Mining Traces of Large Scale Systems

Christophe Cérin
Université de Paris XIII
LIPN, UMR CNRS 7030
F-93430 Villetaneuse - France
cerin@lipn.univ-paris13.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

***Abstract*— Large scale distributed computing infrastructure captures the use of high number of nodes, poor communication performance and continously varying resources that are not available at any time. In this paper, we focus on the different tools available for mining traces of the activities of such aforementioned architecture. In this paper we propose new techniques for fast management of a frequent itemset mining parallel algorithm. We present statistical results about the activity of more that one hundred PCs connected to the web.**

Keywords: Parallel algorithms, global computing platforms, meta-data, data mining application, high performance and distributed databases, trace analysis, data management, resource management.

I. Introduction

Frequent itemset mining (FIM) consists in discovering patterns that appear frequently. In this paper the itemsets are informations about the activities such as the CPU/MEMORY loads, the number of IP packets sent or received from/to a dedicated node, date of the measure... of a set of PCs in a research laboratory. The ultimate goal for that application is to extract information to be pass to the scheduler in order to run jobs with a reasonable knowledge of the "future state" of the global platform.

FIM algorithms are often used in search for other types of patterns (like sequences, rooted trees, boolean formulas, graphs). More than one hundred FIM algorithms were proposed in the literature, the majority claiming to be the most efficient. In any case, it is difficult to appreciate the experimental methodology. For instance it is difficult to have answers to the following questions: what is the part of the execution done in/out-of-core? What is the execution time for generating the 1-itemset (it corresponds in general to a full reading of the database) and the execution time for generating the $k > 1$ itemsets?

Three algorithms play a central role due to their efficiency and the fact that many algorithms are modifications or combinations of these basic methods. These algorithms are APRIORI [1], ECLAT [2] and FP-growth [3].

In this paper we introduce challenges, opportunities and some technical solutions that we believe to be important for mining the activities of large scale systems. The discussion is conducted with our parallel algorithm for frequent itemset generation [4] in mind. Besides the algorithmic and data-structure issues there is a third factor that quite influences the effectiveness of the different approaches found in the literature: the programming technique.

According to this remark, the organization of the paper is the following. In Section II, we introduce the challenges and those we are concerned about in the paper. Section III is about the data structure we use and section IV about the parallel algorithm. We extract its main properties and we show its potential for mining large scale systems. Section V is about the programming techniques and it provides with experimental results. Section VI concludes the paper.

II. Building a FIM algorithm for large scale systems

The main properties that we require for large scale systems are:

- *Scalability:* the system must scale to hundreds of thousands nodes;
- *Heterogeneity* of nodes across hardware, OS and basic software;
- *Availability:* the owner of a computing resource must be able to decline a policy which will limits the contribution of the resource (the resource will be de-connected in a near future);

- *Fault tolerance:* the architecture must tolerate frequent faults while maintaining performance;
- *Security:* all participating computers should be protected against malicious or erroneous behaviors;
- *Dynamicity:* the system must accommodate to varying configuration; an event may happen at any time;
- *Usability:* the system should be easy to deploy and to use.

In this paper we discuss only the advantage of our algorithm [4] in terms of scalability, dynamicity, fault tolerance and performance at the large.

A. Problem description

One challenge that we address in the paper is how to represent and how to mine data sets representing the activities of large scale systems connected to each other. We may assume that a process on one node has been setup in order to collect traces of the activity of all the machines connected to the infrastructure. The missions assigned to the process/node are:

- Collect traces and dispatch them on the clients or on a dedicated node;
- Discover "frequent episodes". For instance, determine which machines had CPU load greater than 40% during the past hour;
- Build heuristics about the future state of the global system from the frequent episodes in order to pass them to the job scheduler(s).

The collect phase necessarily involves disks. Our experimental data set is from more than one hundred machines sampled every 15 minute during approximately half a day. The amount of generated data (for 15 collected events) is about 50Mb in size (uncompressed). Such volume is enough to exhibit the main performance of tools that process the data. We estimate a volume of 11GB if the sampling occurs every minutes and during 10 hours per day.

Very little work as been done on the different ways of mining data located on PC and participating to a global computation. For instance, the Grid Forum working group DAMED (Discovery and Monitoring Event Description) has produced a pre-classification of pertinent events related to machines. They also evaluates the use of data models (for example relational, hierarchical, etc.). They do not yet consider the problem of mining data and the relation between the representation and the algorithmic of mining.

We think that the way we represent data will potentially enforce the efficiency of mining algorithms. Three categories of techniques for mining are under concern in our project in order to confront arguments in favor or against one technique.

B. Tools for mining and challenges

Mining can be done in many ways and with many tools. In the first category of tools we have Unix (grep, perl) tools. We are currently experimenting with `nrgrep`. It is a fast implementation of `grep` with more properties. For instance, you can do approximate string matching: in the pattern description, you allow some misses or conversely we allow the insertion of letters. None of these tools is able to treat the whole mining purpose. However, we used them in this paper, because we were only interested in statistics regarding traces.

In the second category we have the mining techniques based either on a threshold such as MINEPI, WINEPI [5] or based on the expression of constraints directly on the input sequence of data. SPADE [6] which are two examples related to the last technique. The current limitations of such methods are they do not take into account that some part of the information could disappear partly or definitely because a machine become disconnected. We do not know any research work on the best way to add duplication or redundancy concepts in mining algorithms. It is a great challenge which could potentially solve the problems.

The third category of algorithms that can potentially be used in the process of mining is based on the utilization of generic database tools.

Mining from large databases imposes new challenges and opportunities [7], [8] to database technology itself. There is a need for new query languages and query processing methods that will address the requirements posed by database mining. The main challenge is to develop data mining algorithms that will present a tighter coupling with 'traditional' database software.

The main challenge is to store data and also to discover association rules within a single tool. It assumes the integration of an efficient facility for

1) the selective generation of patterns matching the query 2) the management of previously generated results (i.e. the handling of already mined patterns).

Regarding the former case, techniques for pushing constraints down to the mining process have been proposed, mainly for association rules. In the application presented in the paper, we focus on the *pattern query*, that is, finding all sequential patterns that contain an ordered set of user-defined elements. It is a matter of indexing sequential patterns.

To explain our current work, let us consider the following problem that correspond to a smaller problem size instance than the original and practical one.

We reduce the number of events in our table to three: the IP activity of the machine, the name of the event (N) and the timestamp of the event (TS). A user wants to know the number of times that the table contains the ordered sequence $10 \rightarrow 20 \rightarrow 30$. The meaning of such a request is that we want to count, for instance for the CPU load, the number of times that we observe (the ordered sequence) a load of 10%, then a load of 20% than a load of 30%. A SQL query, implementing the above pattern looks like:

```
select IP from R a, R b, R c
where
     a.IP = b.IP
 and b.IP = c.IP
 and a.TS < b.TS
 and b.TS < c.TS
 and a.N = 'Load'
 and b.N = 'Load'
 and c.N = 'Load'
```

Such query can serve to designate a set of machines that a user estimates as machines with a 'bad behavior' because the load 'frequently' becomes 'high'.

Since SQL language does not contain a sequence search statement, the above SQL query is implemented with multiple joins or multiple nested subqueries. This may presents large query response time. Moreover, sequential scan may require considerable I/O operations that slowdown the program.

III. The data structure

Before introducing our parallel algorithm, we introduce the data-structures it uses. The parallel algorithm makes elementary operations on such data-structures in order to compute the frequent episodes.

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: anyway 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) couple corresponding to two tables.

A. An example

Let us consider the following data basis, composed of three tables, namely Accident, Customer and Insurance tables. The Accident table has seven lines as one can see on Figure 1. When considering Accident table, it's 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 2 and the Insurance table has two lines: see Figure 3

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

Now, consider the Insurance table. It has several columns, and each of them is treated separately: for

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

Fig. 1. The root-table: the Accident table.

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

Fig. 2. An intermediate table: the Customer table.

each of these columns, one builds its thesaurus and for each word of the thesaurus we build the set of line indexes it occurs at.

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 data basis by computing the thesaurus of each column of each table and its sub-tables and the line indexes each word occurs at.

Now, let us return to the building of the indexes.

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

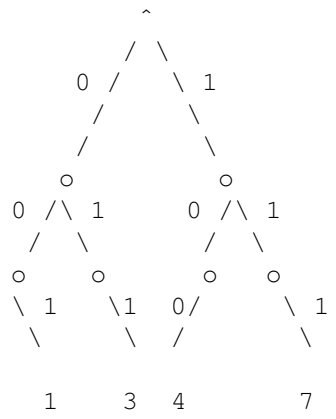
Fig. 3. A leaf table: the Insurance table Compagny.

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 of 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 of items with only 5 different values. Special techniques have been devised in [9] for this purpose but they are out of the scope of this paper. Please contact the authors for complementary material.

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). It is well known that 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 $0 \dots b-1$). Consider the Accident Table depicted on Figure 1 and the "Kind of Cont" column.

The thesaurus of the column is $\{House, Car, Familly\}$. 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 to a query with an AND clause, for instance:

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

IV. The algorithmic part

The problem of association rule discovery can be formalized [10] as follows. Let $\mathcal{I} = \{i_1, \cdots, i_m\}$ be a set of m distinct *items*. A transaction is any subset of $\mathcal{I}$ and each transaction $\mathcal{T}$ in a database $\mathcal{D}$ of transactions has a unique identifier. A transaction is a p-uple $< TID, i_1, \cdots, i_k >$ and we call $i_1, \cdots, i_k$ an *itemset* or a *k-itemset*.

An *itemset* is said to have a support of s if $s\%$ of the transactions in $\mathcal{D}$ contains the itemset. An association rule is an expression of the form $A \Rightarrow B$ where $A, B \subset \mathcal{I}$ and $A \bigcap B = \emptyset$. The *confidence* of the association rule is simply the conditional probability that a transaction contains B, knowing that it contains A. It is computed as $\text{support}(A \bigcap B)/\text{support}(A)$.

Given m items, there are potentially 2^m itemsets whose support is above a given support. Enumerating all itemsets is thus not realistic. However, for practical cases, only a small fraction of the whole space of itemsets is above a given support requiring special attention to reduce memory and I/O overheads.

The "Apriori" sequential algorithm forms the core of many variants of association rules discovery algorithms. It uses the fact that a subset of frequent itemset is also frequent, then only candidates found "previously" are used to generate a new candidate set. This algorithm has three main steps, iterated while new candidates are generated:

- Construction of the set of new candidates;
- Support evaluation for each new candidate;
- Pruning of candidates that have not a sufficient support regarding to a minimum support arbitrarily chosen.

The complete sequential algorithm is as follows:

The Apriori algorithm

$L_1 = \{$ frequent 1-itemset$\}$;
for $(k = 2;\ L_{k-1} \neq \varnothing;\ k++)$
  $C_k =$ Set of new Candidates;
  for all transaction $t \in D$
    for all k-subsets s of t
      if $(s \in C_k)$ s.count++;
  $L_k = \{c \in C_k \mid c.\text{count} \geq \text{minimum support}\}$;
Set of all frequent itemsets $= \bigcup_k L_k$;

Note that in this algorithm, the whole database is read at each iteration step (see the **`for`** `all transaction` $t \in D$ instruction above). Consequently, the performance could not be high with such framework.

In [4] we have introduced new techniques for association rules discovering. We have revisited the Apriori algorithm that serves as the main conceptual block for such purpose in showing how to store and generate candidates by the mean of Radix Trees.

Surprisingly, the new (parallel) algorithm is based on the Apriori framework but it uses only operations on Radix Trees, namely union, intersect operations for candidate generation and in support computation. The parallel algorithm is stated as follows:

Algorithm executed on each Proc. $0 \leq i \leq p$**.**

/ Initially, each processor has locally n/p lines of the transaction database where n is the total number of lines and p is the processor number.*/*

1- In parallel for each processor:
Scanning of the local database for construction of 1-itemset tree.

2- In parallel for each processor:
do
  Broadcast supports.
  /* This part can be unsynchronized */
  /* to perform overlapping */
  Wait for all supports from others.
  Perform the sum reductions.
  Elimination of un-sufficient itemsets support.
  $L_k =$ rest of C_k
  Construction of new candidates sets C_{k+1}.
while $(C_{k+1} \neq \varnothing)$
3- frequent itemsets $= \bigcup L_k$

From a "large scale point of view", the main properties of the algorithm is:

- The local database is read once. This step serves in building the 1-itemset, that is to say the radix-trees coding "where" each item appears. The $k > 1$ itemsets are generated

by intersections, locally on each node. If we assume that a new itemset can arrive at anytime (a new measure in our application), we should minimize its insertion time. In our case, the cost of inserting one item is a constant time, independent of the number of data since it is based on the tree high which s a constant (for instance 20 if we are working with tables with 2^{20} lines). This property is important in the case of the aforementioned property of dynamicity of large scale systems.

- When we exchange information about nodes, only the supports (integers) are exchanged. There is $(p-1)^2$ messages during this steps and the length of each message is proportional to the number of frequent itemsets that are generated (k) multiply by the size of an integer. Thus the volume of information exchanged in any step of the parallel algorithm is exactly $(p-1)^2 \times k \times sizeof(int)$. We note that it is independent of n the number of data in the database. We may assume that in practical cases, this volume is low. The consequence is that in the case of faults, the checkpoint will contain not too much information. For instance, if we use MPICH-V (a fault-tolerant MPI available on `http://www.lri.fr/~bouteill/MPICH-V/`), the NAS Benchmark BT B on 25 nodes (32MB per process image size) leads to the average time of 68s to perform checkpoint with MPICH-V. The average time to recover from failure with MPICH-CL is 65.8s. The application is much more communication intensive that our frequent itemset algorithm. We are optimist to accomplish a checkpoint in less than 1s on 25 nodes.

However, the number of itemsets generated varies from one iteration to another one. In the context of heterogeneous computing (the candidate and frequent itemset generations are computed on different processors and with different communication bandwidths) it is more difficult to estimate the time cost of these two steps, hence potential unbalanced work. Techniques to control the load balancing, such as the technique used in [11] in the case of sorting and for a one-communication-step algorithm can not apply. Thus, the problem of controlling load balancing is challenging both in theoretical and programming terms.

V. The programming technique part

Let us now comment our implementation choices in the case of our sequential frequent episode prototype. Radix Trees can be implemented with pointers (for the left and right children) when they are loaded into the RAM. We know that pointers do not preserve spacial locality (the next item to be used is "closed" in memory to the current item) and it is not also suited for temporal locality (the current item will be re-used in a near future).

To check this fact for our purpose, we have implemented tree operations (union, intersect) with the STL C++ library and lists and with pointers. We have obtained better experimental results for pointers than for lists (implemented under the STL C++ framework).

But the time completion for union or intersect operation is not good enough for large scale computation. For instance, 600 intersection operations on trees containing 150000 elements each last 58.39 seconds on a Sun bi-opteron v20z system. These 600 operations involve 90M of items.

We have decided to shift to bitset abstract data type in order to implement "the line where an item occurs" concept. Remind that Radix Trees have been introduced to store sets of integers. With a bitset, we set to 1 the k-*th* bit if integer k is a member of the set and we set it to 0 otherwise. The STL C++ library offers an interface to bitsets but after some tests with the library and under `g++` release `3.4.1` we have decided to re-implement it, partially in assembler code.

The motivation is to use MMX, SSE-2 or Altivec technologies for 32 bits processors. For such technologies, the processor can address 128 bits registers and we can use them to implement union operation (i.e. "or" operation on two bitsets), intersect operation (i.e. "and" operation on two bitsets. The STL C++ library under `g++` does not use such technologies.

We have also introduced (by hand) prefetching memory instructions. Such optimizations are essential to fully exploit 128 bits registers and to hide memory latencies.

We obtain a gain of at least 30% for our MMX/SSE and Altivec implementations against the STL C++ codes. For instance, the cost of 5000 ”and” operations on two bitsets of size 1048576 bytes (representing two sets of 8388608 elements) is 7.75 second on a Duron at 1.9Ghz. It is a very good result comparing to our tree implementation based on pointers (see above). The effort in coding the new bitset interface is not too important for a great result.

A. GCC 4.1

The GNU Compiler Collection (GCC) is the leading compiler suite for the free and open source software. The large set of target architectures and the standard compliant front ends makes GCC the “de facto” compiler for portable developments. However, until recently, GCC was not an option for high performance computing: it had no infrastructure for data access restructuring nor for automatic parallelization nor for using specialized hardware.

GCC 4.x (experimental) addresses the challenge of high performance computing. Modern compilers implement some of the sophisticated optimizations introduced for supercomputing applications. They provide performance models and transformations to improve fine-grain parallelism and to take into account the memory hierarchy (prefetching). Most of these optimizations are loop-oriented and assume a high-level code representation with rich control and data structures: do loops with regular control, constant bounds and strides, typed arrays with linear subscripts. For instance, the GIMPLE representation was proposed by Sebastian Pop, and Diego Novillo from RedHat, for minimizing the effort in the development and the maintenance of new analyzes and code transformations.

Another improvement of GCC 4.x is the possibility to produce SSE2 and Altivec codes in order to use 128 bits registers. Two new compiling options were introduced: `-ftree-vectorize` and `-ftree-loop-linear`. We have built a test based on a loop to observe if GCC 4.1was able to generate assembler code with prefetching instructions and SSE2 or Altivec instructions.

The test numbered 1 was:

```
for(i=0;i<2048;i++)
   b[i] = b[i] & a[i];
```

The test numbered 2 was:

```
for(i=0;i<2048;i++)
   c[i] = b[i] & a[i];
```

The compiler options were: `-S -O4 -fprefetch-loop-arrays -ftree-vectorize -msse2 -ftree-loop-linear`. In the case of the first example, GCC 4.1.0 20050320 (experimental) was able to generate prefetching instructions but no “pand” operation (the SSE2 operation to do a 128 bits and operation). In the case of the second example, GCC 4.1.0 was able to generate a “pand” instruction but not prefetching instructions. One explanation may consider the need to unroll the loop in order to align the data with the 128 bit size. GCC has probably no information on the final loop index, so it fails to unroll, so it fails to put a pand” instruction.

So we decided to continue to work with our own library that implements prefetching and “pand” instructions. We have implemented a sequential APRIORI based algorithm using bitsets. This code will serve in the future as the sequential main brick of the parallel algorithm. We introduce now some experimental results.

B. Trace analysis and statistics

The trace that we have explored corresponds to a set of 110 stand alone PCs under Windows in a laboratory of researchers, engineers and administration people. The trace records 11 events every 15 min during the day and for a period of 15 days. The trace represents about 50Mb of uncompressed data in size. The name of the table is BigTable. We estimate that if we sample every minute, the file size will be about 11Gb for 15 days and we are optimist.

In order to get statistics we have developed a set of programs written in `shell, python` and we also use `nrgrep-!.1.1` for the matching of patterns. Our experiments are accomplished on an Athlon(tm) XP 1800+ processor equipped with 512MB of DDR PC2100 an IDE 80GB hard-disk.

1) Some Statistics: Figure III shows some statistics about the CPU load. We note that for 84% of the experiments the load is below 10%: the CPU are under-utilized. We also note a relatively frequent occurrence of a CPU load between 90-100%. This may be caused by the measure itself if the process

that make the measure is in the active state during the measure.

Figure IV shows statistics about the number of IP datagrams that has been sent during a period of 15 minutes. Since both the `x` and `y` scale are logarithmic and since the curse drawn when we consider the middles of 'plateau' is a line, we conclude that we have a Zipf law. This measure can serve as a model of the behavior of the network in a "large scale" distributed system. We do not know if others researchers have observed the same phenomenon.

We note also on Figure V that the number of alive processes is under 30, half the time. Figure VI shows the number of observed disk transfers. Surprisingly, we found a "pattern" repeated 3 times. We do not know any law corresponding to this phenomenon. We cannot explain it.

2) Experiments with scripting language and shell tools: The experiments that we have conducted to mine our trace consider the problem of finding the number of occurrences of the 'CPU load equals to 10%' event.

The first one corresponds to Shell programs that use either `nrgrep` or Unix `cut` commands. The execution time is greater than 45 seconds to find all the occurrences.

The second one corresponds to the following `Python 2.3a1` code that uses the pattern matching facilities available with `Re` module. The execution time of the code on our Athlon-1800+ processor is about 8.6 seconds to find the 3214 occurrences.

The third one corresponds a `nrgrep` command line. The execution time for the command is less than 1 second to discover the 3214 matching records. The problem with `nrgrep` is that we have a limited power for the expression of queries comparing to SQL query languages.

The fourth one corresponds to a `Perl` program using pre-compiled patterns to speed-up the execution. The execution time is about 1 second to find the solutions. All these results give the reader some indications of the time cost of an elementary search. Apriori algorithms require many of such elementary search.

3) Performance of our Apriori data mining algorithm: A Frequent Episode sequential algorithm based on [4] and on bitsets has also been implemented. We have chosen 40 items in the BigTable table of 685673 lines.

Our implementation uses `libpcre`[1] for matching patterns. The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5. PCRE has its own native API, as well as a set of wrapper functions that correspond to the POSIX regular expression API.

Since we have 685673 lines in the flat input table, we have set our bitset sizes with 131072 bytes. The total memory size is thus about 40MB, that is to say closed to the table size (47MB). So our implementation is an in-core one.

Under GCC 3.3.4 (pre 3.3.5 20040809) the execution time on a Duron 1.8Mhz for counting the occurrences of these 40 items and including the setting of bitsets with the line numbers where they appear is 14.58 seconds.

This means that we need 14.58 seconds for generating *1-itemsets* including one pass over the input file. With a support equals to 68567, which represents 10% of the number of lines in the input table, we get 13 *1-itemsets*.

The number of *2-itemsets* is 17 (we compute it according to the same support) among potential 78 2-combinaisons.

The number of *3-itemsets* is 6 (we compute it according to the same support) and these is no more frequent itemsets. One of the *3-itemsets* corresponds to a CPU load greater than 90% and the number of running processes is greater than 90 and the available memory is greater than 300MB.

The total number of lines (elements) involved in intersect operations is 89342969. The execution time from the end of the *1-itemset* production to the end of the program is less than 1 second (0.91s). This time includes the generation of *2-itemset*, *3-itemset* and *4-itemset*.

It is a very good result comparing to the previous incomplete tests based on scripting languages. The number of elements involved in intersect operations is quite impressive and the experiment confirms that the data structure choice is a good one.

Under GCC 4.1 and the following

[1] `http://www.pcre.org/`

option flags `-O4 -fomit-frame-pointer -fprefetch-loop-arrays -ftree-vectorize -msse2 -ftree-loop-linear` we got the same execution time. In fact, this test cannot allow us to distinguish the performance of the bitset library alone because it uses the same ASM bitset library.

If we recode the `and` operation as follows:

```
void pandCopy(unsigned char *s,
 unsigned char *t, unsigned char *d,
 long int nb_bit)
{
int count,i;
int *cs=(int *)s, *ct=(int *)t,
    *cd=(int *)d;
count = (int)(nb_bit / 32);
for(i=0;i<count;i++){
  cd[i] = ct[i] & cs[i];
}}
```

and we let GCC 4.1 to do the optimization, we get the same execution time. It is promising since we have checked that GCC 4.1 have not produced "pand" instruction (but prefetcht0 instructions) in our code. No loop unrolling has been made. In the future, we will certainly use GCC 4.x stable release and portable C code instead of our current C and assembler codes. But it is too early at present time to get performance with GCC 4.x.

C. Impact of the statistical results on placement

The final aim of the experimental study is to discover trends in the behavior of PCs connected on a large scale distributed system. The aim is to place tasks. One of the frequent *3-itemset* that we have generated according to our algorithm is: "CPU load $> 90\%$ and Memory Available '$> 300K$ and number of processes > 90". It shows that a high CPU load is frequent: recommendation to place tasks is difficult with this information. We have also another frequent *3-itemset* saying that "CPU load $< 10\%$ and Memory Available '$< 20K$ and number of processes < 40".

It is a little bit surprising and it is due to the choice of the support. More discriminant method should accompany the frequent episode algorithm. For instance, in our result we have 5/6 frequent episodes saying that the load is $< 10\%$ and only 1 frequent episode saying that the load is $> 90'$. We have also 4/6 frequent episodes saying that the memory available is between 20 and 40K. The others occurrences of frequent items in the frequent episode is less or equal to 2.

Moreover, if the application is not a critical one we could spend time on collecting burst events to examine in deep this phenomenon. In this case, the key challenge is to master the disk space to store and/or to factorize massive information with common properties: if the burst occurs for the CPU usage, others informations may not vary a lot.

VI. Conclusion

In this paper we have presented how we are currently implementing in the "ACI Masse de donnée grid project"[2] a middleware in charge of controlling common data structures used in order to store activities of participant PCs in a large scale system. Different techniques have been explored for mining the trace of the activity and in order to get performance.

Our Apriori algorithm is based on Radix Tree and/or Bitset data structures. Such data structures have been proved efficient according to a pointer based implementation but bitsets are more promising. We are currently developing a multithreaded version of our bitset library for clusters of SMP. The multithreaded version of the intersect operation of two Radix Trees, for instance, introduces problems with balancing the work among threads. We are investigating such issues.

Concerning the Apriori algorithm, we will implement an out-of-core version in order to deal with large tables and before implementing the parallel version depicted in [4]. Our objective is to capture tables until 2^{44} lines. A compromise between space and efficiency for the Bitset data structures is currently under concern.

References

[1] R. Srikant and R. Agrawal, "Fast algorithms for mining association rules," in *The International Conference on Very Large Databases (VLDB)*, 1994, pp. 487–499.

[2] O. Zaki, Parthasarathy and Li, "New algorithms for fast discovery of association rules," in *In D. Heckerman, H. Mannila, D. Pregibon, R. Uthurusamy and Park editors, Proceedings of the 3rd International Conference on Knowledge Discovery and Data Miing - AAAI Press*, 1997.

[2]**Acknowledgements:** We also address a special thank to Oleg Lodygensky from LAL loboratory in Orsay - France for his tool that inspect and collect traces.

[3] P. Han and Yin, "Mining frequent patterns without candidate generation," in *In proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000.

[4] G. Cérin, Koskas and Le-Mahec, "Efficient data-structures and parallel algorithms for association rules discover," in *3rd International Conference on Parallel Computing Systems (PCS'04), Colima, Mexico*, September 2004.

[5] H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 259–289, 1997. [Online]. Available: citeseer.nj.nec.com/mannila97discovery.html

[6] M. J. Zaki, "SPADE: An efficient algorithm for mining frequent sequences," *Machine Learning*, vol. 42, no. 1/2, pp. 31–60, 2001. [Online]. Available: citeseer.nj.nec.com/zaki00spade.html

[7] R. Agrawal and K. Shim, "Developing tightly coupled data mining applications on relational database systems," in *International Conference on Knowledge Discovery in Databases and Data Mining (KDD'96)*, 1996.

[8] T. Imieliski and H. Mannila, "A database perspective on knowledge discovery," in *Communication of the ACM 39 (11)*, 1996.

[9] C. Cérin, M. Koskas, H. Fkaier, and M. 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.

[10] M. J. Zaki, "Parallel and distributed association mining: A survey," *IEEE Concurrency*, vol. Vol. 7, No. 4, October-December 1999, pp. 14-25.

[11] M. J. Christophe Cérin, Michel Koskas and H. Fkaier, "Improving parallel execution time of sorting on heterogeneous clusters," in *Proc. 16th International Symposium on Computer Architecture and High Performance Computing (SBAC'04), Foz-do-Iguazu, Brazil*, 2004.

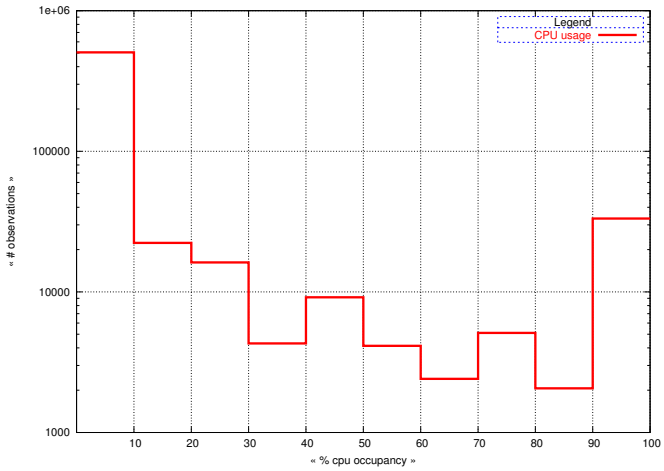


Fig 3: CPU load statistics

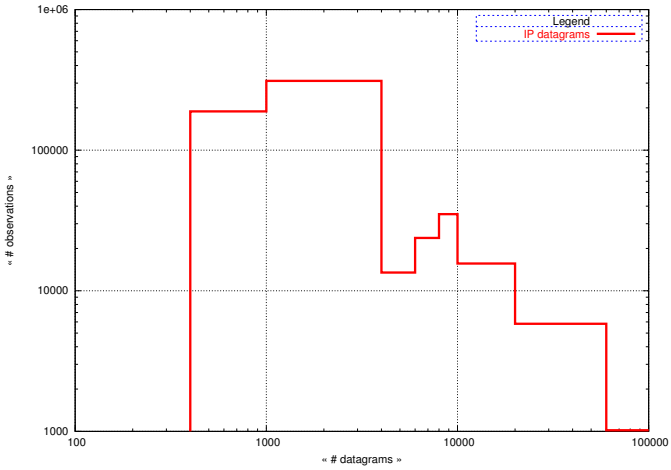


Fig 4: IP datagram statistics

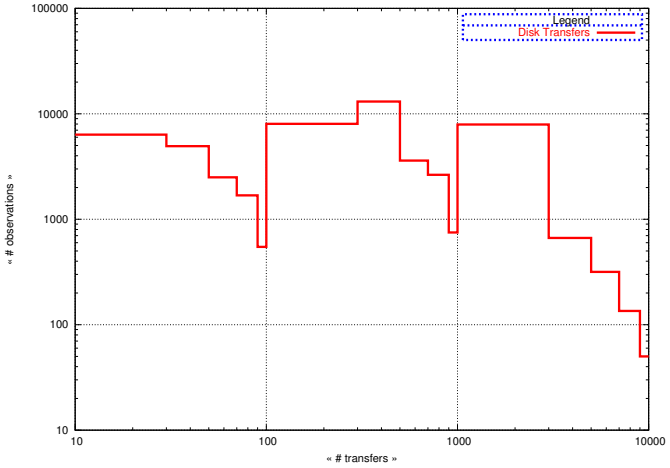


Fig 6: Disk transfers statistics

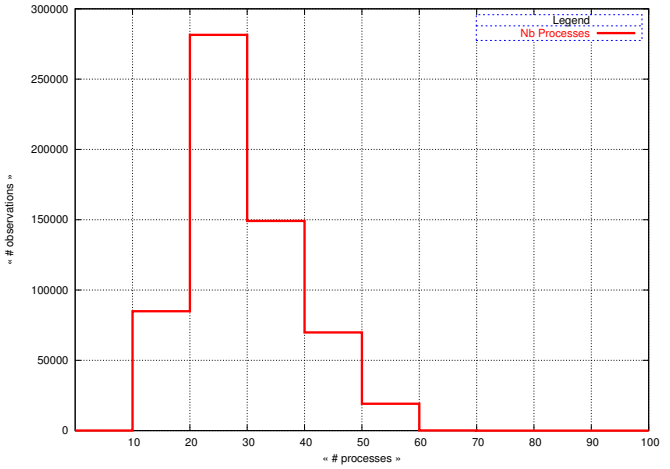


Fig 5: Processes statistics