A RELIABLE PARALLEL BACKEND USING MULTIATTRIBUTE CLUSTERING AND SELECT-JOIN OPERATOR

Jean-Pierre CHEINEY*, Pascal FAUDEMAY**,
Rodolphe MICHEL**, Jean-Marc THEVENIN**

SABRE Project
INRIA , BP 105, 78153 Le Chesnay-Cedex France

Abstract

An access method based upon multi-attribute clustering allows the database administrator to define multiply partitioned relations. For each attribute in the clustering, we can consider the set of subrelations as a relation view. Such a method has been implemented in SABRE. It relies on multi-attribute digital hashing and a linearly growing directory. Using this method, we show that it is possible to improve the multiprocessor hashing join algorithms by a ratio of 3 to 5, with the same hardware configuration. According to our evaluation, the memory requirements are approximately the same as with the hashing algorithms, and the common bus used for disk accesses does not saturate. Any configuration can be linearly extended by adding or removing a disk or a processor, and reliability is guaranteed by a simple management of multiple copies. In case of a disk breakdown, the continuation of operation is possible with minimum loss of speed.

1. Introduction.

The join is one of the "non linear" relational algebra operations; their duration grows with the volume of data, but more rapidly. Beyond a certain volume of data, present algorithms are too slow. It is therefore useful to build fast parallel algorithms for this operation. In order to obtain sufficiently rapid algorithms, solutions have been proposed which are linear up to a large volume of data. Linearization is obtained by a large increase of the logarithm base in the complexity formula. The execution time may be divided by distributing the I/O and processing load among several processors and disks. Such solutions have been proposed, based upon hashing algorithms. In this paper, we present a faster parallel solution, based upon multi-attribute clustering.

A second goal in a multiprocessor system is to keep an access to all data and good performances in crippled mode, with one processor or disk down. We present a solution based upon data duplication, and access of disks through a common bus. During insertions, tuples are sent to processors together with a logical address, which is recognized by the processors which must insert the tuple or its copy. During reads, the search patterns are modified in case of a disk breakdown so that the processor would access the copy in place of the faulty disk. The multi-attribute clustering allows partitioning of each relation according to the query; each subrelation has an index i, and is read by processor (i mod p), where p is the number of processors. In case of a processor breakdown, the mapping between processors and disks is changed by a simple change of the value of p.

We first present some previous results concerning hashing join algorithms and multi-attribute clustering.

To "linearize" the joins, the solution presented by [KITS83],[DEWI84] is to hash both relations R and S which must be joined, upon the join attribute Aj. This hashing gives two sets of subrelations {Ri} and {Si}, where the tuples of the subrelation Ri have a hashing value i with respect to the join attribute. We shall say that this hashing gives the partitioned views $\{Ri\}_{Aj}$ and $\{Si\}_{Aj}$ of relations R and S. Any pair of tuples from R and S which joins on this attribute belongs to subrelations having the same index. The R |X| S join is then reduced to the union of smaller joins, where at least one of the source relations holds in memory: R |X| S = U_i Ri |X| Si. The hashing phase must be completed before any join operation, it needs two read-writes of R and S. The join cannot be done in pipe-line with hashing. It needs a third read-write of R and S from the disks. This algorithm can be executed in three read-writes of each source relation with some conditions concerning the cache memory size, which are recalled in par. 2.2.

An improvement of this solution is to use a multi-attribute clustering of each relation on the join and selection attributes [CHEI86], using predicate trees [GARD84], [VALD84b], [GARD86]. The hashing value of each attribute is represented by a bit string. Each hashing function is called a *level* of the multi-attribute clustering. The corresponding bit strings are concatenated to form the tuple *signature*, which is a multi-attribute hashing value. Tuples are clustered on disk by digital hashing on the signature. Tuples on the same page have a same *signature prefix*, which indexes a physical address in a directory. Tuples with the same hashing value for one attribute, and possibly corresponding to some selection criteria, are characterized by one or several *signature profiles*, where some bits have an unknown value ("•"= "0" or "1"). To access these tuples, the program filters the directory with the list of profiles. In the "predicate trees" method , the directory volume is proportional to the data volume; this property is important in this approach. The result of the directory filtering with a profiles list is a set of subrelations {Ri}. When filtering with several profiles lists, we get several partitioned views $\{Ri\}_{Aj}$ according to several join or selection attributes Aj. Annex A shows an example of clustering using the predicate trees.

For a join on the attribute Aj, we use the views $\{Ri\}_{Aj}$ and $\{Si\}_{Aj}$. When there is also a selection, we use in fact an intersection between each Ri and Si and a list of subrelations from $\{Ri\}_{As}$ and $\{Si\}_{As}$, where As is a selection attribute. Its pages are obtained by filtering the directory, using the appropriate profiles. The join is done in one read of R and S, which corresponds to the third read-write phase of the hashing algorithm. The selection is

* Institut National des Télécommunications
** INRIA, MASI Laboratory (CNRS/ University Paris VI)

Proceedings of the Twelfth International Conference on Very Large Data Bases **Kyoto, August, 1986**

done in parallel with this read phase. Unlike the previous algorithm, this selection can be pipe-lined with the join.

The parallel implementation of the hashing and clustering join algorithms is simple. For the hashing algorithm this has been presented in [DEWI85]. Other solutions are proposed in [GOOD81], [VALD84], [OZKA85]. In the parallel algorithm by hashing, let us consider p processors and a hashing into h buckets. Relations R and S are divided into p fragments of approximately the same size, and each one is hashed by one of the processors into h buckets. This phase is fully parallel, unlike multiprocessor sorting used for some join algorithms. The result pages are transfered on disks corresponding to the bucket number (after some coding of this number), through the common bus. This phase is the only one which is not parallel. The joins of the {Ri, Si} pairs are then uniformly distributed among the p processors. The re-reading of the Ri and Si is done on the common bus or on local busses according to the hardware architecture.

In this paper, we present a parallel join algorithm by clustering. With this approach, the only read and operation phase is the third one. As the data may be accessed through the common bus, each processor with an index (i mod p) reads the pair of subrelations Ri and Si and joins them. This transfer on the common bus is symetrical to that of the data re-writing in the hashing algorithm. We show that this does not cause any bus saturation either. Some complementary calculus is needed when {Ri} or {Si} is an incomplete partition of R or S, i.e. when the hashing value of some tuples regarding the join attribute is not given by the clustering. Pages corresponding to these tuples are given by the directory filtering, their signature prefix does not contain all the bits concerning the clustering attribute.

The proposed algorithms lead to a parallel reliable architecture, which we call "reliable multiple backend" , according to [HSIA85]. This architecture is presently being implemented in the Sabre.3 version of the Sabre database machine, which includes three levels of software: an "interface machine", which analyses queries, an "assertional machine" (ASSM) which breaks down the queries into a sequence of relational operations, the "algebraic machine" (ALGM) which executes the operations of this algebra. The algebraic machine is here replicated into one sample per available processor, each one processing a subset of the database; the subset varies according to the query. This solution is built upon a hardware architecture where all processors can access all disks through a common bus. This architecture is presented in figure 1.

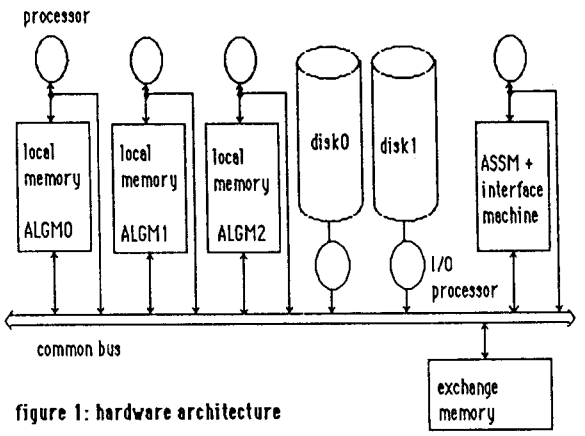


figure 1: hardware architecture

In our proposal, the only data which use the common bus are commands, results and I/O page transfers. We show that the bus is not saturated. The "reliable multiple backend" may also be implemented with disks connected to local busses; additional replications of some relations are then needed; this option will be presented in a later paper.In par. 2.1., we present the algorithms for the parallel join by clustering. In par. 2.2., we give its feasability conditions, from the points of view of the cache memory and relation sizes, and of the common bus throughput. In par. 3.1., we propose a simple adaptation to the growth and decrease (in case of breakdown) of the number of processors, and we examine its efficiency conditions. In par. 3.2., we describe the adaptation of algorithms to reliability targets. Par. 4. concludes.

2. Parallel selection-joins: algorithms and evaluation.

2.1. Parallel algorithms using predicate trees and partitioned views.

In the introduction we have presented the notion of predicate trees and of a partitioned view defined on a relation, clustered by predicate trees; this is the basis of the algorithms described in this paragraph. We first recall this definition, then give indications on the physical distribution of data among the disks, and present the algorithms themselves. These algorithms reduce the I/O time from 3 read-writes of both relations (with the hashing joins algorithms) to 1 read.

2.1.1. Partitioned views of a relation.

The clustering of a relation R using directory K allows definition of several partitioned views of K and R, according to attributes Aj1..Ajm. There is a partitioned view according to attribute Aj when a level of the multi-attribute clustering is defined on attribute Aj. A particular physical page of R (or an entry of K) may belong to several *subrelations* Ri_{Aj}, Ri'_{Aj}, Ri''_{Aj}, etc.. (or Ki_{Aj}, Ki'_{Aj}, etc..). Each page prefix is the concatenation of bit strings representing the hashing values $H(t)_{Aj}$ of tuples t of the same page, according to attributes Aj1, ..Ajm. We suppose that the hashing function $H(t)_{Aj}$ has h values. Each page of R belongs either to a subrelation Ri, i=1..h, when its hashing value i=H(t) according to Aj is defined in the page prefix, or to a subrelation R0 which we call the *residue*, when its hashing value is not defined.

More formally, we say that {Ri} is a partitioned view of R according to attribute Aj (written $\{Ri\}_{Aj}$), if the following conditions are true

- {Ri} is a partition of R; any tuple of R belongs to a subrelation Ri of $\{Ri\}_{Aj}$ and only one.
- let $H(t)_{Aj}$ be the uniform hashing function defined by the multiattribute clustering on attribute Aj, and h be the number of possible values of H ($H(t) \leq h$). $H(t)_{Aj}$ is constant within subrelation Ri, for $i \neq 0$. For any possible value of H(t) ($0 < H(t) \leq h$) there is a subrelation Ri with an index value i=H(t).
- Any tuple t of R either belongs to a subrelation Ri, $0 < i \leq h$, or to the subrelation R0.

There may be tuples with hashing value i in R0 and tuples with the same hashing value i in Ri. We call R0 the residue of the relation. If there is at least one tuple in R0, then we shall say that $\{Ri\}_{Aj}$ is *incomplete*.

For each relation, part of the directory describes the mapping between the prefix of each page of the relation and its physical address (usually on disk). The prefix is said to be "developed" on attribute Aj for which there is a hashing function H_{Aj} with 2^k values, if the prefix has k bits relative to this attribute. The index i of subrelation Ri_{Aj} is the integer defined by these k bits. A tuple belongs to the residue R0 of the partition defined on Aj ($R0_{Aj}$) if its signature has less than k bits defined on attribute Aj. The partitioning of R on Aj may thus be done by filtering the directory, in one read of it. A partition of R may be associated with a partition of the directory, $\{Ki\}_{Aj}$, where each subset of the directory holds the prefix and adresses of pages of the same subrelation.

The proposed algorithms use a uniform partitioning of the relations which must be joined, according to the join attribute.The first levels of the clustering are uniformizing hashing functions, such as folding or modulo, defined on the join attributes. Each processor makes the selection of one or several pairs of subrelations Ri and Si, and joins the selected tuples of each Ri with those of the corresponding subrelation Si. To find the pages which contain the tuples of a subrelation Ri of $\{Ri\}_{Aj}$, the ALGM filters the directory with the following signature profile: all bits of the profile are positioned at "•" ("0 or 1"), except those which correspond to the join attribute Aj and are forced at the hashing value which defines subrelation Ri.

2.1.2. Distribution of tuples among the disks.

In Sabre.3, the ALGM level is replicated on p processors. Each ALGM processes a subset of the database. This subset may vary according to the operation. The different subsets are subrelations of one of the partitioned views defined in the last paragraph. Sabre.3 also relies upon the fact that we may possibly use several different disks, each being connected to the common bus in order for each processor to access it directly. To manage the distribution of data among the disks, a logical disk is associated to each ALGM for insertions, modifications and linear queries (selections and projection without suppressing doubles). Several logical disks may be situated on a single physical disk. However, to simplify the presentation of the degraded mode we shall suppose that there is one physical disk per ALGM.

The distribution among disks is done according to the first level of the predicate tree (each level describes a hashing function, beginning by the one which corresponds to the first signature bits). For this distribution, we apply a modulo function p on the hashing value of the first clustering attribute. Two possibilities may be considered. First the join may be done upon the distribution attribute, then all relations Ri and Si read by one processor are on the index i disk. As each processor addresses a different disk, the parallelism between I/O done by different processors is guaranteed. Secondly the join may be done on another attribute. In this case the distribution of each subrelation Ri among the disks is usually uniform. This also guarantees the parallelism between I/O. However the tasks of the different processors may be serialized by I/O coming from several disks to a same processor. To avoid this, it is useful that the disk managers should serve the different processors in a circular way. This point is developed in the next paragraph.

2.1.3. Selection-join algorithms.

Selection and join are done in pipe-line, and may be seen as one operation, the selection-join. Up to a certain point, the case of several levels of joins in a query may also be treated with our algorithms, but will not be considered in this paper. A first phase of the selection-join is represented by the directory partitioning, which is used both for selection and join. Contrary to the hashing approach, the I/O for this phase are limited to one read of the directory. A second phase is represented by the select-join operation, done in one read.

During the first phase, each processor accesses the whole directory -at least the part relative to the relations used in the following operations. Part of the directory is associated to a logical disk, which usually holds part of each relation. However all directories can be accessed through the common bus. They can be considered globally as one common directory. Using this directory, each processor calculates the directory partitioning (and in that way, the relation partitioning); each subdirectory is made of the addresses of pages having the same hashing value regarding the join attribute, and of a signature prefix corresponding to the subrelation. The subrelations are practically restricted to pages whose signature bits for the selection attributes belong to a profiles list which results from the selection expression. This allows a preselection at the directory level during the partitioning. Each ALGM calculates the subrelations which it must process by applying a modulo p function to the hashing values defined by the partitioned view. A subrelation must be processed when the result is equal to the number of the ALGM. Simplifications are possible when the data useful for the processor are known to be on one logical disk, which should happen frequently. The directory part accessed by each processor is then limited to this logical disk. It is also possible that the join attribute should be a selection attribute. In this case a preselection on this attribute is possible if there is both a folding or modulo clustering on it (a uniform function) and a secondary clustering by intervals (non uniform but order preserving function).

We call R the smallest of the two relations (when considering the probable size after selection), and S the other one. For each i processor and each hashing value j on the join attribute, the selection is first made on relation Rij. For relation Sij, the selection is made page after page; after each page selection we do the join of the result with Rij. The join algorithm must be hashing or index building , which are

practically linear in the number of pages of S, after the index has been built. A cartesian product algorithm would be in O(||R||.||S||), where these two quantities are the cardinalities of R and S; a sort join would not allow the join before a complete reading of Rij and Sij. This simplified algorithm is presented below.

```
On each processor Pi (i=1..p) do
begin
    partition_directory (R, K, KPAR.R)
    partition_directory (S, K, KPAR.S)
    {K is the directory, KPAR is the partitioned
    directory}
    for j=1 to h do
    begin
        SEL (Rij)
        while not end of Sij do
        begin
            SELPAGE (Sij)
            JOIN (Rij, Sij)
        end
    end
end
```

figure 2: selection-join algorithm, simplified version.

We shall now present a more complete version of this algorithm. Indeed the last version does not describe the management of the subrelation "residue", which can be done by hashing or copying according to the subrelation size. Transfers may be directly managed by the "producers" processors after calculating R0 and S0, or directly by the "consumer" processors when selecting data ("producers" and "consumers" processors are of course the same but considered at different times). The order of selections may also be considered.

A first choice is the processing of the "residue" relations R0 and S0. These relations may be empty, in which case the algorithm goes on as before. If they contain only a few tuples, or at least if each subrelation Ri plus R0 fits in memory, it is possible to add R0 to each other subrelation without any other calculus. This solution may be a good one if the transfer time of R0 to all processors is small, and if the calculation time is little increased, e.g. if:

$$|R0| < F.|R|/p^2$$

where F is the uniformity ratio of hashing (size ratio between the largest bucket and the mean one).

This solution will only slow down the machine by a ratio of less than 1/p, which seems to be a good limit in a machine with p processors. If these conditions are not fulfilled, the splitting of R0 is forced in as many pages as the number of values of the hashing function. These page prefixes are added to the partitioned directory with an address value showing that it is a page in memory in processor i (if there is no overflow of R0 on disk). The logical processors which manage the I/O may then read these pages in the corresponding local memories. A transfer done by the source ALGM processor at the end of the hashing is also possible; the partitioned directory will then reference the after-transfer addresses. Subrelation S0 is treated like R0.

Another aspect of the selection-join algorithm is the intertwinning of selections and joins, and possibly the sequencing of these two operations in order to avoid their serialization. Selections on the argument relations of the join are pipe-lined with it. Thus, when retrieving the pages of a subrelation, the ALGM makes an intersection between the signature profile that characterizes the subrelation and the selection profile, before filtering the directory. The ALGM then performs a partial selection on the pages subset which it has obtained, just before doing the local join with the result tuples of the selection.

A partial selection is materialized by a set of selections sent in parallel to each disk. When all ALGM access the same disk at the same time, the ALGM work is serialized by the disk. Much of the parallelism between the ALGM is then lost. To avoid this situation, the following rule may be used: each processor first reads its disk, then the other ones by rotation on the disk number. This insures a good parallelism between each disk and each ALGM operation. The intertwinning of partial selections and of joins also avoids the writing and re-reading on disk to do the join. The selection-join algorithm is finally the following:

```
On each processor i (i=1..p) do
begin
    calculate_profiles (Aj, Q)
    {Q is the selection expression, <Aj> the join attributes
    list}
    partition _directory (R, K; KPAR.R)
    partition_directory (S, K; KPAR.S)
    If R0 not empty then do
    begin
        if (|R0| + F.|R|/h ≤ |M|-2) and
        (|R0| ≤ |R|/p²) then
        { |M| is the cache memory size in pages }
        RESU_R=R0
        { RESU holds the pages of R0 after their splitting.
        It is an input of SEL like KPAR. For
        simplicity we omit those two inputs in SEL
        occurrences.}
        else Hash (R0, RESU.R, KPAR.R)
        {the transfer of RESU pages in destination
        processors is asked by these ones in SEL}
    end
    If S0 not empty then do
    begin
        if |S0| ≤ |S|/p² then do RESU_S= S0
        else Hash (S0, RESU.S, KPAR.S)
    end
    for j=1 to h do
    begin
        k=i
        repeat SEL_k (Sij); k= (k+1) mod p
        until k=i
        repeat SEL_k (Rij); k= (k+1) mod p
        until k=i
        JOIN (Rij, Sij)
    end
end
```

figure 3 : selection-join algorithm, detailed version

2.2. Evaluations.

2.2.1. evaluations of the hashing algorithm.

During external hashing (with a re-writing of relations on disk) of relations R and S, the maximum number of hashing buckets is |M|-1, where |M| is the cache memory size in pages. In order for the smaller relation to stay in memory during the join, its size must be under |M|-2 pages (one page is kept for the other relation and another for the result). The relation size is therefore:

$$|R| \leq (|M|-1)(|M|-2)/F, \quad \textbf{(condition 1)}$$

that is about $|R| \leq |M|^2/F$; F is the uniformity ratio of the hashing function (ratio between the size of the largest of the subrelations and their mean size).

With a multiprocessor hashing, the same condition must be true, |M| being the local memory size. If |B| is the global size of memory in pages (|B|= p.|M|), then:

$$|R| \leq |B|^2/p^2, \qquad \textbf{(condition 2)}$$

which gives $|B| \geq p\sqrt{|R|}$; in some cases, this may imply that |B| > |R|. The conditions regarding the cache memory size are more constraining when the number of processors is larger; the same observation is made later with the "multiple back-end" architecture.

If condition 2 is not true, it is necessary to do a first hashing in p buckets, followed by a second hashing in n= F.|R|/|B| buckets. This re-hashing implies two additional read-writes of R and S. To avoid this, it is possible to pipe-line the distribution and the second hashing. It is then necessary to have (p+1) more pages on each processor for re-hashing, and condition 1 becomes:

$$|R| \leq p(|M|-2)(|M|-1-(p+1))/F^2 \qquad \textbf{(condition 3)}$$

If this condition is not fulfilled, 5 read-writes of R and S become inevitable. In the "multiple backend" architecture, these conditions apply to the size of subrelation R0, which contains the non-partitioned tuples, in case of an incomplete partitioning.

2.2.2. Case of the "multiple backend".

The efficiency of the "multiple backend" algorithms, as a function of the cache memory size and of the number of processors, depends on three factors: the possible overflow of the cache memory, the minimum size of the smaller relation, the possible saturation of the common bus.

problem of the cache memory overflow.

Let us suppose that |R| ≤ |S|. In order to make the join on attribute Aj in one read of R and S, each Ri in $\{Ri\}_{Aj}$ must fit in the cache memory. We suppose that the clustering is done on m attributes, each one having the same weight (same number of hashing bits in the signature). The number of signature bits is then log |R|, the number of hashing bits on the join attribute is (log |R|)/m, and the cardinality of {Ri} is $2^{(\log |R|)/m} = |R|^{1/m}$. The size of a subrelation Ri is then $|R|/|R|^{1/m} = |R|^{(m-1)/m}$. Condition 1 becomes:

$$|R| \leq (|M|-2)^{m/(m-1)}. \qquad \textbf{(condition 4)}$$

As before, the same condition is also true in the multi-processor. The number of clustering attributes with the same weight is typically of 2 to 3. The case where m=2 is that of a relation with one key and a frequent selection attribute (there can also be several selection attributes, each with a smaller weight), or a relation with two keys and no selection attribute (association relation). With some adequate normalisation, it is always possible to be in that case. When m=2, *the memory size constraints are practically identical to those of the parallel hashing algorithm.* If m=3, reasonable parameters for a multiprocessor architecture (b=16 k.bytes, where b is the page size, |M|=400 with eg: p=10) allow a relation size of relation R of 128 M.bytes; with usual tuples, and a load factor of 0.7, this size corresponds to a smallest relation of up to 700,000 tuples. The size of R may be k times larger, if a selection of selectivity s < 1/k is done before the join.

problem of the small relations

For small relations, some pages may have an insufficient number of signature bits on the join attribute to determine their subrelation. We speak of an "incomplete partitioning"; the corresponding pages must be distributed among the subrelations by hashing. If the clustering tree is balanced, which is an approximation, the number of bits of prefixes is (log |R|). The partition can be complete if:

$$\log |R| \geq 2 \lceil \log p \rceil \qquad \textbf{(condition 5)}$$

where [x] is the smallest integer superior or equal to x. For p=16, this will be the case if log |R| ≥ 8, i.e. if |R| ≥ 256 pages. With optimal pages of one track (about 16 k.bytes), this corresponds to a subrelation of 4 M.bytes, which is already a large relation.

However with usual relation sizes, most pages have a longer signature than the theoretical value. In experiments which we have done in Sabre.2 with "predicate trees" clustering [GARD86], most pages of the 2 M.bytes relations had 12 to 13 bits in their signature prefix,so that these pages would not have needed any hashing. The number of pages which must be hashed in order to obtain a complete partitioning should be very small, so that they would almost always fit in cache memory. It follows that this phenomenon should not increase the number of I/O operations.

Conditions for the saturation of the global bus

In the architecture presented here, the reading of pages from each subrelation is done through a common bus. It's possible saturation could limit the useful number of disks and processors. A first approach is to suppose that all disks are permanently read. As there are no other data on the bus, except from the result tuples, the bus is saturated when V= Dq, where V is the bus throughput, q the number of disks, and D is the disk throughput. With usual disks having a maximal throughput of 1.8 M.bytes, like the Fujitsu Eagle, and a common bus throughput of 10 M.bytes per second, saturation would appear with more than 5 disks. The useful number of processors would then be the number needed to do joins on the corresponding volume of data, at the same time.

An other approach is to suppose that the data transfer is done in a cyclic way, first towards processor 1, then towards processor 2, ..., then again towards processor 1, and so on - each processor receiving at each time one subrelation of R and S. The bus will then be saturated if the time needed to transfer a subrelation from R and S to each processor is larger than the time needed by the slower processor to do the join, i.e. if:

$$rl/V + r'l/V \geq J(Fr/p, Fr'/p) + K(r) + K(r') \quad \textbf{(condition 6)}$$

where r and r' are the cardinalities of R and S, l is the mean

length of a tuple, V the throughput of the bus, J(r, r') the time to join two relations with cardinalities r and r', K(r) the time to filter the directory of a relation of cardinality r. K is small compared to J (approximately K(r)= a.|R|, with a near from 3µs). The saturation is then a function of the other factors. We shall suppose that r=r', and J(r, r')= 2kr. The bus is then saturated if:

$$2\ rl/V \geq 2\ kFr/p\ ,\ \text{i.e.: } V \leq pl/kF \quad \textbf{(condition 7)}$$

where F is the uniformity ratio of hashing.
With l=100, F=1.2, r=100000, V=10 M.bytes/sec or V= 256 M.bytes/sec, the array gives the values of p and T (the join duration) at saturation:

k	V	p	T
10^{-3}	10	120	2s.
10^{-4}	10	12	2s.
10^{-4}	256	306	0.08s.
10^{-5}	256	30	0.08s.

V is the throughput of the common bus in megabytes per second. Current busses have a throughput of 10 Megabytes/sec. With high performance algorithms, higher throughputs may be necessary. The present technological limit seems to be represented by interconnection arrays of up to128x128 bits, and a 16 Mhz frequency; such arrays will have a throughput of 256 Megabytes/sec. Regarding the join duration, $k=10^{-2}$ corresponds to present values [BITT83], $k=10^{-3}$ corresponds to the expected duration of some micro-programmed algorithms with index building; $k=10^{-4}$ could correspond to future VLSI processors using the same algorithms; $k=10^{-5}$ corresponds to the optimistic evaluations of [DEWI84] for hashing algorithms in memory, or to the aims presented in [MOTO82] for the join by sorters. With present values of k, current busses are fast enough to avoid saturation, up to a high number of processors.

3. Growth and reliability.

3.1. Linear growth of the number of processors.

The way of adding new disks is compatible with the distribution mechanism, which uses the first signature bits. The mapping with a disk number is done by applying a modulo function to the k first bits, p being the number of disks. There is no change in the hashing functions of the PT nor any re-hashing of tuples.

A first constraint is that the addition of the p^{th} disk must cause an improvement with respect to a configuration with (p-1) disks. This implies that the proportion of the database which is stored on the disk with the heaviest load must decrease. A second constraint is to minimise the relative difference between the buckets number of the more loaded disk and of the less loaded one.

Due to space limitation we do not present the algorithm which gives the number of bits as a function of a number of processors and the accepted lack of balance. With up to 8 processors, the number of bits needed per join attribute is less than or equal to 6; a clustering on 2 attributes needs at most 12 bits of signature. With real distributions, even small relations of 2 M.bytes are nearly fully partitioned on the join attributes. Good performances of joins are compatible with any variation of the processors number from 1 to 8 at least.

3.2. An integrated reliability.

The proposed method allows a good integration of reliability and a simple approach of the problem by defining a replication of the database on several disks. Updated copies are available in case of crash and allow a degraded mode without interruption, and secondly if disks are coupled on local busses (which is probably not the best architecture), transfers between processors are limited; this aspect is not developed in this paper.

We separate the physical distribution of tuples, which is used for insertions, and the logical distribution which is used for queries and makes it possible to limit the search to some part of a relation. The data which are physically present on the disk managed by a disk will be a superset of one of its views.

A data replication can be considered under two conditions. First the storage cost must not be too expensive. The evolution in disk capacities is encouraging. Second, the increase in the search space must not imply an important increase in search times. From this point of view, the predicate trees access method is very interesting; the directory and its filtering time are proportional to the volume of data, which is not the case with other methods. A simple replication is done by using two copies. When updating, an insertion is made on two disks. If the disks are defined by the first k bits of the signature, the insertion on two disks may be defined by replacing the tuple signature with an insertion profile where the k^{th} bit is replaced by an "•" (unknown bit). The general insertion procedure is then:

1) calculate the tuple signature
2) build the insertion profile by setting at "•" the k^{th} bit of the signature.
3) according to the insertion profile, the insertion is then made on two disks

This mechanism makes it possible to duplicate the database while keeping a uniform distribution. Annex B1 presents an example of insertion. A replication in four copies (or any power n of 2) is as simple and uses 2 bits (or n) at "•" in the insertion profile; with the algorithms presented here, the number of disks must be superior or equal to the number of copies. All accesses are done by a query modification mechanism. When accessing a disk, the algebraic machine must ensure the duplicated data are not accessed. The following algorithm is then used:

1) generation of a list of k bit prefixes {Pi}, according to the physical distribution predicate i.e.:
Pi modulo P = disk number.
2) intersection between the signature profile which sum up the search criterium and the prefix list.

This is a simple positioning of the first k bits which are successively replaced by all the values which verify the physical distribution predicate. An example of search is given in annex B2. In case of a disk breakdown, all pages are duplicated on a different disk; a crippled mode is possible without any interruption. When inverting the k^{th} bit during the distribution, the result of the modulo function will always be different from the search distribution, as the k^{th} bit is the lowest weight bit taken into account.

3.3. Degraded mode.

In case of a processor breakdown, the system is reconfigured by renumbering each available processor with a number between 1 and p-1. Thus, the logical partitioning is re-defined on p-1 processors. The physical distribution is still done on q disks. If q=p, one of the processors will make insertions and modifications on two disks.

In case of a disk breakdown, the data no longer available on the crashed disk can be accessed from their copies read over the other disks. The selection algorithm in degraded mode is the following:

1) generation of the list of the k bits prefixes {Pi} corresponding to the crashed disk.
2) generation of the list of k bits prefixes {Pi} corresponding to the disk accessed.
3) intersection between the signature profile and the union of the two set of former lists.

With this algorithm, all subsets which must be accessed are read one time and only one time. In degraded mode the insertion is not modified but only one copy is updated. Annex B3 gives an example of a selection with disk 1 unavailable.

For the recovery, there is a very simple mode. During the restart of the faulty disk, its subsets are restored from copies located among the others disks. It is possible to continue the queries; only updates are stopped during the recovery. During this operation, the restarting disk is still considered unavailable, and queries are applied in degraded mode to the other disks. The normal mode can be used as soon as the copies are done.

The selection of the copies to re-write on the previously faulty disk is done by the degraded selection algorithm, but without normal search criteria. The union is applied to an empty profile, which means that the search profiles of the copies are not modified. The bits which do not take part in the distribution (i.e. from the $k+1^{th}$) are set at "•" in order to select full subsets.

In this architecture we have supposed that one disk unit is updated by one processor in a disk. This is realistic when there are few processors. However, if there are several tens of processors, there are disks which are common to several processors. In this case the reliability algorithms must insure the distribution of subsets and their copies on different disks. A parity element may then be added to the first k bits of the signature in order to guarantee that a subset and its copy will be written on a different disk.

4. Conclusion

We have presented an architecture and multiprocessor algorithms which speed up as well joins as selections, and allow to keep using the machine in case of a disk or processor breakdown. This architecture is easily extendible according to the volume of data or the performances needed. It gives an improvement by 3 to 5 versus the algorithms presented in [DEWI85], with the same configuration. The cache memory size requirements are identical to those of the hashing algorithms, with a clustering on up to 2 join attributes; some clustering on selection algorithms is also possible. Real relations can always be limited to this number of join attributes by adequate normalization. A number of processors varying from 10 to 150 can be used with even very fast internal join algorithms (i.e. joins in memory). This limit could be increased in another architecture with disks connected on local busses, but this would lead to a full replication of some relations; this possibility will be considered in a forthcoming paper. The improvement ratio versus mono-processor existing algorithms is currently a factor 3p/F, where F is the uniformity ratio of hashing; according to our simulations F is usually under 1.2, thus the multiprocessor efficiency may be above 0.8 .

Acknowledgments

The authors want to thank G. Gardarin for fruitfull comments. This work has been supported by INRIA, CNRS-Paris VI-MASI and the French Ministry of Research and Industry (PRC-BD3).

5. References

[BITT83] BITTON D, DEWITT D J, TURBYFILL C : "Benchmarking Database Systems: a Systematic Approach", Computer Sciences Technical Report N°256, 1983, University of Wisconsin-Madison.

[CHEI86] CHEINEY JP, FAUDEMAY P, MICHEL R : "An Extension of Access Paths to Improve Joins and Selections", Second Int. Conf. on Data Engineering, Los Angeles, 1986.

[DEWI84] DEWITT D J et al : "Implementation Techniques for Main Memory Database System". ACM SIGMOD , Boston, 1984.

[DEWI85] DEWITT D J , GERBER R : "Multiprocessor Hash-based Join Algorithms" . Int. Conf. on VLDB , Stockholm 1985 .

[GARD84] GARDARIN G, VALDURIEZ P, VIEMONT Y: "Predicate Tree: an Approach to Optimize Relational Query Operations", Database Engineering Conf. , Los Angeles 1984.

[GARD86] GARDARIN G, FAUDEMAY P, MICHEL R, VALDURIEZ P, VIEMONT Y: "An Integrated Approach to Multi-dimentional Searching Using Predicate Trees and Filtering" in preparation.

[GOOD81] GOODMAN J R: "An Investigation of Multiprocessor Structures and Algorithms for Database Management", University of California at Berkeley, Technical Report UCB/ERL, M81/33, 1981.

[HSIA85] HSIAO D K, DEMURJIAN S : "Benchmarking, Database Systems in Multiple Backend Configuration", A Quatterly Bulletin of the I.E.E.E. Computer Society, Technical Comitee on Database Systems, V8, N°1, 1985 .

[KITS83] KITSUREGAWA M et al: "Application of Hash to Database Machine and Its Architecture". New Generation Computing, N°1, 1983 .

[MOTO82] MOTO-OKA, ed., Proc. Int. Conf. on Fifth Generation Computer Systems, North Holland, Amsterdam 1982.

[OZKA85] OZKARAHAN E, OUKSEL M : " Dynamic and Order Preserving Data Partitioning for Database Machine ". Int. Conf. on Very Large Databases , Stockholm 1985.

[VALD84] VALDURIEZ P., GARDARIN G. : " Join and Semi-join Algorithms for a Multiprocessor Database Machine". ACM Transaction on Database Systems, V9, N°1, 1984

[VALD84b] VALDURIEZ P., VIEMONT Y.: " A Multikey Hashing Schema Using Predicate Trees", ACM SIGMOD, Boston, 1984 .

6. Annexes

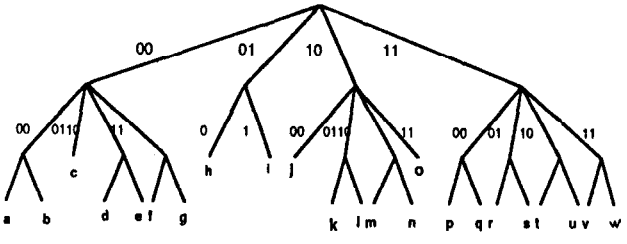
Annex A: clustering and partitionning on a relation

PRODUCT (WINE# , PRODUCER# , QUANTITY)

Predicate tree definition:

1st level: branch number = WINE# MODULO 4
2nd level: branch number = PRODUCER# MODULO 4
3nd level: branch number = 0 if QUANTITY < 100 , else 1

Clustering tree:



There are 2 ALGMs and 2 disks, the physical repartition on those disks is given by the function:
disk number = (1st level branch number) MOD 2

Directory part of disk0

$\{Ri\}_{A2}$	signature prefix	address
R1	**00000**	a
R1	**00001**	b
R2	0001	c
R3	00100	d
R3	00101	e
R4	00110	f
R4	00111	g
R1	**1000**	j
R2	10010	k
R2	10011	l
R3	10100	m
R3	10101	n
R4	1011	o

Directory part of disk1

$\{Ri\}_{A2}$	signature prefix	address
R0	010	h
R0	011	i
R1	**11000**	p
R1	**11001**	q
R2	11010	r
R2	11011	s
R3	11100	t
R3	11101	u
R4	11110	v
R4	11111	w

$\{Ri\}_{A1}$=R1{a b c d e f g}, R2{h i}, R3{j k l m n o}, R4{p q r s t u v w}
$\{Ri\}_{A2}$=R0{h i}, R1{a b j p q}, R2{c k l r s}, R3{d e m n t u}, R4{f g o v w}
Signature profile according to $R1_{A2}$= ••00•
Signature profile for the criterium QUANTITY="50" = ••••0
profiles intersection = ••000

Annex B: management of multiple copies

This example is based on the relation PRODUCT of Annex A. We use only two disks. Thus each disk holds a sample of all relations. However, in normal use, each disk addresses one half of the data.

B1: insertion of tuple /23/101/50/

1) Calculation of the tuple signature S = 11010
2) Calculation of the insertion profile; set 2nd bit at '•'
PI = 1•010
3) apply MOD 2 on the two first bits of the insertion profile
(10) MOD 2 = 0 and (11) MOD 2 = 1 => insertion on disk 0 and 1

B2: query

Which producers have produced wines with QTY=50?

1) Selection profile: ••••0

search on disk 0 search on disk 1

2) determination of prefixes list:
(•• MOD 4) MOD 2 = 0 => 00, 10
(•• MOD 4) MOD 2 = 1 => 01, 11

3) intersection of the selection profile with the prefixes list:
pr1= 00••0 | pr2= 10••0 | => •0••0
pr1= 01••0 | pr2= 11••0 | => •1••0

B3: breakdown of disk 1

Search of the list of prefixes of disk 2:
(•• MOD 4) MOD 2 = 1 => 01, 11
Each ALGM adds to its prefix list prefixes 01 and 11.
For the preceding query, we have four profiles:
pr1: 00••0 |
pr2: 10••0 |
pr3: 01••0 |
pr4: 11••0 | => ••••0 (the whole disk 0 is then accessed)