strong skew.

3 The Enkidu Prototype

Based on the architecture above, we have developed a complete prototype. After a first version programmed in C, we decided to port it under Java, owing to the robustness and portability of this language. Some external components in C, such as the MPO P.E.P. optimizer [6], are currently being adapted through the Java Native Interface.

The Enkidu Prototype consists of about 80 Java classes. Enkidu first aims were the validation of the concept of parallel extension. We also used it as a pedagogic material for teaching parallel databases at a postgraduate level.

In figure 9 we can see the administration interface. Enkidu offers several distribution and execution strategies. It can be used with real data (downloaded from an existing database) or with self-generated data (according to given skew and distribution parameters). It allows the simulation of several concurrent users (we tested up to one hundred concurrent ones), and will soon offer a direct SQL query interface. Enkidu provides several monitoring tool.

Thanks to its Java implementation, Enkidu has already been used under Solaris, Linux and Windows95, allowing us to conduct the tests presented in next section.

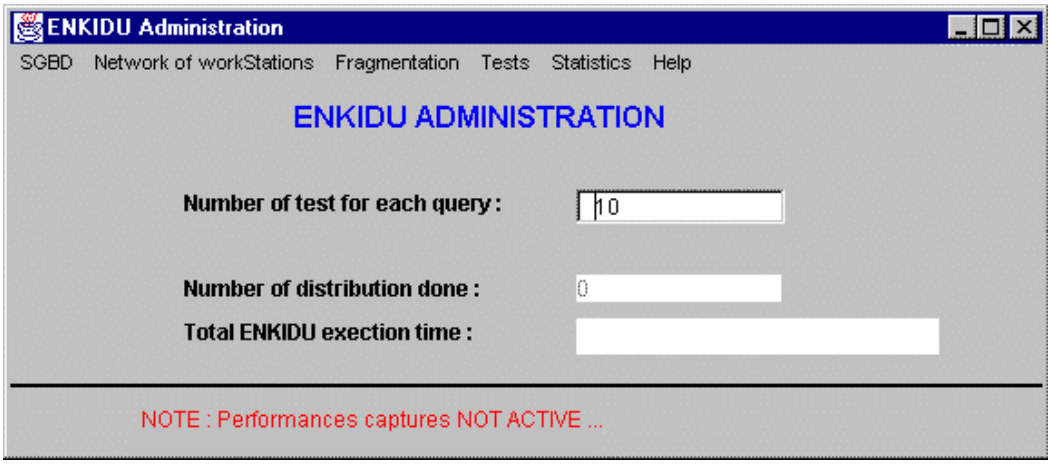


Figure 9: The Enkidu administration interface

4 Measures and analysis

4.1 Thresholds

4.1.1 Underlying hardware

Our tests have been conducted over the Popc machine. This machine, developed by Matra, is an integrated network of PCs. It consists of twelve Pentium Pro

processors running under Linux, with 64 MB memory each, connected by both an Ethernet and a Myrinet [3] network (see table 1). In most of our tests we chose to use the Ethernet network, as it corresponds to a standard company LAN.

Component	**Capacity / type**	**Price in $ (estimation)**
CPU	Pentium Pro	200
RAM	64 Mo	150
Disk	2.5 Go	150
Network # 1	Ethernet	50
Network # 2	Myrinet	1 500
Total	all * 12	ab. 25 000

Table 1: Description and estimation of the Popc Machine

4.1.2 Measurements

The tests presented in this section explore the effectiveness and limits of the threshold strategy. For all these tests, pipelined operators are placed on different nodes, and no intra-operator parallelism is used. Such requirements are used in order to maximize inter-node communications. In case some intra-node pipelining would occur, we can notice that packet management can be used as a buffering strategy.

We conducted two consecutive tests on pipeline chains of different length. On each case we used two kinds of relations. The first join is done on R_1 and R_2 where R_1 is the build relation and R_2 is the probe one. The cardinality of R_1 is 1 000 and the one R_2 is 10 000, such that one tuple over ten of R_2 matches a tuple of R_1. The table have been generated in order to offer no skew. So the average production rate is of 10 packets treated for 1 packet produced. The next join are done by using the result relation as the probing one. Building relations are similar to R_2. According to this structure, the influence of thresholding mainly influence the first join.

The packet size has been expressed in terms of number of tuples. Each packet is 100 tuple large. This relatively small size allows a satisfactory number of packets for R_2 (100 packets) and the intermediate results (> 10 packets).

In order to obtain valuable measures, each point of the following graphics represents the mean value of 100 measures.

The first chain (see figure 10) consists of 2 pipelined operations. On this figure, as for the following ones, the 0 value on the threshold axis indicates that no threshold is used (i.e. the reference value). We can notice that the gain is negative for a threshold of one packet treated. This comes from the multiplication of the number of packets sent (100 packets instead of 10). For greater thresholds we can notice that gains are better, and reach 10 percents for 4, 6 and 7. Gains are much more limited when the threshold is near the mean rate (10).

The second chain (see figure 10) is 6 operators long. In this case we can notice that the loss for a threshold of 1 is much more limited than in the preceding

test (25 percents vs. 50 percents). On the contrary, gains are higher (between 15 and 20 percents) for thresholds between 4 and 7.

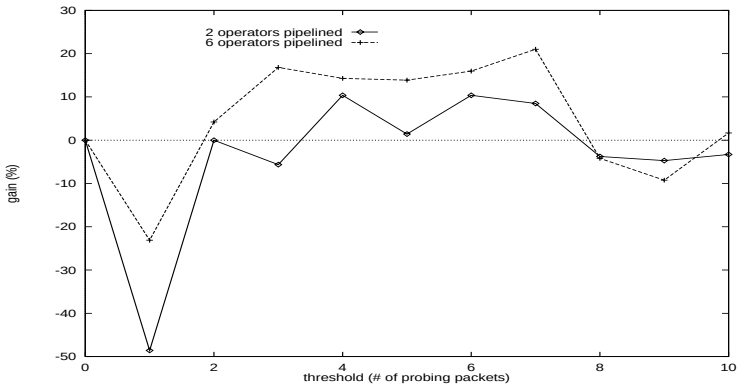


Figure 10: Threshold without skew

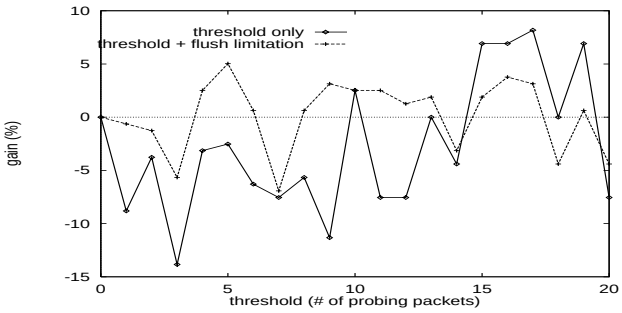


Figure 11: Threshold with zipf skew

In order to illustrate the interest of a flush limitation in case of a noticeable skew, we also conducted the tests presented in figure 11. For this test we used some skewed data by applying a zipf distribution to the probe relation of the first operator and to the build relations of the following ones. The zipf factor was 0.3, and by this way the mean production rate was around 16 incoming packets for one out-coming packet. Of course, this mean rate has poor meaning, as the real rate goes from less than 1 to more than 20. We can notice that a threshold-only execution brings more loss than gains. Adding the flush limitation limits the use of threshold to the phase of low rate. By this way, loss are limited to few points.

4.2 Combining Enkidu with an existing DBMS

In a second time we would like to show the ability of a combined use of Enkidu with the existing DBMS. These tests are based on the **Claude Bernard Data bank**, which is a professional database of medicines available in France [12]. The relatively reduced size of this database (some Megabytes) is compensated by the fact that no index was used during the test. This option was retained in order to simulate any query (i.e. no pre-optimized query). Our queries were

run both on a DBMS (Oracle 7 on a Bull Estrella – PowerPC, 64 MB RAM, AIX) and on Enkidu with 1 and 2 calculators (server on a Pentium 90, 48 MB RAM, calculators on Pentium 166 MMX, 64 MB RAM, Windows 95, networking through Ethernet 10 Base-T). The poor configuration of Enkidu has been chosen in order to simulate a recycled or non-dedicated LAN.

We run a set of queries for 1 to 10 users on our Oracle (users are simulated by a forked pro-C program) and on Enkidu (users are simulated by threads running on the Enkidu server). The figure 12 presents the results of this test. The time indicated corresponds to the global response time. It appears that Enkidu offers very good performances both in terms of execution time (equivalent to Oracle with 1 calculator) and in terms of speed-up (nearly linear between 1 and 2 calculators).

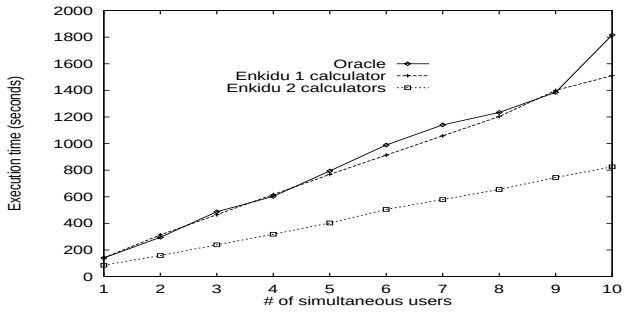


Figure 12: Enkidu vs. oracle

Finally, figures 13 and 14 propose a simulation of a combined use of Oracle and Enkidu. We distribute queries depending on the performance measures and on the current load of the whole system. Figure 13 and 14 respectively indicate the expected response time and the load balance between Oracle and Enkidu. Thus, coupling 2 calculators with Oracle allows to improve the response time by up to a factor 3.

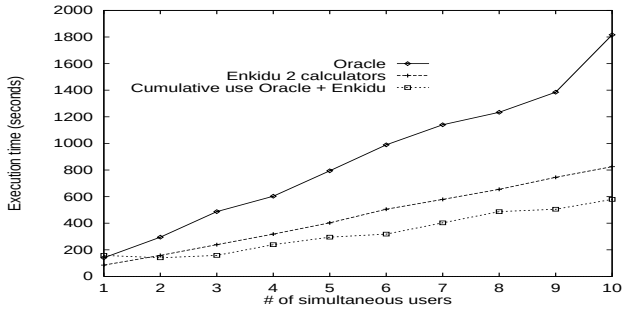


Figure 13: Cumulative use of Oracle and Enkidu

5 Related work

If we focus on networks of workstation, or in a wider way on shared-nothing parallel systems, we can see that many parallel DBMS have already been proposed.

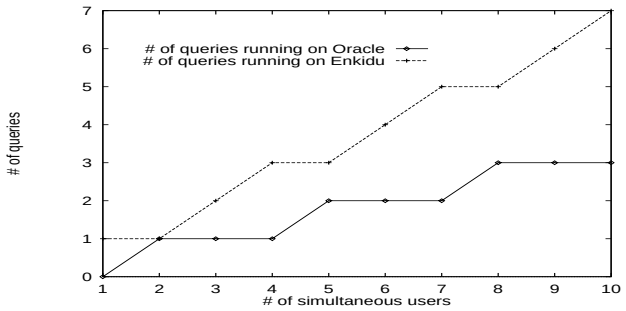


Figure 14: Load balance Oracle / Enkidu

Let us have a first glance at commercial products. Oracle Parallel Server [1] seems now to reach its maturity. This product, while said to be working on workstations networks, is nevertheless working in a shared-disk way, and has a mostly centralized control. IBM DB2 Parallel Edition [2] is, just like Oracle PE, a parallel port of a sequential product. DB2 offers limited distribution techniques (mostly hashing), and is also supposed to work on big configuration, like SP2 or Sun SMPs. Last, Informix XPS [13], which is frankly shared to be "workstations oriented", is in reality supposed to work on SMP machines. Anyway, this third product is the most complete one.

From the research point of view, we can first cite Gamma [9], which, as a database machine system, has been one of the first ancestors of modern shared-nothing parallel DBMS. Gamma was anyway working on a dedicated machine and moreover on a dedicated operating system. Midas [5] is a adaptive port of a sequential relational system to a parallel one, with a noticeable care parallelism concepts, and a special orientation towards workstations. It anyway remains a parallel DBMS. Volcano [14] can be seen as a parallel extension, because of its orientation towards query execution. One limit of Volcano resides in the lacks of its optimizer for shared-nothing systems. MPO, the optimizer used by our evaluator [6], proposes optimized execution plans based on serialized bushy trees, and adapted to resource constraints.

Another limit of existing prototypes comes from their stand-alone nature, while our architecture is specifically constructed in order to work in a coupled mode with a sequential DBMS. From another point of view, many of these prototypes just focus on a precise aspect of parallel evaluation of queries, while Enkidu is a integrated solution.

Finally, while most prototypes are system-specific, our implementation is particularly portable, and forthcoming Just In Time compilers generations will still make its performances grow.

6 Conclusion

In this paper we introduced a novel software architecture for coupling a pre-existing sequential DBMS with a parallel query evaluator implemented on a network of workstations. This architecture has been especially designed to optimize the communication schemes between calculators. A portable prototype, called Enkidu, has been developed, which implements the proposed concepts.

Experiments on a real medical database have shown the pertinence and the efficiency of the heuristics and parallelization strategies introduced.

So, using Enkidu as an extension to a sequential DBMS allows a definitive improvement of the performance of the DBMS for processing read-only applications at a very effective cost, since no expensive additional hardware or software is required.

Further work will be focused on tuning the Enkidu prototype for handling large indexes used in documentary databases.

References

[1] R. Bamford, D. Butler, B. Klots, and N. MacNaughton. Architecture of Oracle Parallel Server. In *Proceedings of the* 24^{th} *VLDB Int'l Conference*, pages 669–670, New York City, NY, USA, August 1998.

[2] C.K. Baru, G. Fecteau, A. Goya, H. Hsiao, Jhingran A., Padmanabhan S., G.P. Copeland, and Willson W.G. DB2 Parallel Edition. *IBM Systems Journal*, 34(2):292–322, 1995.

[3] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.K. Su. Myrinet - a gigabit-per-second local-area network. *IEEE-Micro*, 15:29–36, 1995.

[4] L. Bouganim, D. Florescu, and P. Valduriez. Dynamic load balancing in hierarchical parallel database systems. In *Proceedings of the* 22^{nd} *VLDB Int'l Conference*, pages 436–447, Bombay, India, September 1996.

[5] G. Bozas, M. Jaedicke, A. Listl, B. Mitschang, A. Reiser, and S. Zimmermann. On transforming a sequential sql-dbms into a parallel one : First results and experiences of the MIDAS project. In L. Bougé, P. Fraignaud, A. Mignotte, and Y. Robert, editors, *EuroPar'96*, volume 1124 of *Lecture Notes in Computer Science*, pages 881–886, Lyon, August 1996.

[6] L. Brunie and H. Kosch. ModParOpt : a modular query optimizer for multi-query parallel databases. In *ADBIS'97*, St Petersbourg, RU, 1997.

[7] L. Brunie and H. Kosch. Optimizing complex decision support queries for parallel execution. In *PDPTA*, Las Vegas, AZ, USA, July 1997.

[8] F. Desprez. A Library for Coarse Grain Macro-Pipelining in Distributed Memory Architectures. In *Proceedings of the IFIP 10.3 Conference on Programing Environments for Massively Parallel Distributed Systems*, pages 365–371, Monte Verita, Ascona, CH, April 1994.

[9] D.J. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna. GAMMA : A High Perfomance Dataflow Database Machine. In *Proceedings of the* 12^{th} *VLDB Int'l Conference*, pages 228–237, Kyoto, August 1986.

[10] M. Exbrayat. *Evaluation Parallèle de requêtes relationnelles sur réseau de stations : Le système Enkidu.* Phd. thesis, comp. sc., INSA de Lyon, Villeurbanne, France, January 1999.

[11] M. Exbrayat and H. Kosch. A Parallel Extension for Existing Relationnal Database Management Systems. In *BIWIT'97*, pages 75–81, Biarritz, July 1997.

[12] A. Flory, C. Paultre, and C. Veilleraud. A relational databank to aid in the dispensing of medicines. In J.H. Van Bemmel, M.J. Ball, and O. Wigertz, editors, *MEDINFO '83*, pages 152–155, Amsterdam, 1983.

[13] B. Gerber. Informix On Line XPS. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, volume 24 of *SIGMOD Records*, page 463, San Jose, Ca, USA, May 1995.

[14] G. Graefe. Volcano, an extensible and parallel dataflow query evaluation system. *IEEE TKDE*, 6(1):120–135, February 1994.

[15] A.N. WILSCHUT, J. FLOKSTRA, and P.M.G. APERS. Parallel evaluation of multi-join queries. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD Int'l Conference*, volume 24 of *SIGMOD Records*, pages 115–126, San Jose, Ca, USA, May 1995.