Efficient Data-Structures and Parallel Algorithms for Association Rules Discovery

Christophe Cérin
Jean-Sébatien Gay
Gaël Le Mahec
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

Abstract

Discovering patterns or frequent episodes in transactions is an important problem in data-mining for the purpose of infering deductive rules from them. Because of the huge size of the data to deal with, parallel algorithms have been designed for reducing both the execution time and the number of repeated passes over the database in order to reduce, as much as possible, I/O overheads. In this paper, we introduce new approaches for the implementation of two basic algorithms for association rules discovery (namely Apriori and Eclat). Our approaches combine efficient data structures to code different key information (line indexes, candidates) and we exhibit how to introduce parallelism for processing such data-structures.

Keywords: *Datamining, Association rules discovery, Radix Trees and bit vectors, Apriori, Eclat and Count Distribution algorithms.*

1. Introduction

The process of automatic information inferencing is commonly known as Knowledge Discovery and Datamining (KDD). We consider in this paper the problem of isolating association rules. The problem can be formalized [ZPL97] 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 $\mathrm{support}(A \bigcap B)/\mathrm{support}(A)$.

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. Efficient parallel methods are introduced in this paper.

The paper is organized in five sections. In section 2 we introduce basic sequential and parallel frameworks that are widely used in the literature in order to fix the problems. In section 3 we introduce Radix Trees data structures and their potential use in the process of association rules discovery. Section 4 proposes a novel parallel algorithm that uses Radix Trees for the process of discovering and for the process of implementing operation on Radix Trees. Section 5 concludes the paper.

2. Association Mining algorithms

2.1. Sequential algorithm: Apriori

The "Apriori" sequential algorithm forms the core [AIS93] of all parallel [Zak99, JA] 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.

2.2. Parallel algorithms

2.2.1. Count Distribution The Count Distribution parallel algorithm is simply a parallel version of Apriori algorithm. Each processor has a copy of the database and each processor computes "local" candidates, evaluates the "local" supports and transmits them to a dedicated processor to perform the prefix sum of all of them to obtain the global support of the itemset. Since only the values of supports have to be transmitted to a dedicated processor, the algorithm minimizes communication.

The performance of Apriori or Count Distribution algorithms is limited, even in parallel, for various reasons. First of all, it is required to scan the database at each iteration. Furthermore they enumerate each candidate itemset as many time as we find it in the database even if the transactions are identical.

Second, the transaction database is considered to have an horizontal layout: a transaction has an identifier followed by the items it contains. It appears that this data organization is not suited for the support evaluation phase of the algorithm. Searching a k-subset in a transaction of size s implies to test the $\begin{pmatrix} s \\ k \end{pmatrix}$ subsets of the transaction. Algorithms using vertical transformations of the database suppress the problem. These types of algorithmes are of preferable use as we will see in the next subsection.

2.2.2. The Eclat algorithm An advantage of "Eclat" [ZPL97] faced to "Count Distribution" is that it scans the database only two times. A first time to build the 2-itemsets and a second time to transform it into a vertical form. Eclat algorithm has three steps:

- The initialization phase: construction of the global counts for the frequent 2-itemsets.
- The transformation phase: partitionning of the frequent 2-itemsets and scheduling of partitions over the processors. Vertical transformation of the database.
- The Asynchronous phase: construction of the frequent k-itemsets.

A formal presentation of the algorithm is as follows:

The Eclat algorithm

Initialisation phase:

Scan local database partition
Compute local counts for all 2-itemsets
Construct global L_2 count

Transformation phase :

Partition L_2 into equivalence classes
Schedule L_2 over the set of processors P
Transform local database into vertical form
Transmit relevant tid-lists to other processors

Asynchronous Phase :

for each equivalence class E_2 in local L_2
  ComputeFrequent(E_2)

Final Reduction Phase :

Aggregate Results and Output Associations

This algorithm uses an equivalence class partitionning schema of the database. The equivalence class is based on common prefix assuming that itemsets are lexicographically sorted. For instance AB, AC, AD are in the same equivalence S_A class because of the common prefix A.

Then candidate itemsets can be generated by joining the members of the same equivalence class. For our example, the next candidates of length 3, namely C_3 are ABC and ABD. We can observe that itemsets produced by an equivalence class are always different of those produced by a different class, then the equivalence partitioning scheme can be used to schedule the work over the processors. This method is used in other algorithms such as *Candidate Distribution* and will be used by ours in a different way (using Radix Trees).

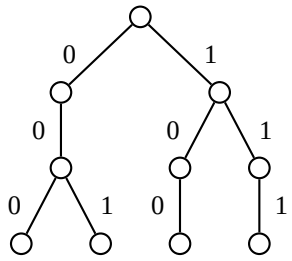
The transformation phase is known to be the most expensive step of the algorithm. In fact, the processors have to broadcast to all other processors the local list corresponding to transaction identifier, for the itemsets.

3. Radix Trees and their use in association mining

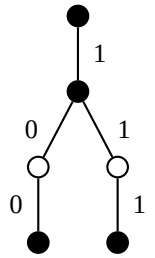
3.1. Introduction

In combinatorics, Radix Trees are used to store sets of strings over an alphabet. In our case, the binary alphabet

is used because we handle integers representing indexes of transactions. There are at least two ways to tackle Radix Trees. It depends on the kind of strings we have. If we use variable length strings, then every nodes in the tree can store a word (internal nodes and external nodes or leaves - see Figure 1). Otherwise if we use fixed length strings, a node stores a word if and only if it is a leaf (see Figure 1).



fixed length strings



variable length strings

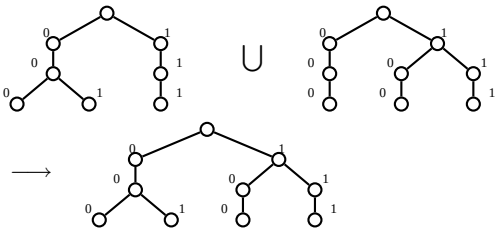
Stored values : 0, 1, 4 and 7.

Figure 1. Different representations of Radix Trees

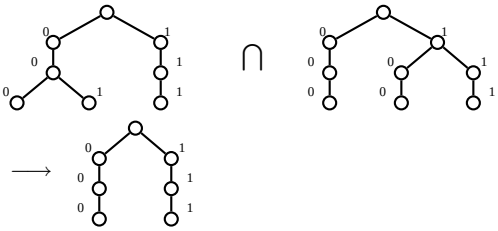
The bottom tree on Figure 1 has two sorts of nodes: black nodes and white nodes. The white color means that no word is stored in a node of this color. Conversely, a black node means that a word is stored in the node. For instance, if we are looking for the word 10 in a Radix Tree with variable length strings, we follow the right edge (1), then we follow the left edge (0) and we check the color of the node. If the color is white, then the word doesn't exist in the tree, otherwise the color is black (see Figure 1) by construction.

In a Radix Tree with fixed length strings, we don't need any colour because if a word doesn't exist then there is no path for it in the tree. We do prefer to use such structure because it is more convenient for implementing tree management operations efficiently and easily.

Radix Trees have the property to represent sets of string in a sorted way. Their tree structure makes the set operations (union, intersection) easier to parallelize. In the operations we consider now (see Figure 2), the size of data is constant and known.



Union of Radix Tree.



Intersection of Radix Tree.

Figure 2. Basic useful operations on Radix Trees

3.2. Operations on Radix Trees

The union of two Radix Trees representing sets is the Radix Tree representing the union of the sets. The intersection of two Radix trees representing sets is the Radix Tree representing the intersection of the sets.

A possible implementation of a tree structure consists in using nodes that contain pointers on successors. This method has some drawbacks, in particular, it uses a lot of memory and it limits the principe of spatial locality (the "next" element to handle is located in memory "near" the current element).

Note also that multithreading operations of Figure 2 is straitforward. For the intersect operation for instance, the principle is as follows. We consider the two roots. If they have two left children, then we add a left child to the new tree and we start a new thread in order to build the "left part" of the intersection. The same construction is made for the right child.

Some technical problems occur with such algorithm. First of all, since we deal with tree height of, say 40 e.g. we handle set of 2^{40} elements, the number of created threads may overpass the physical limit of the operating system. Second, unbalanced computation may occur. For instance, assume that we decide to fix the maximal number of active threads to 2 and we have no thread scheduling mechanism. Starting from the root, we decide to start one thread to realize the intersection for the left child and we also start one thread for realizing the intersection on the right child. If the

left child has many more internal nodes than the right child, the computation will not terminate at the same time and the first terminating thread could be reused for the computation on the left child. So we need also efficient scheduling policies for thread management.

3.3. Radix Trees storage

Due to the huge quantity of data to deal with in practical applications, we have to find methods to store them on disk. Since databases are too important to fit in main memory, we need to balance in and out-of-core computations.

In [Kos04] a method has been introduced and implemented successfully. This method represents the Radix Tree by bit vectors stored on disk.

The solution adopted in [Kos04] to implement Radix trees on disk is to store them on an organization with multiple files.

We choose a file organization with few files by directory in order to avoid costly file system operations and not too large files to operate fast database updates. Indeed, too many files in the same directory could slowdown the application and using too large files cause poor response time for updates. We use a directory tree structure containing small files quickly updatable.

In [Kos04], the items of the database are indexed by storing their identifier in a Radix Tree stored in a directory tree structure where each directory contains three files.

- A file to store the thesaurus (database item lexicon) of the items and the offset of their bit vector on the second file (1).
- A file to store the bit vectors of the identifiers for level n (2).
- A file to store a permutation of the words giving the lexicographical order (3).

To store the Radix Trees on disk, we are using a bit vector of size n per word (file 2) (Radix Trees are using an alphabet of size 2^n). Each prefix designates the next directory containing the path to the database's line of the word. An internal directory may contain 2^n subdirectories.

Let us call k the height of the bit vector hierarchy of Figure 4. Assume the indexes of a found line are $i_0, \cdots, i_{k-1}$ then the corresponding line is $i_{k-1} + 2^n[i_{k-2} + 2^n[\cdots + 2^n[i_1 + 2^n i_0]\cdots]]$.

The permutation file stores a permutation p from $[0, t-1]$ onto itself where t is the cardinality of the thesaurus. The i-th word in lexicographical order is the $p[i]$-th word of the thesaurus. Thus, the thesaurus has not to be maintained in lexicographical order which eases the addition of words because only the permutation file has to be rewritten. A search

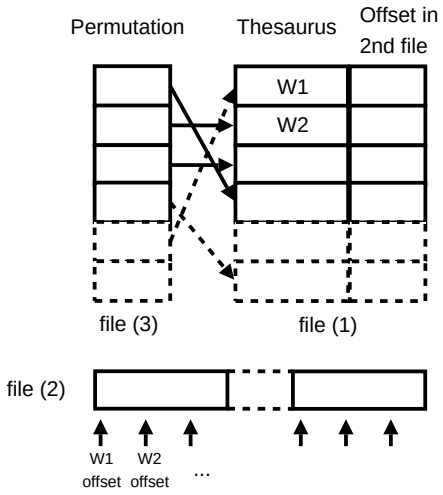


Figure 3. Radix Trees hierarchical representation

in the permutation file is equivalent to a search in a sorted array, so the complexity is in $O(\log(n))$ (where n is the number of words in the directory thesaurus).

In order to search the lines where an item appears, we consult the permutation file (file 3). This gives us the position of the word in the thesaurus (file 1) where we can read its bit vector offset in the file (2). From here, we can deduce the next directories to visit.

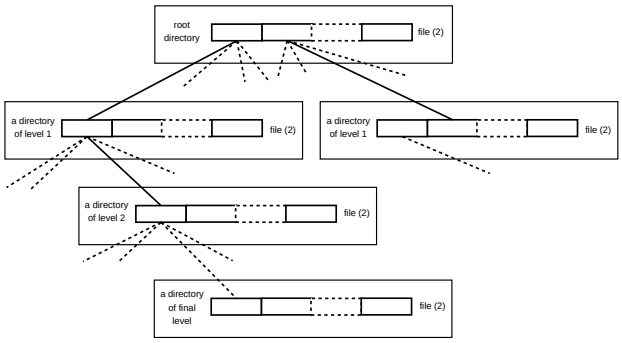


Figure 4. Radix Trees hierarchical representation

The representation of integer set with Radix Trees allows us to save space and to implement efficient searches. Indeed, the common prefixes of different integers are stored only once. Furthermore, each value is inserted and found in constant time (depending of the integer's representation

size), unlike a list structure (linear or logarithmic time depending on the list organization).

Radix Trees are currently used successfully in [Kos04] for building a sequential SQL service as it is defined for database systems [UW02]. The key for efficient parallel implementations of tree management operations (union, intersect) is that computation can be achieved concurrently on each node at a same level in the Radix Tree whose concrete implementation follows Figure 4. We also use Radix Tree structures in the context of association rules discovery, in particular in the context of candidate generation.

4. Radix Tree for parallel association rules discovery algorithms

4.1. Candidates representation

The aim of this section is to show that using Radix Tree for association rules discovery algorithm can improve performance of the whole discovery process. Based on our experience [Kos04] with the use of Radix Trees for the implementation of an SQL service, performance will be improved significantly. First results with our codes implementing a SQL service based partially on Radix Trees demonstrate sigificant improvements (by a factor at least 5) when we experiment with the TPC-C (Transaction Processing Performance Council, benchmark C) and comparing to commercial SQL services.

In [Kos04], Radix Trees are used to code the line indexes of each item in a database. For instance, consider the Accident table of Figure 5. 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 line indexes it occurs at.

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 5. The Accident table.

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, suppose that we have the following query: find all items with the property "Kind of Contract = House" and "Max Amount > 500 000". In order to solve the query, we intersect the corresponding thesaurus Radix Trees. Here we find $\{4, 7\}$.

The intersection can be implemented efficiently because the computational cost is bounded by the number of bits in the representation of integers (in fact we use fixed size alphabet) and not by the number of items in the two sets.

Let us now consider the use of Radix Trees in our context of Association Rules Discovery. The first idea is to put the list of candidates identifier in Radix trees stored locally on each node. For instance, if we have four items A, B, C, D, we start with the complete binary tree as depicted on Figure 6.

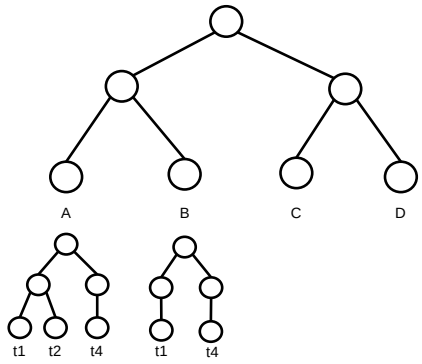


Figure 6. Candidate representation

By adding for each item (A, B, C, D) a Radix Tree that contains the line indexes of transaction identifier (we call such Radix Tree a *transaction tree*), the support evaluation phase consists now in the intersection between transaction trees then by counting the number of leaves. We note here that building the tree of candidates and all the transaction identifier lists can be done in one scan of the database.

The candidates k-itemsets are then represented in the same way. For instance, if the path in the Radix Tree to the item A is 00 and the path to B is 01, the path to AB will be 0001.

Note also that the cost of computing supports is given by a very simple intersection operation between trees. Moreover, as we make progress in the computation, we can store on local disks the partial supports in order to retrieve it efficiently, in case of a reuse.

4.2. Candidates generation

The new candidates for association mining rules discovery can be generated by joining the members of a same

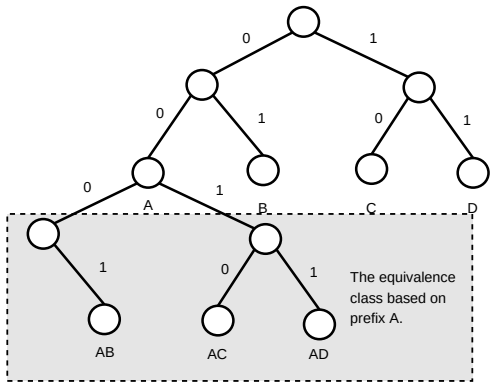


Figure 7. Equivalence class on Radix Tree

equivalence class. By coding itemset in Radix Trees as described on Figure 6, all the members of an equivalence class are in the subtree of the itemset defining the class. Indeed, the itemset that defines a class is the prefix of all the members of this class.

In a Radix Tree where all elements of a subtree have the same prefix, equivalence classes can be viewed as subtrees of it. For instance, the equivalence class S_A is the subtree rooted in A (see Figure 7).

Now, in order to generate the next candidate sets, we have to join the members of the equivalences classes. According to our join operation, Radix Tree implementation of the itemsets consists in rooting the initial subtree on each leaf, considering only leaves obtained by omitting the left neighbours ot the current itemset (see Figure 8).

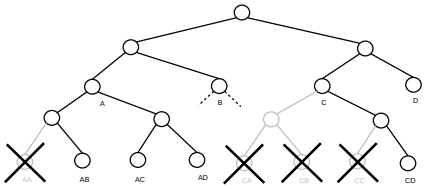


Figure 8. Candidates generation

For instance, to obtain the ABC candidate, we join AB and AC with a Radix Tree rooting operation. To get all candidate sets, we just have to proceed the rooting of subtrees (with elimination of left nodes) on each leaf. In our case $ABC = AB \cup AC$, $ABD = AB \cup AD$, $ACD = AC \cup CD$ etc. Unfolding this algorithms leads to the tree presented on Figure 9.

Moreover, by performing the intersection of the identifier trees list in parallel we obtain the support of the new candidate. As with the Count Distribution Algorithm, the

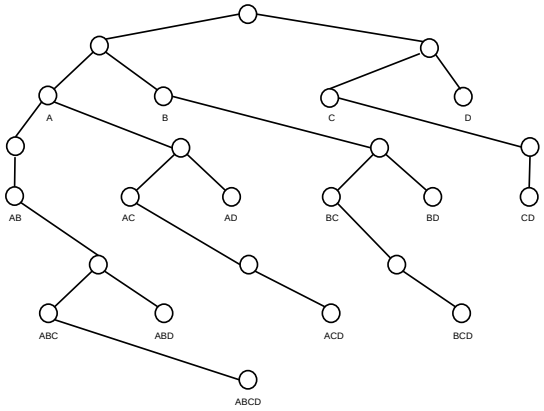


Figure 9. Candidate representation (final)

only communication between processors consists in broadcasting local supports to evaluate the global support.

But before obtaining the distant supports of a candidate set, we can eliminate from the local tree those candidate that don't appear in the local database (i.e supports equal to 0 for instance). We know that if AB don't appear in the base, ABC cannot appear later. We can also overlap the beginning of the next phase of the algorithm with receiving all the supports from other processors.

Indeed, eliminating an invalid candidate which is detected after total support evaluation corresponds to a low cost operation. If the items are homogenous distributed over the transactions, we may assume to anticipate candidates set construction in a good way.

A formal presentation of our parallel algorithm and expected performance are now introduced.

5. Parallel algorithm

First of all, if we have p processors, we assume that the transaction database is splitted into p chunks, one chunck per processor.

Second, in the first part of the algorithm we have to construct the tree of items and their transactions trees. It can be done in one scan of the local database and for each processor in parallel.

To evaluate the support of an itemset, we just have to intersect Radix Trees and proceed to the count of their leaves (that can be done at the same time during the intersect operation). For instance, the support of ABC is the number of leaves of the tree produced by intersecting AB and AC transactions trees. See Figures 8 and 9 for a illustration.

The construction of candidates sets is done as explained above (4.2). Finally, our main 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 de-synchronized */
/* to perform overlapping (see above) */
Wait for all supports from others.
Perform the sum reductions.
Elimination of unsufficient 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 $= \bigcup L_k$

5.1. Hints for complexity analysis

We introduce here some discussion about the cost of each step of the parallel algorithm in terms of time and space.

5.1.1. Construction of the item trees As pointed out previously, our algorithm requires only one pass over the database. This pass aims to build our Radix Trees. Then we operate only on Radix Trees.

5.1.2. Support evaluation and bad candidates elimination We can make the count of tree leaves at the same time we construct the transaction tree i.e by doing intersection operations. The time complexity of an intersection is bounded by the number of different items in the database and not by the number of items in the database. This property justify the use of Radix Trees.

Thus, the local support is known when the candidate set is constructed. If an itemset support is null (i.e. the itemset do not appear in the local database), we can immediatly eliminate it, even if the itemset appears in another partition of the database. For all local support that are not null, we can start a new construction of candidate supersets before knowing the total support. A candidate elimination consists in cutting an edge in a tree.

5.1.3. Candidate set construction The construction of new candidates consists in rooting subtrees on leaves. We eliminate from the tree the candidates that have unsufficient supports and the nodes that make repetition in the subtree rooted (for instance we don't add the item A to the itemset ABCxxx).

In doing this, we do not construct unnecessary candidates. To obtain the support of newly created candidates, we just have to proceed to the intersection of the transaction tree of the added item with the transaction tree of the leaf where it has been rooted. At any time the algorithm "knows" the previous level of the itemset's tree to generate new candidates and evaluate their supports. So, we can save memory by deleting the upper levels.

5.2. Scheduling policies for thread management

Radix Trees operations can also be parallelized by using threads. This is particular useful if we run the algorithm on a parallel machine with SMP nodes. Let us consider the union operation, in parallel, of two Radix Trees.

Starting from the roots of the two trees, one strategy is to activate a thread for computing the union of the two left children and to activate a thread to compute the union for the two right children. We apply recursively this principle until we reach the maximum number t_{max} of authorized threads. In this case, we have to wait for the completion of one thread before going on.

The key idea of our thread management strategy is the following. When a node has two subtrees and it remains an idle thread, we use it on one of the subtrees. The other subtree will be managed by the thread running on the current node.

An operation (union) on a node is completed when it is also completed on the subtrees of the node. So, to minimize the idle time on each processor, we decide to launch threads on the subtree containing the less number of nodes. In the worst case, the thread finishes its work in the same time than the thread which launched it. At this point, the thread can declare itsef as an available thread without waiting for the result of the other subtree.

Pictorially speaking, we proceed as depicted on figure 10 where each arrow symbolizes the work of a processor. Moreover, in the reminder of the subsection, we consider the case $t_{max} = 4$.

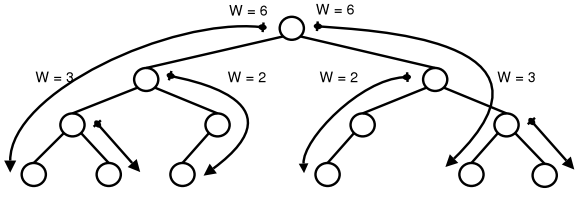


Figure 10. Thread management policy.

On figure 11 we can see that we have no idle time when we use the scheduling policy described above compared to the opposite scheduling policy illustrated on figure 12.

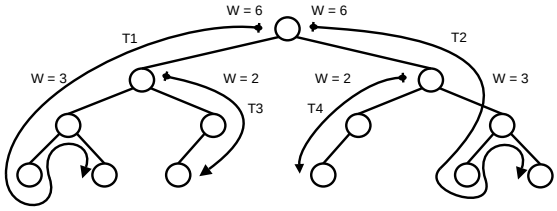


Figure 11. Preferred policy with 4 processors.

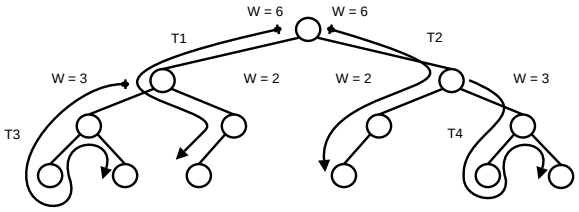


Figure 12. Invert policy with 4 processors.

On Figure 11, threads T1 and T2 finish their work one round after T3 and T4. Then, T3 and T4 can be reused in other operations.

On Figure 12, threads T1 and T2 finish their work one round before T3 and T4 but have to wait the completion of T3 and T4 to root the results.

6. Conclusion

In this paper we have introduced a parallel algorithm using Radix Tree structures in order to discover association rules in a transaction database. Our algorithm has many interesting features. It scans the base only once, performs candidate generation in parallel with only few integer exchanges (representing supports computed locally) between processors.

Based on our experience [Kos04], we guess that implementations will encompass existing implementation because we know that one key to get performance for association mining is the way we manadge intersect operation. In our case, we have proposed a new approach that permit us to compute the candidate support by the intersection of Radix Trees.

Radix Trees offers a good compromise [Kos04] between the storage size required to store them and the efficiency to retreive any information mapped to integers. We are currently implementing the association rules discovery algorithm presented in this paper and we also implement an efficient multithreaded library in order to accomplish, in parallel, Radix Tree operations.

References

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD Int. Conf. on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.

[JA] Ruoming Jin and Gagan Agrawal. An effi cient association mining implementation on clusters of SMP. pages 156–156.

[Kos04] Michel Koskas. A hierarchical database management algorithm, To appear in the annales du Lamsade, 2004, url: `http://www.lamsade.dauphine.fr`

[UW02] Jeffrey D. Ullman and Jennifer D. Widom. *First Course in Database Systems, A, 2/e*. Prentice Hall, 2002.

[Zak99] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, /1999.

[ZPL97] Mohammed Javeed Zaki, Srinivasan Parthasarathy, and Wei Li. A localized algorithm for parallel association mining. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 321–330, 1997.