Parallel Evaluation of Relational Queries on a Network of Workstations
Lionel Brunie, Matthieu Exbrayat, Andre Flory

To cite this version:

HAL Id: inria-00073036
https://hal.inria.fr/inria-00073036
Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Parallel Evaluation of Relational Queries on a Network of Workstations

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

No 3638
mars 1999

THÈME 1
Parallel Evaluation of Relational Queries
on a Network of Workstations

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

Thème 1 — Réseaux et systèmes
Projet ReMap

Rapport de recherche n° 3638 — mars 1999 — 16 pages

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.

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

(Résumé : tspv)

The LIP Laboratory is jointly supported by ENS Lyon, CNRS and INRIA (UMR 5668).

* 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, Ecole Normale Supérieure de Lyon, 46 Allée d’Italie, F-69364
Lyon Cedex 07, France. Email: Matthieu.Exbrayat@ens-lyon.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l’Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52
Évaluation parallèle de requêtes relationnelles sur réseau de stations


Mots-clé : Base de données relationnelle, parallélisation de requêtes, réseau de stations, macro-pipelining.
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 [BBKM98, Ger95]. 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).

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 [Exb99] 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.

![Figure 1: General overview](image1)

### 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).

![Figure 2: Architecture of the server module](image2)

**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).

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 [BK97b]. 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.

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 real 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).

RR n° 3638
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 be done [EK97]. 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 [Des94]. 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 [BFV96], or some tuple-by-tuple pipelining in shared-nothing architectures [DG94, Gra94], 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 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 $\alpha$ as:

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

$$N_{pp} = \alpha.min\left(\frac{||R_1||}{p}, \frac{||R_2||}{p}\right) = \frac{\alpha}{p}.min(||R_1||, ||R_2||)$$  \hspace{1cm} (2)

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

$$N_d = \frac{\alpha}{d}.min(||R_1||, ||R_2||) = \frac{\alpha}{d.p}.min(||R_1||, ||R_2||)$$  \hspace{1cm} (3)

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

$$V_d$$
\[ V_d = S_t \cdot \frac{\alpha}{d,p} \cdot \min(\|R_1\|,\|R_2\|) \]  \hspace{1cm} (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 \( \text{Min}_S \) the physical minimal size depending on the properties of the network (for instance, 5 kilobytes). The expression of packet size is then given by:

\[ S_{rl} = \max \left( \frac{S_t \cdot \min(\|R_1\|,\|R_2\|)}{d,p}, \text{Min}_S \right) \]  \hspace{1cm} (5)

Thus, \( \text{Min}_S \) 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 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 [WFA95]) 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.

**Figure 8: Thresholds and source relations**

So 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 \( N_{l_d} \) transmitted to each consumer is:

\[
N_{l_d} = \frac{S_{t, \alpha, \min(\|R_1\|, \|R_2\|)}}{d.p \cdot \min_S} = \frac{S_{t, \alpha, \min(\|R_1\|, \|R_2\|)}}{d.p \cdot \min_S}
\]  

(6)

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

\[
r = \frac{N_{\text{probe}}}{N_{l_d}} = \frac{N_{\text{probe}} \cdot d.p \cdot \min_T}{T_{t, \alpha, \min(\|R_1\|, \|R_2\|)}
\]

(7)

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

\[
T = \beta r = \beta \frac{N_{\text{probe}}}{N_{l_d}} = \frac{\beta}{\alpha} \frac{N_{\text{probe}} \cdot d.p \cdot \min_S}{S_{t, \alpha, \min(\|R_1\|, \|R_2\|)}
\]

(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 [BK97a], 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.

![Enkidu Administration Interface](image)

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 [BCF+95] network
(see table 1). In most of our tests we chose to use the Ethernet network, as it corresponds to
a standard company LAN.

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

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](image)

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 [FPV83]. 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).

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.

![Graph showing execution time vs. # of simultaneous users for Oracle, Enkidu, and Cumulative use of Oracle + Enkidu.](image1)

**Figure 13: Cumulative use of Oracle and Enkidu**

![Graph showing # of queries running on Oracle vs. # of simultaneous users for Oracle and Enkidu.](image2)

**Figure 14: Load balance Oracle / 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. Let us have a first glance at commercial products. Oracle Parallel Server [BBKM98] 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 [BFG+95] 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 [Ger95], 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 [DGG+86], 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 [BJL+96] 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 [Gra94] 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 [BK97a],

---

*INRIA*
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 an 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


