A multithreaded SQL service

Christophe Cérin
Université de Picardie Jules Verne
LaRIA, Bat Curi, 5 rue du moulin neuf
F-80039 Amiens cedex 1- France
cerin@laria.u-picardie.fr

Michel Koskas
Université de Picardie Jules Verne
LaMFA/CNRS UMR 6140, 33 rue St Leu
F-80039 Amiens cedex 1- France
koskas@laria.u-picardie.fr

Jean-Luc Gaudiot
The Henry Samueli School of Engineering
Electrical Engineering and Computer Science, zotcode 2625
University of California, Irvine
Irvine, CA 92697-2625
gaudiot@uci.edu

May 8, 2004

Abstract

In this paper, we introduce a novel SQL service based on radix tree data structures and we show how to multithread the service in order to execute it on distributed platforms. The SQL service is currently developed in the context of SMP platforms but should be deployed in the future on grid platforms. The development of the SQL service is part of the Grid Explorer project[1]. The project (French part) aims at building a grid emulator to study the behavior of grids. One need in the project corresponds to a database software to store experimental conditions and to analyze traces of execution (for instance to find, in "real time," frequent episodes for the CPU load metric in order to predict the future state of the grid). We plan to use our SQL service on an SMP platform for this purpose.

Keywords: multi-threading at user level, data structures for databases, concurrent and parallel algorithms, database components.

1 Introduction

A family of novel sequential algorithms and data structures designed to offer the same services as any database product is introduced in [KK03]. The services run on any affordable computer, for instance PCs with AMD or Intel processors. The novelty resides in the fact that the service does not use 'btree-like' data structures, hence novel algorithms to create or to modify tables and to answer queries. In short, we use an 'inverted' table representation (also called denormalized basis), and an efficient representation of sets of integers which help us in processing arithmetic operations, such as intersection or union operations. This representation employs sets of integers that have to be sorted in many ways (with several comparison functions) and as fast as possible. The problems and solutions for sorting are covered in [CKFJ04].

[1] This work is supported by ACI Masse de Données and the Grid Explorer project funded by FSE for the French ministry of education and research. It is also partly supported by the National Science Foundation under Grants No. CSA-0073527 and INT-9815742. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF

The paper describe our approach to intorduce parallelism to the Zêta-Data project which is part of the Grid Explorer project[2]. The target architectures are SMP machines and Grid infrastructures.

To evaluate the performance of database products, the accepted practice has been to use the benchmark provided by the Transaction Processing Council (TPC)[3]. The TPC benchmark suite consists in eight tables (Lineitem: 6 million lines, Orders: 1.5 million lines, Customer: 150,000 lines, Partsupp: 800,000 lines, Part: 200,000 lines, Supplier: 10,000 lines, Nation: 25 lines and Region: 5 lines) so that the database is at least 1GB in size.

The first and complete sequential implementation improved performance by at least a factor of 8 compared with commercial products and notably for computational demand queries including a query with many answers (found 90% of the time), a query of a very combinatoric type, a correlated sub-query, a query with many computations on the data matching the query, insertion or deletion of 10% of the database. A cartesian product of two tables (one with 6 million lines - the "lineitem" table in the TPC - and the other with 1.5 million lines - "orders" table of the TPC) was the last query investigated.

The organization of the paper is as follows. In Section 2 we recall briefly the two needs motivating our work. Section 3 exposes the main data structures and explains where parallelism could be introduce in the SQL service. We also introduce the architecture, in terms of modules, of our SQL service. Section 4 introduces our choices with multithreading tools (at the user level) and we give experimental results. Section 5 concludes the paper.

2 Motivations

GRID and peer-to-peer systems store and exchange huge amounts of data: the size is on the order of petabytes (10^{15} bytes) and the average aggregate throughput is on the order of terabytes/s (10^{12} bytes/sec). The storage and analysis of data on such large scale presents significant research challenges.

The goal of our Data Grid Explorer is to build an emulation environment to study large scale configurations. Today, it is difficult to evaluate new models for data placement and caching, network content distribution, peer to peer systems, *etc.*

Recent research efforts have included writing simulation environments from scratch, employing detailed packet-level simulation environments, local testing within a controlled cluster setting, or deploying live code across the Internet or a Testbed. Each approach has a number of limitations. Custom simulation environments typically simplify network and failure characteristics. Packet-level simulators add more realism but limit system scalability to a few hundred simultaneous nodes. Cluster-based deployment adds another level of realism by allowing the evaluation of real code, but unfortunately the network is highly over-provisioned and uniform in its performance characteristics. Finally, live Internet and Testbed deployments provide the most realistic evaluation environment for wide-area distributed services. Unfortunately, there are significant challenges for deploying and evaluating real code running at a significant number of Internet sites. The main interest of emulation is the ability to reproduce experimental conditions and results.

The project Data Grid Explorer aims at implementing a large scale emulation tool for the researcher communities of a) distributed operating systems, b) networks, and the users of Grid or P2P systems. This large scale emulator consists of a database of experimental conditions, a large cluster of 1000 PCs and tools to control and analyze experiments.

The project includes 8 studies concerning the instrument itself and 16 others that make use of the instrument. It is structured horizontally by transverse working groups, namely:

[2] See: http://www.lri.fr~fci/GdX

[3] http://www.tpc.org

- Infrastructure
- Emulation
- Network
- Applications

The infrastructure task relates to the organization of the platform and the software to allow experiments to be carried out. In particular, it will define the system and hardware, establish access methods, and describe how resources are to be shared. This task will place an emphasis on incorporating flexibility into measurement techniques.

The emulation task aims at the creation of a tool that is half way between simulation, which is purely software based, and experimentation that is carried out *in-situ*. It will provide a facility for a wide range of studies of distributed systems, and this at all architectural levels: large scale communication configuration, such as the Internet, or distributed applications, such as the Web, P2P rings, *etc.* This topic will define the hardware and software equipment required to emulate the behavior of network components. It will also define how to configure the platform for the particular context of a study, and how to place the monitoring equipment. Finally, it will focus on the re-creation of realistic experimental conditions. The emulation task is central and will be used by many others projects.

The network task concerns all projects related to the design, the development, the evaluation, and the experimentation on new architectures and communication protocols, aiming at improving communications on the grid or on the Internet. These improvements concern performance, quality of service, availability, security, flexibility, manageability, etc. The idea is to propose tailored solutions, adapted to each of the various needs. Note that the network experiments will use scenarios resulting from the application task and will rely on an infrastructure that emulates the corresponding scenario.

The applications task will study the adaptation of application to the Grid and P2P infrastructures. It will target applications from varied fields: scheduling, bio-computing, databases (the Zêta-Data project), *etc.* The main goal is to anticipate the performance problems that would be encountered in large scale configurations. This task will allow those working on the other tasks to more precisely understand the problems of adapting such applications, and should help them develop better solutions as a result.

In summary, this platform should allow: researchers in computer science to carry out simulation/emulation on large scale configurations (to study security, scheduling, performance evaluation, fault tolerance, etc.), researchers on the GRID to test the interconnection of large clusters researchers in networks to carry out experiments on a platform of significant size to study the impact of Internet topology on communications, and researchers from other disciplines (physics, biology, *etc.*) to test their applications on the GRID.

As seen previously, we need a database layer to store the experimental conditions that could be played on the emulator. We propose to use our SQL service to accomplish the task. For scheduling purposes, we also propose to explore the performance of our tool when faced with tailored algorithms to mine results of experiments. Since our sequential SQL service has greater performance than many other similar tools, we surmise that it can also serve in mining data, for instance to find the best CPU to place future tasks. This last topic is not explored in this paper. We concentrate on the development of a multithreaded version running on an SMP machine dedicated to store the experimental conditions, the activities (CPU loads, network traffic...) of nodes, and the results of the experiments.

More precisely, the development plan of the Zêta-Data project is depicted in Figure 1. The vertical branch on the left of Figure 1 corresponds to the introduction of multithreading, starting from coarse grain multithreading (parallelization at the level of procedures) to fine grain multithreading (parallelization at the instruction level for the data structures construction). The right branch corresponds to the introduction of specific solutions for disconnected operations, data accesses via redundancy techniques....

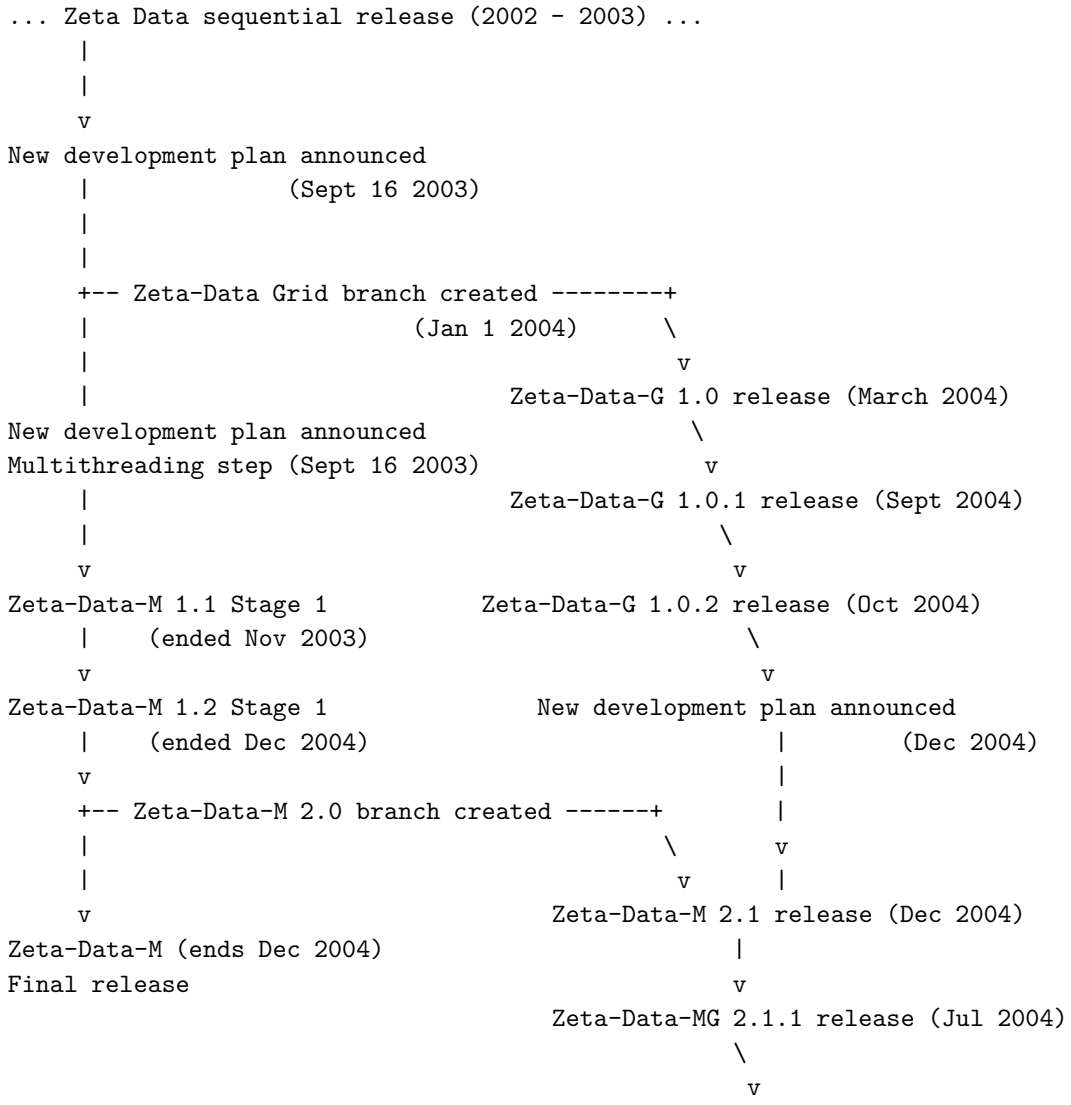


Figure 1: Development plan of the Zêta-Data project

It can also be seen on Figure 1 through the arrow crossing from left to right that we plan to merge the two developments.

So, the Zêta-Data-M 1.1 software is now multithreaded with pthread and variants libraries (see Figure 1). Only the engine part (the module answering to a query) is multithreaded. At the present time, we have multithreaded coarse grain tasks. Future developments will include a fine multithreading of all the modules in a sens we will introduce in sub-section .

3 Data structures and multithreading

We introduce now the core data structure of our sequential database management system and we show potential parallelism in different treatments related to fundamental algorithms, in particular for the intersection operation that appears in the implementations of BETWEEN or AND clauses for instance.

3.1 Some algorithms in the Zêta-Data project

Our algorithms in the SQL service basically sort arrays and deal with heaps. We need some terminology from database systems to clarify some situations. Usually, in hierarchical databases (databases "made" of several tables) the different tables are linked by couples of primary keys – foreign keys.

Definition 1 (Superkey) *A superkey is any column or set of columns that uniquely identifies each record in a table. Not every superkey is a good candidate key.*

Definition 2 (Candidate) *A candidate key is a superkey containing the minimum number of columns to uniquely identify each record in a table. Not every candidate key is a good primary key.*

Definition 3 (Primary key) *A primary key is the candidate key used to uniquely identify each record in a table.*

Definition 4 (Foreign key) *A foreign key is a column or set of columns in one table that matches a candidate key in another table.*

Any table must have a *primary key* even it is implicit: the index or address of the physical record of a line is indeed a primary key. When tables are stored on disks and if we have to sort the tables, is convenient to associate keys with the line addresses and to sort the couple (keys, line address) instead of moving data which is too costly. Assume now that we have two "sorted lists" of (key, line address) pairs corresponding to two tables.

As mentioned above, our algorithms use a denormalized representation of the database. This means that each table of the database is expanded in only one table of the database. Indeed, each table is "expanded" in different tables (because in practice, tables cannot fit entirely in memory) in such a way that a foreign key matches its primary key. Then each column is treated as if it were alone in the table (see [KK03] for full explanations).

3.2 An example

Let us consider the following database, composed of three tables, namely Accident, Customer, and Insurance tables. The Accident table has seven lines as one can see on Figure 2. When considering

Client Id	Contract Date	Max Amount	Seller	Kind of Cont.	Min Ref.	Acc. Id
1	12-21-1992	450,000	2	House	900	1
2	02-24-2000	12,000	17	Car	830	2
3	11-28-1996	230,000	11	House	1,350	3
4	05-30-2001	780,000	2	House	2,400	4
1	07-17-1992	27,500	3	Car	912	5
1	04-13-1998	1,000,000	2	Family	100	6
2	09-11-1999	830,000	2	House	11,000	7

Figure 2: The root-table: the Accident table.

Accident table, its primary key is "Acc-id"," which is the name of the last column of Accident Table. A foreign key is "Client id."

The Customer table has four lines: see Figure 3, while the Insurance table has two lines: see Figure 4

Name	Adress	Ins. Id	City	Country	Client Id
Valesa	zzz	1	Warsaw	Poland	1
Thatcher	xxx	2	London	Great Britain	2
Profi	yyy	2	Roma	Italy	3
Carlis	ttt	1	Barcelona	Spain	4

Figure 3: An intermediate table: the Customer table.

Name	Capital	Localization	Profit	Ins. Id
Tartempion S.A.	12,384,948	USA	3,123,123	1
Truc-Muche Inc.	21,987,890	Europe	1,123,341	2

Figure 4: A leaf table: the Insurance table Compagny.

In this case, one may consider that Insurance and Customer are sub-tables of the Accident table because of the links made of the pair foreign key - primary key.

Now, consider the Insurance table. It has several columns, and each of them is treated separately: for each of these columns, one builds its thesaurus and for each word of the thesaurus we build the set of lines indexes at which it occurs.

For instance, the column "Kind of Contract"'s thesaurus is `House, Car, Family` and the sets of line indexes are: `House` occurs at indexes 1, 3, 4 and 7, `Car` occurs at indexes 2 and 5 and `Family` occurs at index 6.

Now, let us expand the sub-tables. For instance, the column `City` of the table customer has thesaurus `Varsaw, Roma, Barcelona`, and `London`. The line indexes the word `London` occurs in the Accident table are hence 2 and 7.

Finally, we get a full description of the database by computing the thesaurus of each column of each table and its sub-tables and the line indexes at which each word occurs.

While computing these thesauruses and sets of integers, one has to sort couples containing a word of the thesaurus and a line index at which this word occurs. Hence, we need also a multi-criteria sorting program. This explains why we do not use radix trees in [CKFJ04] to achieve this sorting. Indeed, the radix tree sorting algorithms do not seem to run efficiently with pairs.

Now, let us return to the building of the indexes. While building or modifying the table indexes, one has to sort the fields of each column of the expanded tables. This means for instance that a table with 5 lines (for instance the "region" table of the TPC) may be expanded in a table with 6 millions lines (for instance the "lineitem" table of the TPC). Thus, one has to sort the fields of each column of the table "region" expanded in the table "lineitem". This means that one has to sort arrays of 6 million items with only 5 different values. Special techniques have been devised in [CKFJ04] for this purpose but they are out of the scope of this paper.

3.3 Set operations

The intersection operation of two sets of integers of size n, m is also of great importance in our project.

If one intersects two sets of integers of size n, m by looking if each element of the first set appears in the unsorted second set, the total cost will be $\mathcal{O}(nm)$.

Another way to intersect two sets of integers is to sort the two sets and read simultaneously each of the sets. Hence the cost of sorting is $\mathcal{O}(n \log n + m \log m)$ and reading them costs $\mathcal{O}(n + m)$. Therefore, the cost of this operation is $\mathcal{O}(n \log n + m \log m)$.

If one sorts only one of the two sets and seeks for every element of the other set in the sorted one, the cost is $\mathcal{O}(n \log n)$ to sort, say, the first one, and the cost of attempting to find every element of the second one is $\mathcal{O}(m \log n)$. Hence the total cost is $\mathcal{O}(n \log n + m \log n) = \mathcal{O}((n + m) \log n)$. If $n \gg m$ this last method to compute the intersection is cheaper than the two others. We implement intersection in such a way in our sequential release, in memory, and by pipelining disk read/write operations: it is obvious that in our case we need to deal with disks since the data does not fit in memory.

Note also that the previous algorithm can be parallelized in a straightforward way: one thread performs the sort on the smallest set, then p threads are activated (if we have p elements in the other set) to check if elements are present in the first set.

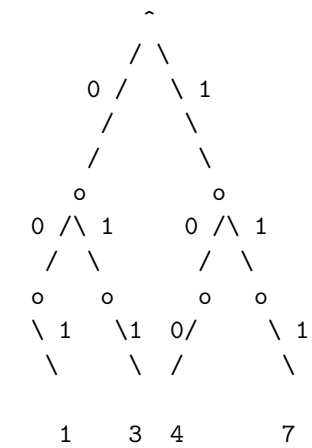
We call such algorithm a *fine grain multithreaded algorithm* because it works at the level of the data structures.

In fact, the problem is more difficult to handle because sets of integers are implemented with suffix trees: we should implement the thread management in a very careful way when updating the tree containing the result of the intersection.

Let us go back to the example to fix the main difficulties. Let S be a set of integers written in basis $b = 2$ for instance (it is convenient to chose as basis a power of 2). The integers may be represented in a radix tree.

A radix tree is a tree which allows to store a set of words over an alphabet A of same length (here the alphabet is the set of digits $0 \ldots b - 1$). Consider the Accident Table depicted on Figure 2 and the "Kind of Cont" column. The thesaurus of the column is $\{\mathrm{House}, \mathrm{Car}, \mathrm{Familly}\}$. The lines where 'House' appears are $\{1, 3, 4, 7\}$, the lines where 'Car' appears are $\{2, 5\}$ and the line where 'Family' appears is $\{6$.

A radix tree representation of set $\{1, 3, 4, 7\}$ is simply:



Suppose that we have to check if $5 = 101_2$ key is present is the previous tree. We descend along the tree until we encounter the prefix 10 after that, since the last bit (1) is not present, we conclude that 5 does not belong to the intersection.

The previous scheme explains also how to answer a query with an AND clause, for instance:

```
SELECT ALL
FROM Accident
WHERE
    = Accident KindOfCont 'Car'
    AND
    >= Accident MaxAmount 12,000
GROUP {NULL}
```

The AND clause is not problematic for the multithreaded implementation since we do not modify the tree. Problems occur with the OR clause or the union operation because we have to modify the tree.

According to our previous example, if we have to take the union of $\{1, 3, 4, 7\}$ with $\{5\}$ we have to add a 'branch' corresponding to the right 1 bit in 101. If concurrently we take the union (insert) of an integer with the 101 prefix, a "race condition" problem occurs. We have not yet implemented such multithreaded operations. It will be accomplished during the next months of development.

3.4 Radix trees on disks

Another source for multithreading is related to the way we store radix trees (indexes of tables) on disks. Assume we have to store on a disk the set $\{0, 1, 3, 4, 7, 11\}$. The tree corresponding to the set is depicted in Figure 5. On this figure, we draw a dashed line when a node has no left or right children. We store on disk the sequence of bits 11111101111010111010 obtained according to the following principle:

- We start from the root of the tree. We write 1 if the node has a left child (or 0 if not - this case is depicted by a dashed arrow on Figure 5) and then we write the representation of the left child (if any, else we have a 0) followed by the representation of the right child (if any, else we have a 0)

Currently we use a tree hierarchy with tree height set to 20 and two levels in the hierarchy. On Figure 5 we observe that each tree corresponds to a bit vector of about 256Kb in size if the tree is complete, *i.e.,* it contains 256K integers. Note also that with two levels in the hierarchy we handle integers coded on 40 bits, *i.e.,* we can potentially deal with 1099511627776 different items in the tables.

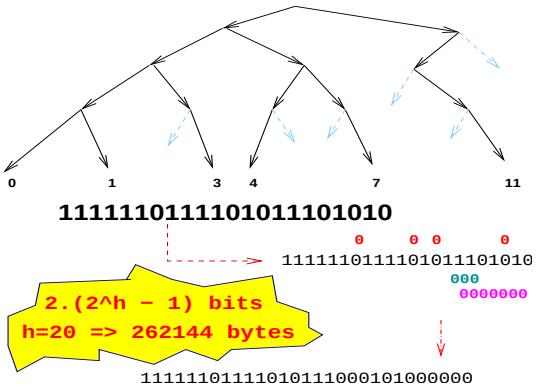


Figure 5: Vector of bits construction and storage on disk

In order to parallelize the construction in memory of the bit vector, we can apply a divide and conquer strategy that follows the previous inductive definition: at each node, we start two new threads to compute the result for the left and right children. Once the vector has been built in memory, we flush it to the disk. We note that since the tree is not necessary balanced, the time duration of computing the bit representation for the left child and for the right child is not necessary equals. So we have here a supplementary problem about how to balance the work evenly.

The construction of the tree starting from the disk representation can also be multithreaded in a similar way. Only one sequential reading of the representation on disk serves to allocate the minimal number of memory to store the entire tree nodes.

Operation on Radix Trees can also be multithreaded. Let us consider the AND operation that is useful to determine the common line indexes between two tables for instance.

First of all we start a thread to 'make' the AND operation on the two root-trees of the hierarchy. We denotes by A, B the root-trees. This produces a new tree that we call C. Second we consider only leaves in C and we determine the pairs (p, q) such that p is a leaf of A, q a leaf of B and p, q correspond to a leaf r in C denoting the same integer. Then we start new thread on each (p, q) pairs in order to accomplish the AND operation in parallel.

The problems we have presented in this subsection motivate and give a flavor of the basic data structures and algorithms embedded in the Zêta-Data project. We have also shown sources for parallelism at the level of construction of the basic underlying data structures. It is also remarkable to note that the SQL service does not employ btree-like (Btree+, Rtree...) data structures but only radix trees stored on disks. For that reason, we consider that our project is innovative.

3.5 Multithreading coarse grain tasks of the engine

In the previous section, we have seen that it is possible to multithread our program according to a fine grain strategy, we mean that we can multithread the building of an internal data structure. In this section, we introduce a coarse grain strategy for multithreading our sequential program. This work corresponds to the present status (Apr 2004) of our project.

3.5.1 Overall architecture of the SQL service

The architecture of the SQL service is depicted in Figure 6. We have 4 modules. The Engine module is responsible for answering to a query corresponding to the requete.pol input file. The Engine module

needs also the tables (`.dat` files), the indexes and offsets [4] files. The Engine module writes the answer to the `reponse.txt` file.

The roles of "Create Indexes" and "Create Decalages" modules are implicit. At least, the "Insert, Delete, Modify" module allow the creation, the delete or modify of ASCII tables, indexes and offset files.

In the remainder of the section, we introduce the work done for the parallelization of the engine module.

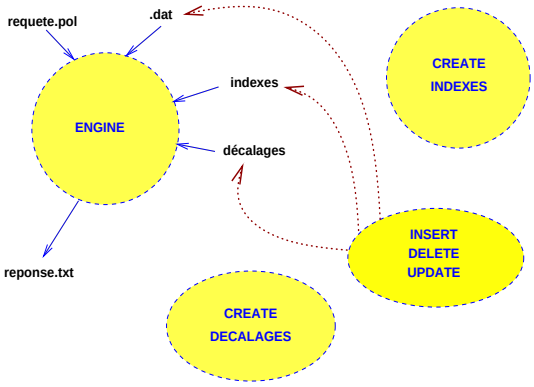


Figure 6: Overall architecture of the SQL service

3.5.2 Parallelized tasks for the engine module

Two main tasks have been multithreaded. The first one corresponds to the computation of line indexes that answer to a WHERE clause. In fact, we activate a thread for every packet of $2^{20} = 1048576$ of lines in the input table. For instance, the exploration of the LINEITEM table in the TPC will generate 6 threads because the LINEITEM table has 6001215 lines ($6 * 1048576 > 600121 > 5 * 104857$) for a total size of 759861366MB.

The second task that we have multithreaded is the computation of the final result; more precisely it is a matter of evaluation of the partial results corresponding to AND, OR...clauses. We have also multithreaded by packets of 1M of lines because the previous steps returns as much intermediate results as the number of packets of size 1M. For the LINEITEM table, the number of active threads is also 7.

Then the partial results are combined and we can write the final result into the `reponse.txt` file.

4 Tools and experimental results

The sequential results are presented in Figure 7 and for the TPC benchmark. The results are obtained for an Athlon 1800 processor with 1GB of memory. The first and complete sequential implementation provides performance improved by a factor at least 8 (see column DBA) compared with commercial products (see columns DBM1 to DBM3) and for queries including a query having a lot of answers (query Q1 - found 90% of the time), a query of a very combinatoric type (Q6), a correlated sub-query (Q17), a query with a lot of computations on the data matching the query (Q19), and insertion or deletion of 10% of the database (see the RF1 and RF2 lines). A cartesian product of the two tables (one of 6 million lines - the "LINEITEM" table in the TPC - and the other one of 1.5 million lines - namely the "ORDERS" table in the TPC) was the last query investigated. Note that in this last case, no commercial product can respond in less than one day!

[4] It is a meta representation for dates not introduced here for the sake of simplicity

Request	DBA	DBM1	DBM2	DBM3
Q1	8s	47s	370s	33s
Q6	2s	24s	22s	24s
Q17	3s	8s	10s	8s
Q19	3s	19s	24s	25s
RF1 (Insert)	4s	231s	96s	53s
RF2 (Delete)	5s	121s	85s	42s
Cartesian Product	7s	non Op.	Non Op.	Non Op.

Figure 7: Performance of the sequential implementation for the TPC-H benchmark

Two multithreaded releases have also been implemented. The first one is coded with the Pthread library, the other one with PM2[5] (*Parallel Multithreaded Machine*).

PM2 [NM95b, Nam01] is developed at LABRI (*Laboratoire Bordelais de Recherche en Informatique*), a research laboratory located at Bordeaux, France, jointly supported by the INRIA and CNRS institutions. PM2 was originally designed at LIFL, University of Lille, France.

PM2 is a distributed multithreaded programming environment designed to efficiently support irregular parallel applications on distributed architectures of SMP nodes.

The thread management subsystem of PM2 is called Marcel [NM95a], the communication management subsystem is called Madeleine but we do not use it in our implementation yet. Marcel is designed to manage several hundreds of threads on each available physical processor. The PM2 programming interface provides functionalities for the management of this high degree of parallelism and for dynamic load balancing. Interesting features of PM2 include its priority driven scheduling policy, its thread migration mechanism and its ability to support various load balancing policies. PM2 has been designed to provide threads as light as possible: the switching time is well under the micro-second. Marcel is completely written in portable C, but a dozen of lines of assembly code, which makes it portable across most processors and flavors of Unix systems, including Solaris and Linux. However, significant improvements have been made for Linux by introducing specific support into the operating system.

Our PM2 code is currently running on a single processor PC (linux Mandrake 9.1) motherboard. We have noticed similar execution times between our sequential implementation and our PM2 implementation with an overhead of few percent (under 5%) in favor of the sequential execution time. We have also made the same observations with our Pthread implementation. Theses results show that our application can not be used to determine the best thread library to use.

We have noticed that about 80% of the execution is passed in the query evaluation that is to say in the second portion of code that we have multithreaded.

We are currently porting PM2 on the dual-processor Sun Fire V20z server based on AMD Opteron at 2GHz. With a single architecture, the Sun Fire V20z server supports both 32-bit and 64-bit computing, allowing the user to support any existing x86 infrastructure.

We have experimented with Pthreads and with a Sun Fire V20Z with two 36 GB Ultra320 SCSI disk drives, a total of 4GB of memory and a ReadHat 9 (with `-m 32` flag because our 64 bit code is not yet enough stable). We have noticed execution times higher by a factor of 5 for the sequential executions of

[5] See `http://www.pm2.org`

the TPC and comparing with the Athlon results.

The query which involves the greatest number of threads during its evaluation is Q6 because it considers the LINEITEM table which is the largest table (760MB and about 6 million lines). Six threads are launched for this query because we deal with chunks of one million lines. We have noticed a performance increase of 40% in favor of the multithreaded code against the sequential code but this number is to be used carefully.

Indeed, the time of Q6 execution is now under 0.2 seconds on Sun V20z and we could not precisely ascertain whether the processes were forked on distinct nodes or not. We also need a tool and much greater problem sizes in order to appreciate the 'quality' of linux task scheduler and its impact on performance.

5 Conclusions

In this paper we have shown how to introduce threads management in the Zêta-Data project which is part of the Grid Explorer project. The current development of our SQL service focus on multithreading tasks at the procedure level. Promising experimental results have been demonstrated presented.

Future implementations will consider a fine grain approach in order to multithread the construction of radix trees. The particularity of our SQL service is that it does not use Btrees like data structures but radix trees stored on disks and a flat expansion of tables.

The next step in the project will be to consider that ASCII tables and indexes are distributed across a Grid Platform and we should offer the service despite some overloaded processors, unreliable links between sites that obtain dynamically their IP addresses and some sites may disconnect at any time.

Acknowledgment

We would like to thank Pascal Bonnau, Sun Microsystems France S.A, for his effort in making possible the availability of a Sun V20z server.

References

[CKFJ04] Christophe Cérin, Michel Koskas, Hazem Fkaier, and Mohamed Jemni. Sequential in-core sorting performance for a sql data service and for parallel sorting on heterogeneous clusters, revision version for special issue of future generation computer systems (published by elsevier) on system performance analysis and evaluation, 2004.

[KK03] Michel Koskas and Elie Koskas. A hierarchical database management algorithm, 2003.

[Nam01] Raymond Namyst. *Contribution à la conception de supports exécutifs multithreads performants.* Habilitation à diriger des recherches, Université Claude Bernard de Lyon, pour des travaux effectués à l'école normale supérieure de Lyon, December 2001.

[NM95a] Raymond Namyst and Jean-François Méhaut. *Marcel : Une bibliothèque de processus légers.* LIFL, Univ. Sciences et Techn. Lille, 1995.

[NM95b] Raymond Namyst and Jean-François Méhaut. PM2: Parallel multithreaded machine. a multithreaded environment on top of PVM. In *Proc. 2nd Euro PVM Users' Group Meeting*, pages 179–184, Lyon, September 1995.