a Hierarchical Database Manager

Michel Koskas

a Hierarchical Database Manager

Michel Koskas*

Résumé

Cet article décrit un nouvel algorithme [1] permettant de gérer des bases de données. Son champ d'application le plus naturel est néanmoins le datawarehouse (OLAP). Il repose sur une représentation dénormalisée de la base. Les données sont stockées dans des thesaurus et des arbres à préfixes (une représentation hiérarchique de champs de bits) qui ont des propriétés intéressantes.

Mots-clefs : base de données, champs de bits, arbres à radicaux, stockage hiérarchique

Abstract

This paper describes a new algorithm[2] dealing with databases. This algorithm allow to fully manage a database, but their most natural field of applications is the datawarehouse (OLAP). It lies on a de-normalized representation of the database. The data is stored in thesauruses and radix trees (a hierarchical representation of bitmaps) which have interesting properties.

Key words : database, bitmaps, radix trees, hierarchical storage

1 Introduction

It is often said that database sizes grow by a rate of 10 % a year and that this growth is greater than the one of the abilities of computers. Databases are more and more used

* LAMSADE, Université Paris-Dauphine, 75775 Paris cedex 16, France. kos-kas@lamsade.dauphine.fr

1. brevet en cours de dépôt

[2]patent pending

in more and more fields: social actors, armies, commercial agents use more and more data. A pertinent use of an enormous amount of data may show a huge profit to the data owner. For instance the “wallmart” stores managers realized, thanks to data mining, that on saturdays, customers who bought pampers for babies usually also bought beer. They re-arranged their stores in order to put aside the beer and the pampers. The result was that the sales of both these articles rose up. (The admitted explanation is that on saturday, it is more often men who make home shopping.)

Usually, to deal with databases, one may use multidimensional arrays, special indexes (bitmaps), relation caching, optimized foreign key joins, B-trees or approximation. The algorithm presented in this paper uses radix trees. It shall be denoted the A-algorithm. These trees may be understood as a hierarchization of bimaps vectors. It allows one to answer to SQL queries or to manage the base (to add or remove tuples, a primary key, a foreign key or an attribute from a relation or even to add or remove a relation to or from a database).

We show in the next relation a comparison between programs written in C++ designed to use these algorithm and the same requests performed on the same machine but using the three most popular commercial products allowing databases management. The requests were taken from the TPC (see [6]).

When one deals with databases, one may have to answer to two very different kind of queries: one of them is “What is the content of attribute C in the relation T at the record Id 17?” and the other is “At which record Ids may I find this given tuple for the attribute C in relation T?”.

The first query may be very easily answered by reading the relation the wanted attribute belongs to.

But for this first request, there is a case in which the answer is not that easy. This is the case when the attribute C does not belong to T but to a relation T' linked to T by foreign keys and primary keys.

One may answer this kind of request (“What is the content of tuple with record Id 17 of attribute C in relation T with C not belonging to T”) by using a de-normalized data representation.

One may also answer very easily to a request like “Where may I find this given tuple in attribute C in relation T?” by using radix trees (radix trees may be seen as a hierarchical representation of bitmaps indexes). The bitmaps are widely used in database management. One may refer to [10] for a recent work in this matter.

The data of databases is very often stored in B-trees (see [4], [3] or [5] for instance.).

The complexity of the computation of an "and" request with the A-algorithm is averagely $O(i \ln L)$ where i is the cardinality of the intersection and L the maximum of the cardinalities of the numbers of records of the relations involved in the request. In the

worst case (whose probability of appearance tends to 0 when the size of the data tends to infinity), this complexity of this computation is $O(L \ln L)$. This is the complexity of the algorithms using balanced trees for instance.

The complexity of an "or" request with the A-algorithm is $O(L \ln L)$, which is also the case of the use of B-trees. The complexity of insertion, suppressions or updates are, with the A-algorithm, $O(\ln L)$. These operations are also performed in $O(\ln L)$ with algorithms using B-trees.

The reader may refer to [8] or [9].

1.1 Plan of the paper

The paper is organized as follow: the section 1 introduces the problem discussed in this paper. Its plan is the present subsection (1.1). The next subsection is dedicated to a presentation of the TPC benchmark (1.2) and the performances of the A-algorithm are presented in 1.3.

The next section is devoted to an introduction to radix trees (2). The two next subsections (2.1 and 2.2) explain how one can perform set operations over radix trees.

The next section deals with the creation of the indexes of a database using radix trees (3). A fundamental case is when the database is made of a single relation itself containing a single attribute. The subsection 3.1 explains it. The next subsections detail the creation of the thesaurus (3.1.1), the storage of the indicative functions (the sets of records ids of each word of the thesaurus are stored in radix trees, subsection 3.1.2). It is convenient to use macro words to accelerate the computations of between clauses (subsection 3.1.3). The two next subsections give details of the storage of the attribute (3.1.4 and 3.1.5) and the next one summarizes the storage of an attribute (3.1.6).

The two subsections are dedicated to cases when a relation has several attributes (3.2) of when the database has several relations (3.3).

Once the indexes are built, one may request the database (section 4). The first step is to compute the expansion relation and to remove the join clauses (4.1). The atomic requests are treated in the subsection 4.2. An important case is the between (4.2.1) because it has several sub cases (4.2.2, 4.2.3, 4.2.4, 4.2.5). Then one may mix these atomic cases to perform any “where” clause which does not contain sub requests (4.3). Its logical connectors are the “or” (4.3.1), the “and” (4.3.2) and the “not” (4.3.3). A more problematic case is the case of comparison between attributes (4.4) which is very similar to a cartesian product (4.5) Then one has to manage the sub queries when they are correlated (4.6) or not obviously (4.7). The last step is to perform computations on the tuples which are at the record ids found in the “where” clause (4.8).

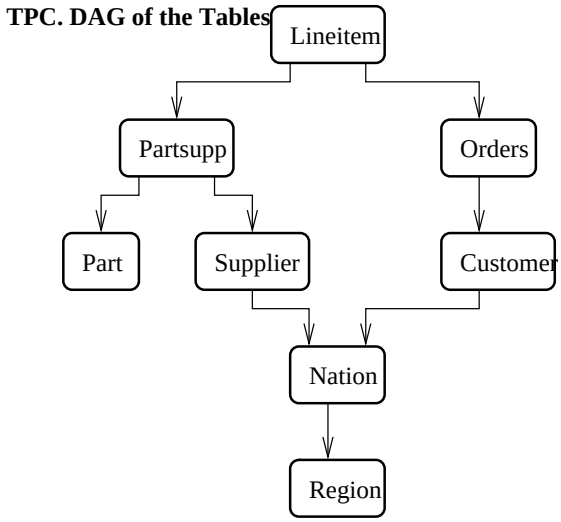

Figure 1: The dag of the relations of the TPC

The next section deals with the base management (5). One may manage a relation (5.1) by adding or removing records (5.1.1 and 5.1.2), add or remove an attribute to a relation (5.1.3 and 5.1.4), add or remove a primary key or a foreign key (5.1.5, 5.1.6, 5.1.7, and 5.1.8), add or remove a relation (5.2 and 5.2.1)

The before to last section (6) is the conclusion and the last section is dedicated to aknoledgements (7).

1.2 The TPC

The TPC (the Transaction Processing Performance Council, see [6]) is a benchmark designed to measure the performances of database manager programs. One can download a relational database made of eight relations: Lineitems, Partsupp, Part, Supplier, Orders, Customer, Nation and Region. This base may be scaled by a scale factor as big as 1000. When its tuple is 1, the size of the database is roughly 1 GB. In this case, the relation Lineitem is made of 6 millions lines, Partsupp of 800,000 lines, Part of 200,000 lines, Supplier of 10,000 lines, Orders of 1,500,000 lines, Customer of 150,000 lines, Nation of 25 lines and Region of 5 lines. The dag of the relations is as follow (an arrow between two relations T_1 and T_2 from T_1 to T_2 means that the relation T_1 contains a foreign key replicating a primary key of T_2 (see figure 2).

The attributes of the relations were the following:

The tpc benchmark contains 22 queries : 20 of them are queries of the data and the two last queries are insertion and suppression of 10% of the lines of `lineitem`.

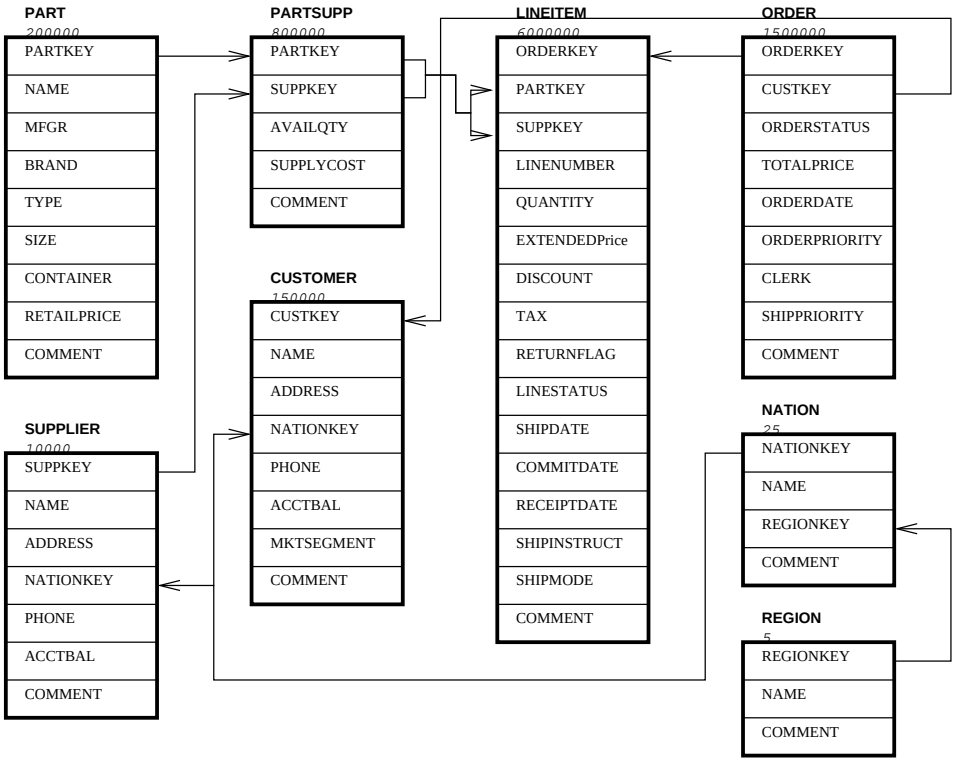

Figure 2: The attributes of the relations of the TPC

The queries performed for this paper where the $Q1$, $Q6$, $Q17$ and $Q19$. These requests are:

Q1:

```
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as
sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 +
l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    lineitem
where
    l_shipdate <= date '1998-12-01' - interval ':1' day
(3)
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus;
```

Q6:

```
select
    sum(l_extendedprice * l_discount) as revenue
from
```

```
    lineitem
where
    l_shipdate >= date ':1'
    and l_shipdate < date ':1' + interval '1' year
    and l_discount between :2 - 0.01 and :2 + 0.01
    and l_quantity < :3;
```

Q17:

```
select
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    lineitem,
    part
where
    p_partkey = l_partkey
    and p_brand = ':1'
    and p_container = ':2'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem
        where
            l_partkey = p_partkey
    );
```

Q19:

```
select
    sum(l_extendedprice* (1 - l_discount)) as revenue
from
    lineitem,
    part
```

```
  where
      (
          p_partkey = l_partkey
          and p_brand = ':1'
          and p_container in ('SM CASE', 'SM BOX', 'SM
PACK', 'SM PKG')
          and l_quantity >= :4 and l_quantity <= :4 + 10
          and p_size between 1 and 5
          and l_shipmode in ('AIR', 'AIR REG')
          and l_shipinstruct = 'DELIVER IN PERSON'
      )
      or
      (
          p_partkey = l_partkey
          and p_brand = ':2'
          and p_container in ('MED BAG', 'MED BOX', 'MED
PKG', 'MED PACK')
          and l_quantity >= :5 and l_quantity <= :5 + 10
          and p_size between 1 and 10
          and l_shipmode in ('AIR', 'AIR REG')
          and l_shipinstruct = 'DELIVER IN PERSON'
      )
      or
      (
          p_partkey = l_partkey
          and p_brand = ':3'
          and p_container in ('LG CASE', 'LG BOX', 'LG
PACK', 'LG PKG')
          and l_quantity >= :6 and l_quantity <= :6 + 10
          and p_size between 1 and 15
```

Request	A-algo	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.

Table 1: Performances of the A-algorithm compared to SQL Server 2000, Oracle 8i and DB2

```
          and l_shipmode in ('AIR', 'AIR REG')
          and l_shipinstruct = 'DELIVER IN PERSON'
      );
  p_partkey = l_partkey
          and p_brand = ':3'
          and p_container in ('LG CASE', 'LG BOX', 'LG
PACK', 'LG PKG')
          and l_quantity >= :6 and l_quantity <= :6 + 10
          and p_size between 1 and 15
          and l_shipmode in ('AIR', 'AIR REG')
          and l_shipinstruct = 'DELIVER IN PERSON'
      );
```

1.3 Performances

The comparison between programs using the algorithm detailed in this paper and the main programs one can buy were all performed on the same PC, using a single processor of 2GH, 1GB of RAM, and all the programs written in C++ ; the data was the TPC data using a scale factor of 1, so the size of the database (the flat relations) was roughly 1 GB. The cartesian product was performed over two copies of the main relation of the TPC, the relation lineitem, holding 6 millions lines. DBM1 is Microsoft SQL Server 2000, DBM2 is Oracle 8 and DBM3 is IBM DB2).

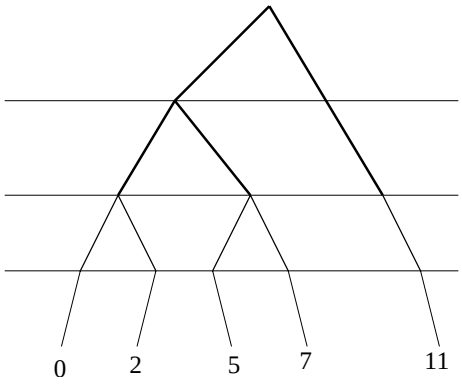

Figure 3: The set $\{0, 2, 5, 7, 11\}$ stored in a radix tree

The algorithm is based on a full use of hierarchical data representation (by the use of radix trees), for the data and the record Ids they belong to.

In a first part we recall the use of radix trees. This tool will be very useful to fully manage the database.

2 Radix trees

A radix tree is a convenient mean to store a set of integers or words of a dictionary, especially when they have all the same length. When dealing with integers, one can manage to force them to have the same length, by adding prefixes made of repeated 0s. A radix tree stores its elements in its leaves. When storing numbers written in basis 2, the nodes may only have a left son (labeled with 0) or a right son (labeled with 1). The path between the root of the tree and one of its leaves writes a word onto the alphabet $\{0, 1\}$ and this word form the digits of the stored integer.

Let us then consider for instance a set of integers written in basis 2 and of same length in this basis, S. One can store S in a tree whose paths between the root and the leaves are the integers of S. For instance, the set $S = \{0, 2, 5, 7, 11\} = \{0000, 0010, 0101, 0111, 1011\}$ may be stored as (see figure 3).

The advantages of storing a set of integers in such a way are numerous: the storage is efficient because common prefixes are stored only once in the tree, and, as we will see in the next subsections, the computations over sets of integers are quite easy to perform and efficient.

An algorithm to build a radix tree whose leaves are the elements of a set S is the following:

Algorithme 1

1. RadixTree Build(set S, height H)
2. Parameters: a set S and H = 1 + $\lceil \ln_2(Max(S)) \rceil$[3]
3. Result: a Radix Tree
4. Result = Node
5. if (H = 0) return Result.
6. Build $S_0 = S \cap [0, 2^{H-2} - 1]$ and $S_1 = S \cap [2^{H-2}, 2^{H-1} - 1]$
7. if ($S_0 \neq \emptyset$) Result->LeftNode = Build(S_0, H-1)
8. if ($S_1 \neq \emptyset$) Result->RightNode = Build($S_1 - 2^{H-2}$, H-1)
9. return Result

2.1 Intersection

Let S and S' be two sets of integers. We wish to compute the intersection $S \cap S'$ and let us denote s and s' their cardinalities.

One way to do it is to sort the two sets (which costs $O(s \ln s + s' \ln s')$and to compute this intersection of the sorted sets in time $O(max(s, s'))$. So this intersection may be computed in a time like $O(s \ln s + s' \ln s')$.

One may also sort only one of the sets, say S and look for every element of S' in the sorted set S. The cost is like $O(s \ln s + s' \ln s) = O((s + s') \ln s) \leq O(s \ln s + s' \ln s')$.

Now if we suppose that S and S' were stored in radix trees, the cost of the intersection is like $O(i \ln s)$ where $i = \#(S \cap S')$, where $\#(S)$ is the cardinality of the set S. Indeed, the intersection between the two radix trees may be performed level by level

2.2 Union

The cost of computing the union of two sets of integers, S and S', of cardinals s and s' is the cost of making a multi-set union plus the cost of computing the intersection $S \cap S'$ in order to remove the common elements to S and S'.

[3]the parameter H has got not to be re-computed at each recursive call

Here again, one can begin by sorting the two sets and then compute $S \cup S'$. The cost of this algorithm is $O(s \ln s + s' \ln s' + s + s') = O(s \ln s + s' \ln s')$.

In a similar manner, one may sort one of the sets, say S and compute $S \cup S'$ by looking for each element of S' in S. The cost of this algorithm is $O(s \ln s + s' \ln s) = O((s + s') \ln s)$.

Now if we suppose again that S and S' are stored in radix trees, the cost of the computation of $S \cup S'$ is $u \ln s$ where u is the cardinal of $S \cup S'$. Indeed, the two trees may be read simultaneously and the resulting tree may be computed on the flight.

3 Creating indexes

In this section we will explain how one can use radix trees to build convenient indexes to store and manage databases.

In a first subsection, we will suppose that the database is made of only one relation which contains only one attribute. This case, though artificial, is fundamental to understand the proceed described in this paper.

Then we will suppose that the database is composed of one single relation, made of several attributes and at least one Primary Keys. It may be convenient to suppose that a relation may contain several Primary Keys. Indeed, in practice, it may so happen that a Primary Key, made of several attributes, could be only partially filled while another could be fully filled.

The last subsection we be dedicated to the indexes creation of a full database.

3.1 One relation, one attribute

Primary Keys. A primary key is an attribute, or a set of attributes such that two different tuples of the relation may not have the same tuples on this attribute (or all these attributes).

There is one implicit and convenient Primary Key in any relation : the record Id (it is indeed a Primary Key because no two different lines have the same record Id). So we will assume that the tuples of the relation are identified by their record Ids.

If one has to store, request and manage a date base made of one single relation made of only one attribute, one may compute the thesaurus of the attribute and then, for each word of this thesaurus compute the set on integers it appears at.

Then each set may be stored in a radix tree as explained above.

0	Male
1	Female
2	Female
3	Male
4	Female
5	Male
6	Male
7	Female
8	Female
9	Male
10	Male

Table 2: An example of simple relation

3.1.1 Thesaurus creation

Let us notice that this step necessitates a sort : one has to build the set of couples (word, record Id), which is sorted according to the first element and according to the second for the couples which have the same first element. Then one builds on the thesaurus and the set of record Ids each of these words appear at.

Let us take an example: let us consider the following relation (see table 2).

(In this example, the record Ids are indicated explicitly.)

One builds the couples (Male, 0), (Female, 1), (Female, 2), (Male, 3), (Female, 4), (Male, 5), (Male, 6), (Female, 7), (Female, 8), (Male, 9), (Male, 10)

and sorts them according to the first element of the couples:

(Female, 1), (Female, 2), (Female, 4), (Female, 7), (Female, 8), (Male, 0), (Male, 3), (Male, 5), (Male, 6), (Male, 9), (Male, 10).

Then one is able to build the thesaurus and, for each word of the thesaurus, the set of record Ids this word appears at:

“Female” appears at record Ids $\{1, 2, 4, 7, 8\}$ and “Male” appears at record Ids $\{0, 3, 5, 6, 9, 10\}$.

When this is done, it is easy to answer a request like "What are the record Ids the word “Male” appears at?", but quite uneasy to answer to the request “what is the tuple at record Id 7?”. For this last request, see subsection 5 below.

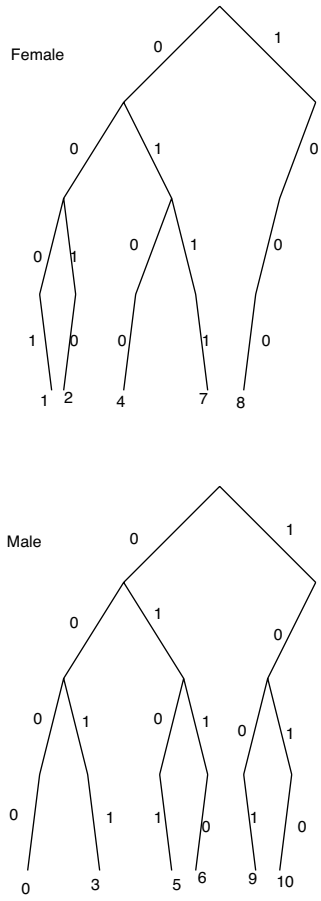

Figure 4: Example of the representation of an attribute of a relation

3.1.2 Storing the indicative functions

Now these sets of record Ids each word of the thesaurus appear at can be stored in radix trees. This is convenient and powerful to compute intersections, and so on. . .

In the preceding example, one has: (see Figure 4)

3.1.3 Creating macro-words

Another question one may have to answer to when dealing with the attribute of a relation of a database is a between: one may want to know for instance for which record Ids the words lye between two given values.

Let us imagine for instance that an attribute is made of dates, formated in YYYYM-MDD. Compare two dates is compare lexicographically the two words.

But we may also enlarge the thesaurus, with words that are truncates of the initial words. Let us indeed add words to the thesaurus of the attribute, for instance any truncate of six characters or any truncate of four characters.

Then any word of the thesaurus will be represented three times: one time as itself, one time as a truncate of six characters and one time as a truncate of four characters.

Any word of six characters, say aaaamm, will occur each time a word aaaammxx occurs. In other words, the record Ids the word aaaamm appears is exactly the union of the sets of record Ids any word aaaammxx appears at.

In a similar manner, any word of four characters, say aaaa, will occur each time a word aaaaxxyy occurs and its radix tree will be the union of the corresponding radix trees.

In summary, one builds not only the radix trees of each word of the thesaurus but also the thesaurus of each prefix of given lengths of words. So when one has to solve a "between" clause, one splits the wanted interval with respect to the prefix length pre-computed and read the matching radix trees. For instance, the interval $[19931117, 19950225]$ would demand, without the macro words, 466 reading of radix trees because this interval contains 466 different words. If one splits this interval with respect to prefix lengths of 6 and 4, one has: $[19931117, 19950225] = [19931117, 19931130] \cup [199312, 199312] \cup [1994, 1994] \cup [199501, 199501] \cup [10050201, 19950225]$. The first interval contains 14 different words (not truncated). The second contains a single truncated word (6 characters), and the reading a single radix tree gives the set of the records Ids words like $199312dd$ appear at. The third interval contains one single truncated word (4 caracters) and the reading of the single matching radix tree gives the set of records Ids words like $1994mmdd$ appear at, and so on... Finally only 42 readings of radix trees are made necessary instead of 466.

3.1.4 Managing lacks

Now, it may also so happen that some tuples were not filled. But each must have an attribute, even an attribute meaning that there is no attribute at this record Id.

The tuples meaning a lack of information should be chosen in a way as few disturbing as possible, which means we should choose very seldom tuples. We may for instance chose : $\#Empty\#$ for a string, -2^{31} for a signed integer on 32 bits, $2^{32} - 1$ for an unsigned integer on 32 bits, -2^63 for a signed integer on 64 bits, $2^{64} - 1$ for an unsigned integer on 64 bits and so on...

0	Male
1	Female
2	Female
3	Male
4	Female
5	Male
6	Male
7	Female
8	Female
9	Male
10	Male

Table 3: An attribute

0	Female
1	Male

Table 4: The thesaurus

3.1.5 An additional storage

As explained above, the storage of an attribute by thesaurus and radix trees makes quite uneasy to answer a question like “what is the tuple at record Id 17?” for instance.

This is why it is necessary to store the attribute in its natural order. Of course, instead on storing the attribute itself, it may be much more affordable to store the word indexes in the thesaurus.

For instance, the preceding attribute shall be stored:

shall be stored as

and the attribute:

remark it sometimes happen that a word could appear or disappear from a thesaurus while adding or removing records to a relation. In this case, we might think we have to rewrite the whole attribute each time this situation happens. This is nevertheless not true: one may store an unsorted thesaurus and a permutation which stores is contents. Thus when words are no longer in the s-thesaurus or when a new word appears in it, on has only to re-write the permutation instead of the whole attribute.

0	1
1	0
2	0
3	1
4	0
5	1
6	1
7	0
8	0
9	1
10	1

Table 5: An cheap storage of the attribute

3.1.6 Summary of the full storage of an attribute

3.2 One relation, several attributes

Now when a relation has several attributes, each one of them may be treated as if it were the only attribute of the relation. *This means to say that there should exist a thesaurus for each attribute and the matching radix trees for all word of any of these thesauruses.*

The only remaining question is the storage of the primary keys.

When dealing with a Primary key, one has to be able to answer efficiently to two questions: at what record Id can we found a given tuple of a primary key, and what is the tuple of the primary key at a given record Id.

One may answer efficiently to both these questions by storing the attribute or the attributes of the primary key in its (or their) natural order, namely with increasing record Ids and by storing in more a permutation allowing one to find a given tuple efficiently.

For instance, let us imagine a primary key made of two attributes, whose tuples are:

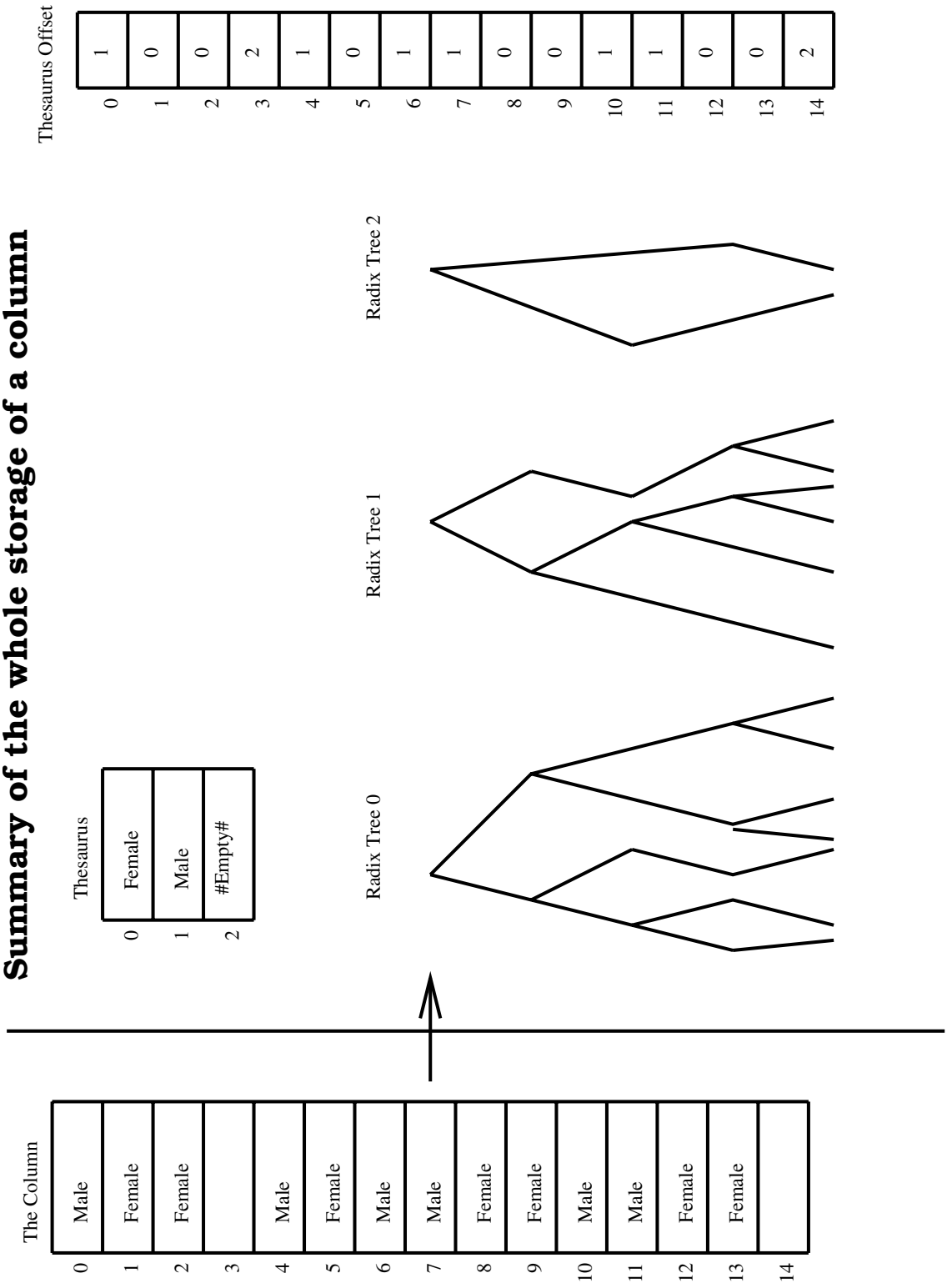

Figure 5: Summary of the whole storage of an attribute

(0)	1	3
(1)	2	1
(2)	3	2
(3)	2	3
(4)	1	2
(5)	3	7
(6)	2	2
(7)	1	1
(8)	3	3
(9)	4	3

In this example, the record Ids are still explicitly expressed between parentheses. One then store these two attributes exactly as they are and a permutation. To store the permutation, one has to chose a comparison function. For instance one may compare first the first attribute and the second in case of equality.

In this case, the sorted primary key is:

(7)	1	1
(4)	1	2
(0)	1	3
(1)	2	1
(6)	2	2
(3)	2	3
(2)	3	2
(8)	3	3
(5)	3	7
(9)	4	3

By removing the tuples (but keeping the record Ids) one obtains the permutation (7401632859) and thus is able to find a given value by dichotomy.

When storing a whole relation, it is also convenient to store the number of its records.

3.3 Several Relations

In a relational database, there are usually several relations linked between them by foreign keys recalling primary keys.

As we explained, a primary key is an attribute or a set of attributes whose tuple may be an unique identification of the record within the relation (the record Id is a fundamental example of primary key. See [1] or [2]).

Let us suppose that a relation is made of several billion of records, but that some attributes may take only five different tuples (for instance, in a genealogy database, one may want to store for each client the country, the continent the customer was born, the country, the continent where his mother was born and the country and the continent his elder child, if any, was born). Instead of recalling fully the names of all these countries and continents for each record, one may build two other relations, one of countries and another of continents. Then on each record, instead of recalling all these countries and continents, one may recall only the primary key of the relation of the countries for the customer, his mother and elder child if any. And in the relation of the countries, one may also recall only the primary key of the relation of the continents the country belongs to. This storage is much cheaper.

Here is a little example of such a practice:

(li)	cn	Inc	BirCoun	BirCont	MoCoun	MoCont	EldCoun	EldCont
(0)	Dupont	817	France	Europe	Tunisia	Africa	England	Europe
(1)	Gracamoto	1080	Japan	Asia	Japan	Asia	USA	America
(2)	Smith	934	England	Europe	India	Asia	England	Europe
(3)	Helmut	980	Germany	Europe	Germany	Europe	Germany	Europe

(cn means “customer name”, Inc “Income”, “BirCoun “Birth Country”, “BirCont “Birth Continent”, MoCoun “Mother’s birth country”, MoCont “Mother’s birth Continent”, EldCoun “Elder’s birth country” and EldCont “Elder’s birth continent”.)

This relation may be rewritten in several relations:

Continents:

Continent	
(li)	Continent
(0)	Africa
(1)	America
(2)	Asia
(3)	Europe

Country		
(li)	Country	Continent
(0)	France	3
(1)	Tunisia	0
(2)	England	3
(3)	Japan	2
(4)	USA	1
(5)	India	2
(6)	Germany	3

And the customers' relation becomes thus:

Customers					
(li)	cn	Inc	BirCoun	MoCoun	EldCoun
(0)	Boyer	817	0	1	2
(1)	Gracamoto	1080	3	3	4
(2)	Smith	934	2	5	2
(3)	Helmut	980	6	6	6

and the set of three relations is indeed much shorter to store than the full relation.

But this also points out that any relational database may be seen as a set of independent relations.

In the preceding example for instance, we can consider the relation continent by itself, the relation country with the relation continent expanded inside and the relation people with the relation country and continent expanded inside (which is the very first relation, the full one, of this example).

These expansion relations are thus:

Expanded Continents	
(li)	Continent
(0)	Africa
(1)	America
(2)	Asia
(3)	Europe

Expanded	**Countries**	
(li)	Country	Continent
(0)	France	Europe
(1)	Tunisia	Africa
(2)	England	Europe
(3)	Japan	Asia
(4)	USA	America
(5)	India	Asia
(6)	Germany	Europe

Expanded Customers								
(li)	cn	Inc	BirCoun	BirCont	MoCoun	MoCont	EldCoun	EldCont
(0)	Boyer	817	France	Europe	Tunisia	Africa	England	Europe
(1)	Gracamoto	1080	Japan	Asia	Japan	Asia	USA	America
(2)	Smith	934	England	Europe	India	Asia	England	Europe
(3)	Helmut	980	Germany	Europe	Germany	Europe	Germany	Europe

Of course, it may so happen, like in this example, that a relation should be expanded several times in another. This means that the attributes of an expanded relation should be refereed to as the attribute of the expanded relation expanded in the expansion relation via the list of couples (Primary Key Foreign Key) allowing one to move from the expansion relation to the expanded relation.

Now we define an expansion relation as a relation in which as much relations as possible were expanded in. From now on, we will consider only expansion relations and the database will be made, from now on, of independent expansion relations.

For each of these expansion relations, one can build the indexes as explained above.

And now, we are ready to request or manage the database.

4 Requesting

In this section we explain how one may use the indexes created as explained below to perform efficient SQL requests. Usually, a request involves several relations. It may be split in two parts: the first part means to discriminate record Ids and the second part (if any) means to perform computations over the data of the found records.

The first part may contain join clauses (the link between a foreign key and the matching primary key), comparison between an attribute and a constant (with arithmetic connectors as “=”, “$\geq$”, “>”, “$\leq$”, “<”, between, like, in…), or a comparison between two attributes (for instance like in a cartesian product), theses requests being logically connected by logical connectors like “and”, “or”, “not” ….

The second part may contain arithmetic operations like a sum, a mean, a product, a star operator, ….

4.1 Removing the join clauses: choice of the expansion relation

As explained above, each of the relations, say R, is considered as an "expansion relation" which means that any relation R' linked to R via a foreign key are expanded in R.This means that the attributes of R' are developped in R, the thesauruses of these attributes are stored as the ones of R and the matching radix trees are computed. So the join clauses are irrelevant in such a relation.

But a request involves usually several relations. How should we chose the appropriated expansion relation? The relations involved in the request are all expanded in a nonempty set of relations, say $\mathcal{T}$. Exactly one of these relations is expanded in none of the others. This relation is the expansion relation appropriated to solve the request.

Now, the where clause may contain some join clauses. These clauses must be logically linked to the remaining part of the request by an "and" operator. So the first step consists in simply remove these clauses by replacing the (Join Clause And Remaining) part by (Remaining).

Now let us study how we can manage the where clause cut down from its join clauses

4.2 Atomic requests

We call here an atomic request a fundamental part of a where clause, namely a comparison clause linked to the remaining of the where clause by logical operators. If t is a relation and c one of its attributes, an atomic clause may be `t.c = 3`, `t.c between "HIGH" and "MEDIUM"`, or `t.c like Word%` for instance.

We explain in the above subsections how to deal with the atomic requests.

4.2.1 Equality between an attribute and a constant

This is the simplest case: one has only to find the wanted value in the thesaurus, read its radix tree which gives him the record ids this word appear at.

4.2.2 Between

This is the fundamental example of atomic request. Any of the others may be treated as a between. It is the clause the macro words where made for.

Let us take back the "date" example given below: one made macro words of length 4 and 6 for an attribute containing dates and wishes to compute the record Ids dates lying in $[19931117, 19950225]$ appear at.

As explained above, one may split the interval by same common prefixes length than the macro words lengths. Thus, one obtains $[19931117, 19950225] = [19931117, 19931130] \cup [199312, 199312] \cup [1994, 1994] \cup [199501, 199501] \cup [10050201, 19950225]$.

The computation is then simple: one read the radix tree of the 19931117, "OR" it with the radix tree of 19931118, ..., "OR" the result with the radix tree of 199312 (the macro word whose radix tree is precisely the "OR" of the radix tree of all the dates beginning with 199312), then "OR" the result with the radix tree of the macro word 1994 (whose radix tree is the "OR" of the radix trees of all the dates beginning with 1994 and so on....

As explained above, one reads only 42 radix trees to perform this computation instead of 466...

Of course one can also manage opened or semi opened intervals by simply excluding the corresponding word.

4.2.3 Greater than, lower than, greater than or equal, lower than or equal

Each of these atomic requests is in fact a between. Indeed, if we call `m` the minimum value of the thesaurus and `M` its maximal value, then any of these requests are either of the form (m, a) or of the form (a, M). So if we can manage the between clause, we can also manage these atomic requests.

4.2.4 In

The `in` clause is a way of mixing equality clauses and Or logical connections. So we can manage them simply.

For instance, `t.c in (a, b, c)` may be rewritten in `t.c = a or t.c = b or t.c = c`. The management of the `or` clause is explained below.

4.2.5 Like

The like clause is another example of between clause: for instance, the clause `t.c like word%` may be rewritten in: `t.c between [word, wore[`. Here again, manage the `between` clause also manages the `like` clause.

4.3 Mixing atomic requests

Now the where clause may mix atomic clauses by using logical operators: the `or`, the `and` and the `not` clause. The three next subsections are dedicated to these logical

clauses.

We would like to empathize that the result of an atomic request is a radix tree.

We will suppose (and show) that this is the case of any where clause.

4.3.1 Or

Now we have to OR two radix trees: The clause is `(Left Clause OR Right Clause)`. The `Left Clause` and `Right Clause` when solved, return a radix tree. So all we have to do is to compute recursively the resulting radix tree of the full clause.

4.3.2 And

The `And` clause may be performed exactly as the `Or` clause. However, the computation is a little more efficient.

Indeed, we have to `and` two radix trees; this computation is made recursively, checking the matching nodes of the two trees simultaneously. But when one of the trees contains a node and the other tree does not contain the matching node, it is of course irrelevant to perform the and of the sons of this node.

4.3.3 Not

the `not` clause is the most difficult atomic clause to perform with radix trees.

Each relation's size (its number of records) is stored. So perform a `not` over a radix tree may be done as follow: (the goal is to perform `not` T with T a radix tree).

let us define a n-full radix tree (n-frt) as a tree designed to contain all the numbers from 0 to $n-1$.

Then to perform a `not`, one may go from a n-frt (where n is the number of records of the expansion relation the request is solved onto) and remove the nodes corresponding to T.

To remove a node, one may proceed by removing the node and removing recursively its father if it has any child left.

For instance, if the expansion relation has 13 records, the `not T` with `T` the following tree (see Figure 6)

is (see Figure 7)

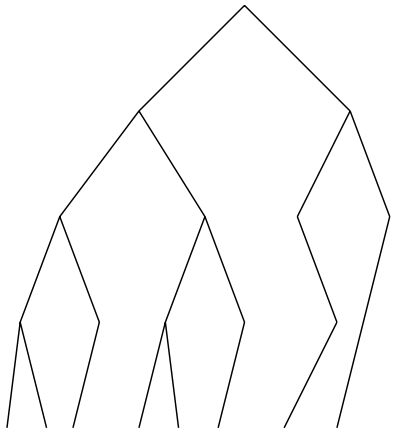

Figure 6: A 13-radix tree before a NOT operation

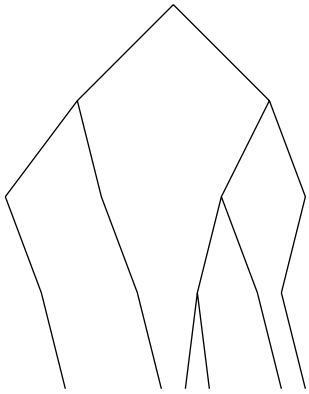

Figure 7: The same 13-radix tree after the NOT operation

4.4 Comparison between attributes

The comparison between two attributes is the most complex request to perform with this data representation and has a lot to do with the cartesian products section just below).

Let `t` be the expansion relation of the request and `c` and `d` be two of its attributes. A comparison between attributes may be a part of a where clause in which we discriminate the records such that for instance `t.c > t.d`. We empathize the fact that this comparison is done at the same record Ids (this is the difference with the cartesian product).

So how can we perform this clause?

Let $\mathcal{T}_c$ and $\mathcal{T}_d$ be the thesauruses of the attributes `t.c` and `t.d`. We are looking for the record Ids such that `t.c > t.d`. Here is how we may proceed. For each word w of the thesaurus $\mathcal{T}_c$, we can compute the radix tree r of the interval $[m_d, w']$ where w' is the greatest word of $\mathcal{T}_d$ lower than w. Then by performing an `and` over the w's radix tree and r, one obtains the corresponding record Ids for w.

By `Or`-ing the results of all the words of $\mathcal{T}_c$, one obtains the wanted radix tree.

Let us notice that the radix trees r (as above) are not to be computed independently one of the others: if the words w are read in an increasing order, one simply has to `Or` the radix trees corresponding to the intervals $[w_i, w_{i+1}[$.

The other such clauses may be performed in a similar way.

4.5 Cartesian products

The cartesian product is usually considered as a combinatoric computation over relation of a relational database. Actually, this computation may be performed quite simply and efficiently. The authors performed for instance a cartesian product of two relations (both of them of 6 millions records) in 7 seconds on an average computer.

Let us consider such a request: compute the number of times `t.c > t'.d` independently of the record Ids.

For instance, if the attributes `t.c` and `t'.d` are:

t.c
z
ab
z
c
da
e

and

t.d
b
a
as
sa
ca
ba
abra

then the number of times `t.c > t.d` is $7+1+7+5+6+6=32$. The complexity of the naive algorithm is a constant times the product of the two relations' size. It is not possible to actually perform cartesian products of two relations with modern machines in using this algorithm.

So how can we compute efficiently this result?

The attributes are stored with their thesaurus and the radix trees corresponding to each of the words of these thesauruses.

We may, to each word of the thesaurus, compute the number of records it appears at by a simple reading of the radix trees.

In the preceding example, this gives:

t.c	
ab	1
c	1
da	1
e	1
z	2

t.d	
a	1
abra	1
as	1
b	1
ba	1
ca	1
sa	1

One can also, for the attribute `t'.d` compute, for each word the number of words lower than or equal to it. This gives:

t.d, cumulated cardinalities	
a	1
abra	2
as	3
b	4
ba	5
ca	6
sa	7

Then, by reading the thesauruses and the corresponding numbers, one can compute the result. For each word `w` of `t.c`, one has to look for the greatest word `w'` of the thesaurus of `t'.d` lower than `w` and add to the result the product of the number of occurrences of `w` multiplied by the cumulated number of occurrences of `w'`.

This gives:

w	w'	w-card	w'-cumul. card.	product	partial result
ab	a	1	1	1	1
c	ba	1	5	5	6
d	ca	1	6	6	12
e	ca	1	6	6	18
z	sa	2	7	14	32

This algorithm's complexity is a constant time the sum of the sizes of the thesauruses, which is usually much less than the product of the relation's number of records even if the relations do not contain twice any word (a sum of thesauruses sizes is to be compared to the product of relations sizes...)

4.6 Correlated sub queries

This subsection and the following are dedicated to sub-queries. Indeed, the where clause may contain other where clauses and these sub queries may be or not correlated to the principal one.

What is a correlated sub query? An example of such request is the request 17 of the TPC. This request is:

```
select
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    lineitem
    part
where
    p_partkey = l_partkey
    and p_brand = '[BRAND]'
```

```
    and p_container = '[CONTAINER]'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem
        where
            p_partkey = p_partkey
    );
```

In this request, one has to perform the computation of the sub query in taking into account the condition requested in the principal part of the query (because the `p_partkey` of the sub-query belongs to the principal part of the request).

So this kind of requests may be rewritten in order to have to perform a non correlated sub query. The preceding query would thus become:

```
select
    sum(l_extendedprice) / 7.0 as ag_yearly
from
    lineitem
    part
where
    p_partkey = l_partkey
    and p_brand = '[BRAND]'
    and p_container = '[CONTAINER]'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem
            partsupp
        where
```

```
            p_partkey = p_partkey
            and p_brand = '[BRAND]'
            and p_container = '[CONTAINER]'
    );
```

So a correlated sub query may be rewritten in a not correlated sub query. This is the subject of the next sub section.

4.7 General sub queries

Now a SQL request containing not correlated sub queries may be treated simply: each sub query not containing any sub query is treated as a request by itself and the result of the computation takes the pace of the full sub request.

4.8 Perform computations on the found records

Now when dealing with a database, we are able to perform computations of record Ids of the expansion relation (matching the request) according to the `where` clause.

Now let us suppose that the goal of the request is to perform computations over some attributes of the relation but only for the found record Ids. For instance, it may be to compute an average tuple like in the preceding example.

The tuples of any attribute of an expansion relation are stored in the order they appear in it. So it is easy to read this file only for the record Ids matching the first part of the SQL request and perform the requested computation.

5 Managing the database

Now we are able to store a whole database and to perform efficiently SQL requests onto it. Usually, the quickest the SQL requests are performed, the slowest the database is managed.

This is not true in this case: not only are the requests performed fast but the management of the database are also fast (see the table of performances 1 in the introduction of this paper).

Why is that so? The indexes do not contain sorted data, expect the permutations linked to the thesauruses. In particular, there is no stored sorted data upon one or more attributes.

To manage a database, one may wish to add or remove records of a relation of a base, add or remove a primary key, add or remove a foreign key, add or remove a relation. We will see successively these items in the next sub sections.

5.1 Managing a relation

Manage a relation is the most common operation of the management of a database. Indeed, the most usual case is when the database manager wishes to add or remove records of a relation. All the other transformations change the database scheme or organization and these transformations are much more seldom.

Usually, when removing records of a relation we will refuse to reschedule the whole relation. This means that some record Ids will be declared free and the corresponding data will be removed from the expansion relations (we will see how below). So a relation will usually have "holes' ' : some record Ids will not be considered as filled by anything. These record Ids shall be stored in a file, containing firstly these record Ids and lastly the first index from which all the record Ids are free.

5.1.1 Adding records to a relation

So we wish to add records to a relation. Let us keep in mind that for us, all the relations of the database are expansion relations.

By reading the files of the free record Ids of this relation, we may assign a record Id to each of these records.

So we complete the records by filling the attributes of the relations that may be expanded in this relation (for instance if an attribute is a foreign key, we read the corresponding data in the corresponding relation).

Then we compute the thesaurus and the radix trees of the records to add to the relation and perform "Or" to the thesauruses and the radix trees of each attribute of the relation.

5.1.2 Removing records to a relation

We have here number of problems to solve: all the relations of our database are expansion relations and we do not want to reschedule the whole relation after to have removed some records of it. (If we did so, we would have to rewrite all the thesauruses, all the radix trees of this relation and of all the relations the first one may be expanded in.)

So we have the Ids of the records of the relation `T` to be removed (these record Ids may have found thanks to a `where` clause). So for each attribute `c` of the expansion relation,

and for all the words w of the thesaurus of these records, we build the radix trees that we x-or with the radix tree of w of c.

and then we just store the resulting radix tree in place of the preceding one.

We assume here that we do not have to change anything to the expansion relations in which T was expanded. Indeed, if we remove a record expanded in another relation, we should in this case throw an exception because the delete instruction was illegal.

5.1.3 Adding an attribute

Add an attribute c to an expansion relation consists in several operations. We have indeed to treat the relation T the attribute to be added belongs to and the relations in which T is expanded.

The treatment of T consists in building the thesaurus of c, the radix trees of each word of it and to store this whole stuff.

The treatment of each relation T′ in which T is expanded consists in reading the record Ids of T′ that must be added to T. Then one computes the thesaurus, the radix tree of each word of it and store the whole thing.

Let us take an example.

Related relations

T0		
(li)	c1	fk1
(0)	a	2
(1)	b	1
(2)	c	0
(3)	b	1
(4)	e	2

T1			
(li)	pk1	c2	fk2
(0)	0	S	0
(1)	1	T	1
(2)	2	V	0

T2		
(li)	pk2	c3
(0)	0	X
(1)	1	Y

The corresponding Expansion relations are thus:

Expanded T0							
(T0)			(T1)			(T2)	
(li)	c1	fk1	pk1	c2	fk2	pk2	c3
(0)	a	2	2	V	0	0	X
(1)	b	1	1	T	1	1	Y
(2)	c	0	0	S	0	0	X
(3)	b	1	1	T	1	1	Y
(4)	e	2	2	V	0	0	X

Expanded T1					
(T1)			(T2)		
(li)	pk1	c2	fk2	pk2	c3
(0)	0	S	0	0	X
(1)	1	T	1	1	Y
(2)	2	V	0	0	X

Expanded T2		
(li)	pk2	c3
(0)	0	X
(1)	1	Y

and let us suppose we wish to add an attribute c2 to T2, whose tuples are Y and Z.

The (expanded) relation T2 becomes

Expanded T2			
(li)	pk2	c3	c2
(0)	0	X	Y
(1)	1	Y	Z

To compute the new expanded relation T1, one reads the tuples of the primary key pk2 and copy the matching tuples of T2 in the new attribute of T1. This gives:

T1						
(T1)				(T2)		
(li)	pk1	c2	fk2	pk2	c3	c2
(0)	0	S	0	0	X	Y
(1)	1	T	1	1	Y	Z
(2)	2	V	0	0	X	Y

in a similar manner, one reads the tuples of pk2 in t0 to compute the new expanded relation T0. This gives:

T0								
(T0)			(T1)			(T2)		
(li)	c1	fk1	pk1	c2	fk2	pk2	c3	c2
(0)	a	2	2	V	0	0	X	Y
(1)	b	1	1	T	1	1	Y	Z
(2)	c	0	0	S	0	0	X	Y
(3)	b	1	1	T	1	1	Y	Z
(4)	e	2	2	V	0	0	X	Y

5.1.4 Removing an attribute

Remove an attribute is a simple operation. It only consists in erasing the file corresponding to this attribute for the relation `T` it belongs to and in all the relations `T'` in which `T` is expanded.

5.1.5 Adding a Primary Key

A primary key is stored by the data of the involved attributes and a permutation storing the order of the data of the primary key.

Add a primary key is thus simply to store the data of the involved attributes and the corresponding permutation.

5.1.6 Adding a Foreign Key

Adding a foreign key is quite more complicated.

A foreign key fk, belonging to an expansion relation T is hooked to a primary key pk, belonging to an expansion relation T′. The relation T′ must be expanded into T according to the couple (foreign key, primary key) being treated even if this relation is already expanded into T by the mean of another foreign key.

For each record of T, of index i, the foreign key has a tuple v and we can find the record Id $p(i)$ of T′ where $pk = v$.

Then we add al the attributes of T′ in T. To perform this, for each attribute C of T′ we read the tuple $T'.c[p[i]]$ for all integers i. Then we do as usually by building the thesaurus of this attribute and for each word of this thesaurus the matching radix tree.

5.1.7 Removing a Primary Key

Remove a primary key consists in deleting the corresponding files. Of course, if this primary key is the target of a foreign key, we should throw an exception because such an instruction should be illegal.

5.1.8 Removing a Foreign Key

Let us denote fk the foreign key to be removed and T the relation it belongs to. This foreign key targets a primary key, pk, belonging to a relation T′.

To remove a foreign key breaks the link between two relations. This means that T′ is no longer expanded in T and in none of the expansions of T.

5.2 Managing the base

Manage the database itself consists in adding or removing a whole relation...

5.2.1 Adding or removing a relation

Adding (removing) a relation consists in adding (removing) all its attributes, all its primary keys and and all its foreign keys. All these algorithms have been explained above.

6 Conclusion

One of the problems of the use of bitmaps is the size of the involved vectors and the fact that usually many of the bits are equal to 0. In this paper we exposed a database manager algorithm. It may be used to fully manage a database with performances showed in the table 1. The use of radix trees seems to be an interesting hierarchization of bitmaps. They show the advantages to make possible an affordable storage of bitmaps, only the parts with 1s are stored. They also allow a computation level by level, which gives good performances in particular ti solve "and" requests.

This algorithm is also parallelizable and a possible future work is to implement a parallel version of the A-algorithm (this work is in progress, in cooperation with Christophe Cérin). This is to be compared to performances obtained with parallel B-trees like in [7], for instance

In order to keep the efficiency of the A-algorithm, one has to pre-compute the join clauses. The use of macro-words makes also faster the resolution of “between” clauses.

The author would like to apply these ideas to related problems, like find all the occurrences of a pattern in an image whatever the foreground would be, or to find a sound in some sounds whatever the noise would be.

7 Acknowledgment

The author would like to thank warmly Dominique Bernardi for numerous and instructive discussions, Christophe Cérin for numerous comments on an earlier version of this paper and give a special thanks to Maude Manouvrier for very interesting and instructive conversations and comments on an earlier version of this paper.

References

[1] Raghu Ramakrishnan and Johannes Geheke, Database Management Systems

[2] Abraham Silberschatz, Henry F. Korth, S. Sudarshan, *Database System Concepts*, McGraw-Hill ISBN 0-07-228363-7.

[3] R. Bayer and E. McCreight, *Organization of Large Ordered Indexes,* Acta Iform. 1 (1972), 173 189

[4] Gaston H. Gonnet, Ricardo Baeza-Yates.*Handbook of Algorithms and Data Structures*,`http://www.dcc.uchile.cl/~rbaeza/handbook/`

[5] Chris Jernaine, Anindya Datta, Edward Omiecinski. (1999) *A Novel Index Supporting High Volume Data Warahouse Inbsertion.* VLDB Conf. 235-246.

[6] `http://www.tpc.org`

[7] Shogo Ogura, Takao Mura, *Paging B-trees for Distributed Environments*

[8] D. Lomet, *B-tree Page Size When Caching is Considered*, SIGMOD Record (ACM Special Interest Group on Management of Data), 27 (1998), 3, p 28

[9] O'Neill, P. and O'Neil E. *Databases: principles, programming and performance*. 2nd ed., Morgan Kaufman Publishers, San Francisco, CA., 2000.

[10] Tadeusz Morzy, Maciez Zakrzewicz, *Group Bitmap Index: A Structure For Rules Associations Retrieval*, American Association For Artificial Intelligence, 1998.