

Hash Join Algorithms in a Multiuser Environment

Hansjörg Zeller, Jim Gray

TANDEM

19333 Vallco Parkway, LOC 3-05

Cupertino, CA, 95014

Summary

As main memory becomes a cheaper resource, hash joins are an alternative to the traditional methods of performing equi-joins: nested loop and merge joins. This paper introduces a modified, adaptive hash join method that is designed to work with dynamic changes in the amount of available memory. The general idea of the algorithm is to regulate resource usage of a hash join in a way that allows it to run concurrently with other applications. The algorithm provides good performance for a broad range of problem sizes, avoids limitations for memory and relation size and uses advanced I/O controllers with track-size I/O transfers. It has been implemented as a prototype in NonStop SQL, a DBMS running on Tandem machines.

TABLE OF CONTENTS

1. Introduction.....	1
2. Execution Cost for Different Join Algorithms.....	1
2.1 Comparison.....	5
2.2 Parallel, Hash-Partitioned Joins.....	5
2.3 Influence of Join Selectivity and Skew.....	6
3. Hashing Strategies for Hash Joins	7
3.1 Main Memory Hash Tables with Duplicate Keys.....	7
3.2 Hashing in Virtual Memory	9
4. An Adaptive Hash Join Algorithm	10
4.1 Performance Measurements.....	16
5. Conclusions.....	17
6. References.....	18

1. Introduction

A variety of hash join implementations has shown that hash joins are the method of choice, if no indices are available on join columns and if the join result does not need to be sorted on the join columns. Most of these implementations have been done on database machines or in research environments. This paper, addresses the problem of having hash join algorithms perform in a multi-user environment on a multi-purpose computer. In such an environment it is very difficult to assign a static amount of memory to a process performing a join, especially if this process needs substantial amounts of memory. In addition to this, overflow handling is a problem area in hash join algorithms. Overflow of a hash table can result from an incorrect estimate of the query optimizer or from a lack of memory at run-time. By introducing dynamic decisions about bucket partitioning (like it is done in the GRACE join [Kits83], a better performance for overflow cases can be achieved. Finally, an acceptable performance should be possible in the worst case, where data are extremely skewed.

Section 2 expresses join cost in terms of CPU cycles, I/O operations and main memory consumption. Section 3 discusses hash access strategies for main memory hash tables, and section 4 introduces the adaptive hash join algorithm.

2. Execution Cost for Different Join Algorithms

It is well known, that simple nested loop joins on unclustered attributes have CPU cost in the order of n^2 and that sort/merge joins reduce this cost to $n \log n$. To be able to be more precise, we will make the following assumptions:

- Let **R**, **S** be base relations, containing **n** tuples each, stored in **b** blocks each on a disk. The block size is **B**.
- In the following, we assume a join of **R** and **S** which could be expressed in SQL as

```
SELECT *  
FROM R,S  
WHERE R.r = S.s;
```

The join result consists of **n** tuples, where each tuple of **R** is joined with exactly one tuple of **S**. We will only consider equi-joins in this paper since sort/merge and hash algorithms are not able to perform arbitrary joins.

- Assume that there are no indices (and no clustering) for **R.r** and **S.s**.

These assumptions describe a join which cannot make use of index structures and which is not a degenerated case, where the join result consists of either 0 or n^2 tuples. R and S are of the same size to make the cost formulae simpler.

A Simple Nested Loop Join will scan the outer relation sequentially and will do a full scan of the inner relation for each tuple read from the outer relation. This requires $n + n^2$ read operations (n for the outer, n^2 for the inner relation). This formula expresses the cost in terms of CPU cycles. In a simple algorithm, this would need $b + n * b$ disk I/O operations. This can be reduced substantially by adding a loop over the blocks of the tables and by performing a set of nested loop joins for each pair of blocks in the inner and outer table. This reduces the number of I/O operations to $b + b^2$. However, any ordering in the inner and outer table is destroyed.

In its simplest form, the algorithm uses a constant amount of main memory; it needs just two block buffers, one for the inner and one for the outer relation. Introducing a *database buffer* could reduce the I/O operations to $2b$, if $b + 1$ block buffers (1 for the outer, b for the inner relation) are available. The number of CPU cycles remains unchanged.

The minimum amount of main memory needed by Sort/Merge is three disk block buffers because in the sort phase, two input buffers and one output buffer are needed. The CPU cost for a merge join is determined by the sort cost for the participating tables plus a linear cost for performing the actual merging. Dependant on the memory size, it may be possible to keep one or both participating tables in main memory after they have been sorted and therefore to save I/O operations for writing temporary files. Fig. 1 shows a qualitative cost function for disk accesses as a function of the amount of memory available.

A simple hash join is performed in two phases. In the *building* phase, the inner relation is hashed into main memory. The join attributes are used as a hash key. For now, we assume a very simple algorithm with no overflow handling, requiring that the entire inner relation fits in main memory. After this, the second phase called *probing* phase is performed. In the probing phase, the outer relation is read sequentially and for each record in the outer relation the matching records in the inner relation are retrieved. Probing can be done at a constant cost because the inner relation is now in memory and has a hash access path on the join attributes.

Since base relations are read only once and no temporary relations are needed, this algorithm uses $2b$ disk accesses. It also needs $b + 1$ block buffers (size B , not counting the fragmentation overhead of the hash table) in main memory to keep the whole inner relation and one buffer for the outer relation. This is similar to a nested loop join, when the inner relation is in the database buffer, but it needs less CPU cycles, because a hash

access is done to the inner table rather than a sequential search. Since hash methods work -- nearly -- independent of the size of the hash table, the join cost contains four linear components for reading the inner table, hashing it, reading the outer table and probing into the hash table:

$$o(\text{Simple Hash Join Cost (CPU+I/O)}) = o(n+n+n+n) = o(n) \quad (1)$$

In the world of Hash Joins, there are many different methods to cope with insufficient memory. The methods we would like to introduce here are Grace and Hybrid Hash Join. In Grace, the inner relation is divided into buckets, each of them small enough to fit in main memory. The outer relation is divided in the same way; however the size of the outer buckets does not matter. The buckets are stored in disk files and then processed sequentially. Because inner and outer relation are written once into a temporary file, we need $6b$ disk accesses (read R, S , store R_{temp}, S_{temp} , read R_{temp}, S_{temp}).

If $k+1$ disk blocks fit in memory, a bucket must not be larger than k blocks (one block buffer is needed for the outer relation). As a consequence, we partition inner and outer relation into b/k buckets in the building phase. This will require b/k input and output buffers. Hence, the minimum amount of memory for this algorithm is

$$\text{MemoryNeeded} = \max\left(\frac{b}{k}, k\right) = \sqrt{b} \quad (2)$$

pages. CPU cost is independent of the actual amount of memory available, if the overhead of switching between pairs of buckets is neglected.

The Hybrid algorithm described by DeWitt and Gerber [DeWi85], introduces a dynamic decision between Grace and Simple Hash Join. For memory sizes between $\sqrt{b+1}$ and $b+1$, the hybrid join performs a continuous transition between Grace and Simple Hash Join. This is achieved by using the excess memory not needed by the disk buffers to store a variable number of tuples (this becomes the first bucket) in main memory rather than in a disk file and therefore reducing the number of disk I/O operations continuously, as seen in Fig. 1.

In a Hybrid Join, main memory is split into n block buffers for creating n buckets on disk and in an area of size r (in blocks) to build the hash table for the first bucket. This changes when the temporary files on disk have been built and buckets $2 \dots n+1$ are processed. Then one bucket is hashed in main memory (using up to k blocks) and one input buffer for the outer relation is used. If we assume optimal memory utilization, the first bucket has the maximal size of $r = k - n$ and the n other buckets are also of maximal size k . In this configuration, the number of disk I/Os is the number of reads for the participating tables plus the I/Os for writing and reading the buckets $2 \dots n+1$ to a temporary file:

$$\# \text{ Disk I/Os} = 2b + 4(b-r) \quad (3)$$

The r blocks of the first bucket (assuming that buckets in inner and outer relation are of the same size) are not written to the temporary file. Using the equations $n k = b - r$ (the n last buckets are each as big as the available memory and have a total of $b - r$ blocks) and $r + n = k$ (the first bucket occupies all the space not needed by the output buffers for buckets $2 \dots n+1$), the number of disk I/Os can be expressed as

$$\# \text{ Disk I/Os (Hybrid Hash)} = 6b - 4k + 4 \times \frac{b - k}{k - 1} \quad (4)$$

for values of $\sqrt{b} \leq k \leq b$. It should be mentioned, that -- because the number of buckets must be a whole number-- it is not always possible to achieve an ideal bucket partitioning.

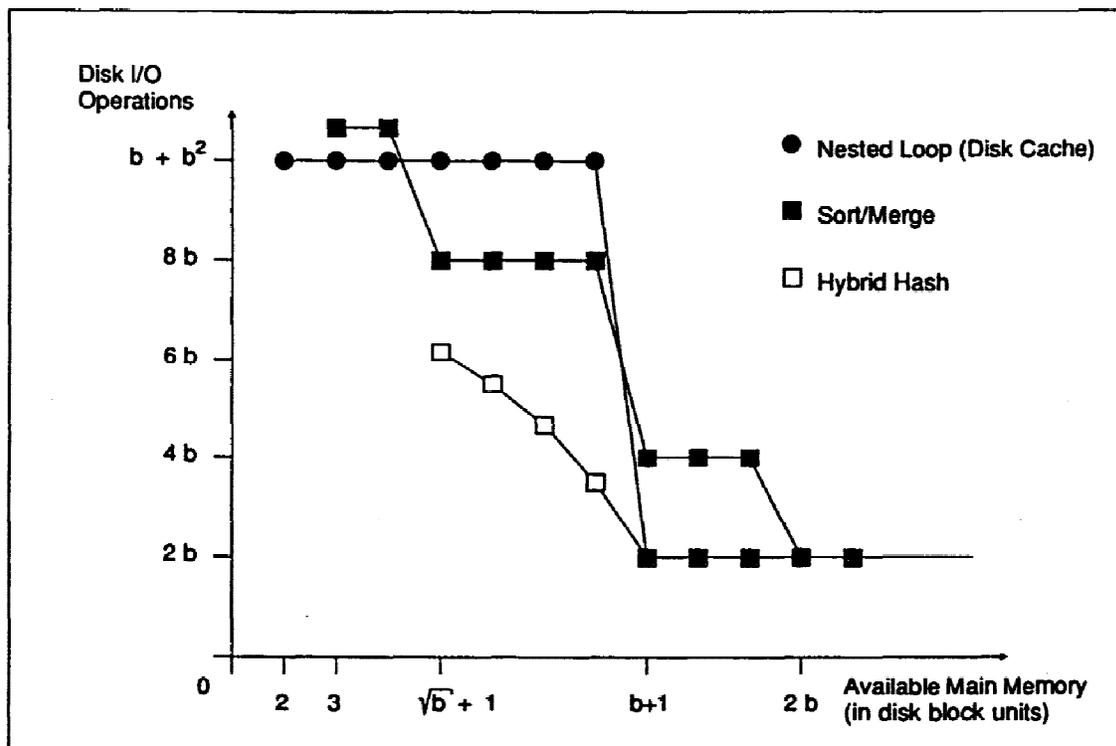


Fig. 1: Disk I/O vs. Memory Consumption (Qualitative)

2. 1. Comparison

A comparison between Simple Nested Loop, Sort/Merge, Simple Hash, Grace and Hybrid Hash Joins shows that hash joins are the algorithms of choice if a minimum of main memory is available for the operation. The minimum amount of memory just grows as a square root function, so it is not too hard to meet the condition. Table 1 lists CPU cost, I/O cost and memory consumption for the different join algorithms and indicates whether temporary relations on disks have to be created:

Join Type	Memory (in blocks)	CPU	I/O	Temporary Tables
Simple Nested Loop	2	$o(n^2)$	b^2	no
	b	$o(n^2)$	2b	no
Sort/Merge	3	$o(n \log n)$	$6b + 4b \log b$	yes
	\sqrt{b}	$o(n \log n)$	8b	yes
	b	$o(n \log n)$	4b	for 1 table
	2b	$o(n \log n)$	2b	no
Simple Hash	b	$o(n)$	2b	no
Grace Hash	\sqrt{b}	$o(n)$	6b	yes
Hybrid Hash	$\sqrt{b} \dots b$	$o(n)$	6b ... 2b	yes

Table 1: Cost Functions of Join Algorithms

2.2 Parallel, Hash Partitioned Joins

Until now we always assumed that the join was performed by one processor. On a multi-processor system, it is desirable to have several processors and probably several disks working on a join query. This is achieved by hash partitioning data between these processors [Gerb86]. Hash partitioning is independent of the join algorithm itself, but it is restricted to equi-joins. The principle of hash partitioning is to split a large join into p smaller joins. The p joins are independent of each other and can be executed in different processors. Applied to the cost functions in the last section, the number of records and blocks has to be divided by p to get the cost for one partition. The part of reading base relations, however, cannot be done in parallel, unless the base relations reside on several disks (probably served by several processors). For algorithms with a fast growing cost function, like simple nested loop, the gains with parallel execution are most visible. They have a speedup growing more than linear, meaning that two processors may perform more than twice as fast than a single processor. Hash joins seem not to be accelerated so much by parallel execution. This is due to their linear cost function. They are, however, able to take advantage from multi processor systems in another way: if the main memory

of one processor is not sufficient, a hash partitioning between several processors can help to reduce disk I/O. Through utilization of more memory (in several processors), parallel hash joins can also get a more than linear speedup.

Common problems with parallel algorithms are contention on shared resources and communication costs. In so-called Shared Nothing Systems (Tandem, Teradata, GAMMA), mainly communication costs determine the overhead for parallelism. Contention is not a big problem, because after the partitioning phase, every processor works on its local copy of the data. As mentioned, the partitioning phase can only be done in parallel, if data are stored on several disks. The amount of data to be sent is less or equal to the size of the base relations (most projections and selections may be applied before sending the data). If we assume blocks of equal length for messages and disk I/Os, then the communication overhead depends on the selectivity factors and the quotient of message cost and disk I/O cost. In typical Shared Nothing Systems this quotient is low (<0.3) and therefore the overhead is acceptable.

2.3. Influence of Join Selectivity and Skew

Until now, the model was restricted to a join on two relations of equal size with a 1:1 relationship between their records. Cost functions change if 1:n and n:m relationships are used and if relations of unequal size are introduced. Generally, for asymmetric algorithms like Hash Joins it is an advantage to have a small and a large relation. This is obvious because only the smaller relation has to be kept in main memory and the size of the larger relation does not influence the amount of memory needed. Hence, even extremely large joins can be done very efficiently if the inner relation is small enough. On the other hand, merge joins perform worse under such circumstances because it is expensive to sort a very large relation. Nested loop joins are not very sensible to different sizes of inner and outer relation as long as the inner relation does not fit entirely in main memory.

The kind of relationship in the join also determines the number of its result tuples. If join attributes are a key in at least one of the participating relations (1:1 and 1:n relationships), there cannot be more result tuples than base tuples. If this is not the case (n:m), then the number of result tuples can grow in a more than linear fashion, like a cartesian product. In this case, it is impossible to compute join results in linear CPU time. Such a situation is often used as a strong argument against hash joins, because hash access methods are not designed to cope with highly skewed key distributions. On the other hand, a merge join will gradually degenerate into a nested loop join in such extreme cases. With an appropriate overflow technique a hash join may actually be the most efficient algorithm for highly skewed data distributions. In general, a merge join will be better, if the join selectivity is low, and a hash join will be better for high join selectivities. This is because for low selectivities a merge join will only need few repositionings in the participating files, while in a hash join no such optimization can be performed. For skewed attribute values and low join selectivities, bit vector filtering can be a big advantage and it can

reduce the disadvantages of hash joins in these configurations. In [Schn89] it is stated that bit vector filtering is also beneficial in most other cases.

3. Hashing Strategies for Hash Joins

3.1. Main Memory Hash Tables with Duplicate Keys

Although they are relatively easy to implement, hashing strategies are an important aspect of hash joins. In the literature, several modifications of a standard hashing algorithm are proposed giving a variety of methods with different time and space requirements. The general problems with hashing are non-uniform key value distributions and dynamic reorganization if the number of records in a hash table increases or decreases significantly. Most of the proposed strategies assume that the hash method is used to create a *primary* access path on a *disk file*. For hash joins, neither the hash key is unique nor are the records stored in disk blocks¹. While it is a big disadvantage to have duplicate key values, having all data in main memory may help make the implementation efficient.

Fig. 2 shows the different stages of hashing in a parallel Grace or Hybrid Hash Join. First, the base relation is split into several partitions using a hash function to determine the partition number. Each partition is assigned to a processor. For each partition, data records are again hash partitioned into buckets and stored on disk. Then one or more buckets at a time are read from disk and moved into an in-memory hash table. This is the third stage of hashing and the critical one because data are actually stored in main memory. The partitioning into buckets is done for the purpose of avoiding an overflow of the main memory table. If two or more buckets are combined, because they fit into main memory simultaneously, this is called *bucket tuning*.

In main memory, a certain number of hash *chains* are created. Tuples are assigned to exactly one chain by their hash values. Therefore, two tuples with the same key or the same hash value will always be in the same chain, while one chain can contain tuples with different key values and even different hash values². With a good hash function and a uniform key value distribution, all chains will have approximately the same length.

In a disk file with a hash access path, one disk block is usually used for storing the records of a hash chain. If the block overflows, different techniques may be applied (e. g. to store the overflowing tuples in the next block). In main memory, however, it is not

¹ The bucket files used by Grace and Hybrid Joins are not hash tables. They just contain all the tuples belonging to one bucket in an arbitrary sequence. When a bucket file is read back into main memory, the actual hash table is built.

² In all partitioning stages, the same HASH function can be used, while three different SPLIT functions can be used to generate partition, bucket and chain numbers in the appropriate ranges. Usually, the modulus operator is used as a split function.

necessary to reserve a fixed amount of space for each hash chain because there are no block accesses to be minimized. The tuples of a hash chain may be connected by a linked list instead, if a pointer field is attached to each record. Tuples can then be stored adjacent in a heap in their sequence of arrival. However, to access the head of a hash chain, an array of chain anchors has to be allocated.

With this technique, the hash structure becomes less sensitive to skewed data because hash chains may be of arbitrary length without any overflow handling. The portion of the hash table used to store tuples is a compact section growing linearly as more and more tuples are inserted. Only the table as a whole can overflow if its size exceeds the memory size. Another critical parameter is the number of hash chains, because this also determines their length. Since only one pointer is needed per hash chain, it may be economical to make the chain anchor array relatively large, so that chains have only a few records. A typical chain anchor array might use from 1 to 10 % of the hash table (4 byte/anchor, 100 byte/record, 4 records/chain).

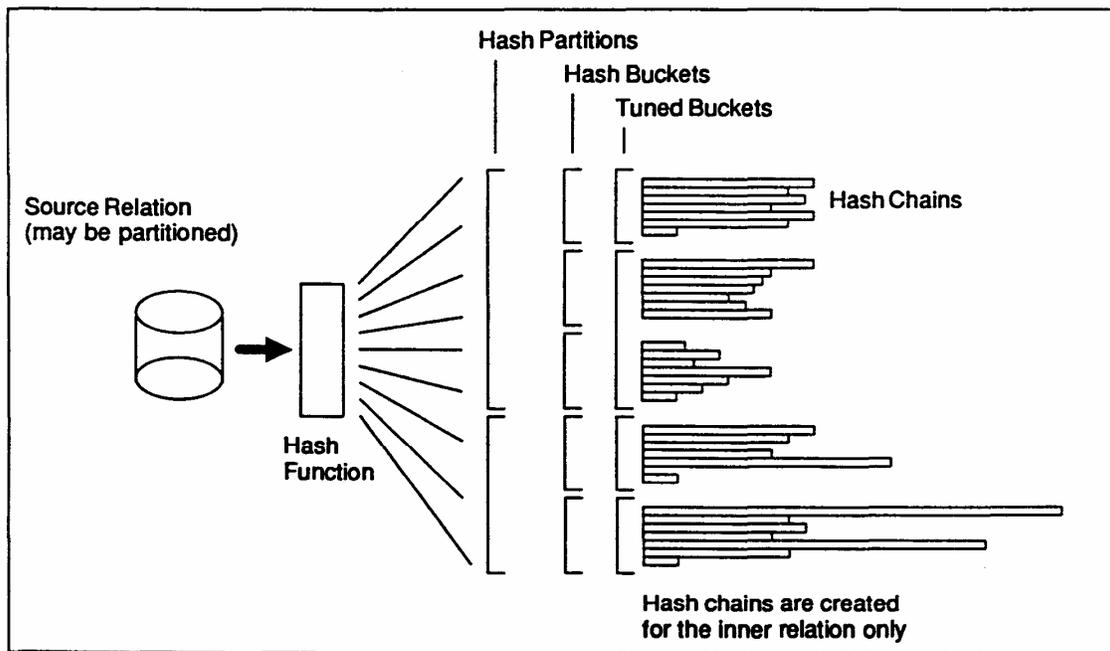


Fig. 2: Stages of Hashing and Bucket Tuning

Another aspect is the transfer of tuples from main memory to the temporary bucket file. This transfer should not involve record-by-record moves. Instead, it is desirable to have the same data representation in the bucket file and in main memory. Since records are

kept in linked lists, they may be stored anywhere. Fig. 3 shows a hash table using block buffers for both resident and non-resident buckets³.

This storage method allows swapping out buckets dynamically, if this becomes necessary. Also, it is not necessary to write any buffers to the bucket file, until the available memory is exhausted. Up to that point, all buckets are kept memory-resident in linked lists of block buffers. While a bucket is swapped out to the bucket file, the link pointers between buffers are replaced with file positions. This allows to store the buffers in random order on the hybrid file, or to share one hybrid file between many buckets.

Fig. 3 also shows that several buckets share one buffer. This refers to the algorithm described later in this paper.

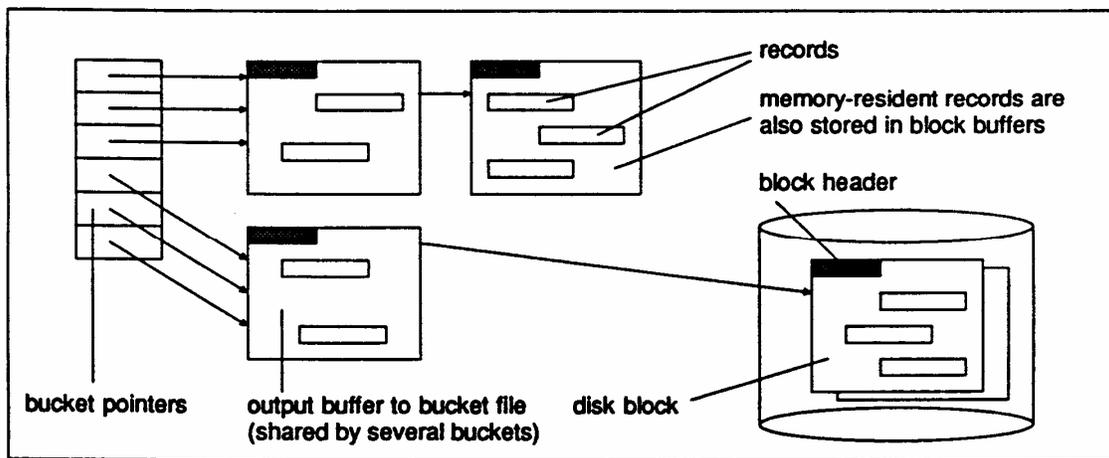


Fig. 3: Using Block Buffers as Memory Allocation Unit

3.2. Hashing in Virtual Memory

In a multiuser system, it may not be desirable to reserve large amounts of main memory exclusively for hash join purposes. Therefore, the hash table must reside in virtual memory and may be partially written out to a swap file. Generally, the performance of a hashing strategy will suffer dramatically from page faults, because the access patterns in the hash table do not show locality and therefore no replacement algorithm (like LRU) for main memory pages will work satisfactorily with hash tables. Therefore, only a small number of page faults can be accepted. If hash chains are stored in linked lists, many

³ Fig. 3 does not show the actual hash chains (the linkages between records and the pointer array). The shown array of bucket pointers has one entry per bucket and is therefore much smaller than the chain anchor array.

memory pages may be accessed while following one hash chain. This can increase page fault rates if main memory is scarce.

The linear growth of the hash table is a very desirable property for main memory tables. It is also helpful that the linked hash table does not need to be contiguous, because no address computations are made in it, except for the chain anchor array. Big gaps inside a hash table may in some operating systems cause large swap files to be allocated, wasting disk space resources. To determine the amount of paging disk I/Os acceptable for a hash join, it should be considered that paging I/Os cause random accesses on the paging disk, while file I/Os of Sort/Merge and Hybrid joins have sequential access patterns. For this reason, join file I/O can make use of large block transfers to disks, while paging I/O usually uses blocks of small or medium size only (0.5 ... 16 KB).

In many operating systems, there is no safe way to determine the current amount of 'free' main memory. So, it would be desirable to have a join algorithm being able to dynamically adjust the hash table size, while the join is already in progress. This adjustment can be driven by a measurement of the number of page faults during the join process. The next section of this paper describes an extension to the Hybrid Join algorithm which is designed to cope with this and other problems.

4. An Adaptive Hash Join Algorithm

There are two principal input parameters to a hash join algorithm: the size of the inner relation and the available memory. Unfortunately, these parameters cannot always be determined exactly at the time the join starts. Memory availability in a multiuser environment changes continuously, if no memory can be locked by the process, and the size of the input relation often depends on selectivity factors of predicates. Therefore, the algorithm described in the following reacts dynamically to changes of these input parameters. Possible reactions are:

- a) Swapping some memory-resident buckets out to the bucket file,
- b) Dynamically splitting buckets into smaller ones,
- c) Changing the transfer size (and therefore the buffer size) to the temporary file.

In a Hybrid Join, data are hash partitioned three times as shown in Fig. 2. However, no bucket tuning takes place. This is because a Hybrid Join is designed to use only a few disk buffers for doing the data partitioning into buckets. The rest of the memory in this phase is used to build the hash chains for the first bucket (which may be of different size

than the others). If enough main memory is available to hold all the data, no bucket files on disk are created and the bucket partitioning phase is skipped completely.

The Adaptive Hash Join differs from a Hybrid Join in the following way:

- All buckets have the same size. Initially, all buckets are memory-resident. If not enough memory is available during the process of building the inner hash table, buckets are written out to the bucket file until enough memory is freed up. Therefore, several buckets may be memory-resident at the same time.
- Initially, a multiple of the number of buckets that is presumably needed is allocated. This allows bucket tuning. However, to save buffer space, several buckets share one block buffer. Recall that block buffers are used to hold memory-resident tuples as well as to serve as output buffers to the bucket file. Allocation of an excess of buckets and sharing buffers outweigh each other so that the total number of buffers is approximately the same as in a Hybrid Join. In the following, a set of buckets that share a buffer will be called a cluster. Clusters may be split dynamically if they receive more than the estimated number of records.

We are now able to describe the Adaptive Hash Join Algorithm:

1. At the beginning of phase 1 of the join, the **B** buckets are divided into **B/C** clusters. For each cluster, a buffer of size **S** is allocated. The initial values of **B**, **C** and **S** are determined on the estimates made by the query optimizer. **B** never changes throughout the join process, but the number of clusters, the default buffer size **S** and the memory size **M** may vary during the join. The initial value for **M** is determined by the operating system. If $B/C * S > M$ (there is not enough room for all the buffers), then **S** is decreased accordingly. If the minimum for **S** is reached (minimum transfer size for a disk I/O), then **B** is set to $\max((M/S), 1)$ and **C** is set to **1**.
2. Records are read from the building relation and partitioned into buckets. Records are stored in the buffer that is allocated for the bucket's cluster. This is repeated until all records are read or a buffer becomes full. If all records are read, the algorithm continues with step 9.
3. If a buffer overflows, a new value of **M** is determined by the operating system. If the bucket's cluster is already swapped out, then the buffer is written to the bucket file and cleared and the algorithm proceeds with step 7.

4. If enough memory is available, and the bucket's cluster is not yet swapped out to the bucket file, then a new buffer of size S is allocated and inserted into a linked list of buffers for the cluster. If not enough memory is available to allocate a new buffer, then clusters are swapped out, until enough space is available. Swapping out a cluster is done by writing all its buffers to the bucket file and deallocating the buffers in memory -- except the last one, which will serve as an output buffer to the bucket file.
5. If all clusters are swapped out and there is still not enough memory available, the default buffer size S is decremented. Existing output buffers are shortened to the new size until enough memory is available or the minimum size of S is reached. Note that this indicates that there is not even enough room to hold just the output buffers to the bucket file.
6. If still not enough memory is available, this is ignored. This is possible because the algorithm performs in virtual memory. Restriction of memory usage is done for performance reasons only.
7. Then, a cluster overflow check is performed. If the cluster consists of more than one bucket, and if it contains more records than would fit into memory of size M , then the cluster is split into two smaller clusters using separate buffers. Already existing buffers of the old cluster (residing on disk) remain unchanged; they now contain tuples from more than one cluster. For the allocation of the new buffer, the same heuristics as described in steps 4, 5 and 6 are applied.
8. If the cluster overflows but consists of only one bucket, then the overflow is ignored at this point of the algorithm. This situation will be solved in step 11. Continue with step 2.
9. After all tuples of the inner relation have been read, the next phase of the join starts: Read the outer relation. Partition the outer relation into clusters in the same way as the inner relation. For each record, determine the cluster which will contain the matching inner records. If the cluster is memory-resident, join this record immediately; otherwise store it in the according bucket file of the outer relation. Repeat this until the end of the outer relation is reached. Note that the outer relation is partitioned according to the final partitioning of the inner relation. No dynamic change of the partitioning schema takes place.
10. Determine the actual value of M . Read as many clusters from the bucket file as fit in memory. Create the hash chains for these tuples. If there are no more clusters, the join is completed.

11. If not even one cluster fits into memory, then read a part of the next cluster and build the hash chains. Save the rest of the cluster for the next execution of step 10.
12. Read the matching cluster from the outer relation and perform the join (record by record, accessing the inner relation using the hash chains). Go to step 10.

In cases where a constant (and sufficient) amount of memory is available and where the actual size of a bucket does not exceed its expected size, steps 2, 3, 4, 9 and 12 approximately describe a Hybrid Join. Steps 6 and 8 will have to be replaced by another overflow strategy in a Hybrid Join. Steps 1, 3, 4, 5, 6, 7, 8, 10 and 11 contain dynamic decisions based on run-time parameters that can change during the join process. The strategies a), b) and c) mentioned earlier are implemented by steps 4, 7 and 5.

In the following, a few scenarios will be discussed. The varying parameter will be the actual number of tuples in the building relation, compared to the estimated number of tuples. This will show that an acceptable performance will be obtained in all cases, making the algorithm suitable to be used as a standard method, like Sort/Merge. An estimate of the number of records will be supplied by the query optimizer. This estimate may be derived from statistical information and can therefore be magnitudes lower or higher than the actual number of records in the inner table:

- If the estimate of the record count was much too high, there are not enough records in each cluster to fill one disk buffer. Main memory is utilized badly, because many buffers that are filled only partly are allocated. The real effect is not so bad, however, because main memory is divided into pages and many pages fit into the unused portions of a large disk buffer. These unused pages do not occupy main memory, they only use swap file space. On the other hand, it is now possible to perform bucket tuning, and less than the estimated number of iterations of steps 10 through 12 are necessary.
- If the estimate was somewhat too high, more clusters than expected have been kept memory-resident. Less buffers could have been used, therefore some memory space was wasted.
- If the estimate was about right, it was not necessary to allocate more buckets and to build clusters. The standard Hybrid algorithm avoids this, but the overhead doing it should be minimal.
- If there are more records than expected, but still less than the maximum that can be handled using the available buckets, then some clusters will be split. The split

clusters will still be small enough to fit in memory. However, some overhead is introduced because the common part of the split clusters is read several times from disk in step 10. If this would cause some blocks to be read more than three times, then it is more efficient to repartition the shared part of the buckets in a separate step. The number of disk accesses is still limited to an acceptable amount.

- If the estimate was much too low, then the clusters will be split until they have no more than one bucket. Even these buckets will be larger than the available main memory. This will cause step 11 to be executed. It is the only case where data from the outer bucket file are read more than once. This situation is described by [DeWi85, Naka88] as *Hash Loop Join*. Hash Loop Joins can still have better performance than Sort/Merge Joins, but they may also be more expensive.
- In case of attributes with very many duplicate values (e. g. “M” and “F” for sex), only some buckets will overflow and cause a Hash Loop Join. Basically, the same rules apply to this case. The performance in comparison with Sort/Merge depends on the join selectivity. If many output tuples are generated, the Hash Loop Join will perform better. If only few tuples match the join condition, a Sort/Merge Join will need less disk accesses and be faster.

Fig. 4 is a graphical display of some of the scenarios. The lower right corner shows the critical configuration where an unexpectedly large relation has to be joined in a small memory. In this case, the advantages of using large buffers are lost. The necessary cluster splits will also cause some performance degradation. The upper right corner shows a large table with sufficient memory. Buffers remain large and less cluster splits are necessary because bigger clusters can be processed. On the left side of Fig. 4, the configurations for sufficient (or even too much) memory are shown. The algorithm will always try to keep the buffers to the bucket file as large as possible. This has a higher priority than keeping clusters memory-resident. Buffers are usually used for more than one I/O operation. If memory-resident clusters are written to the hybrid file, the data is only written once. Therefore, it is more efficient to keep a larger buffer space at the cost of less memory-resident clusters. The only case where space is wasted by output buffers is where too many clusters are allocated.

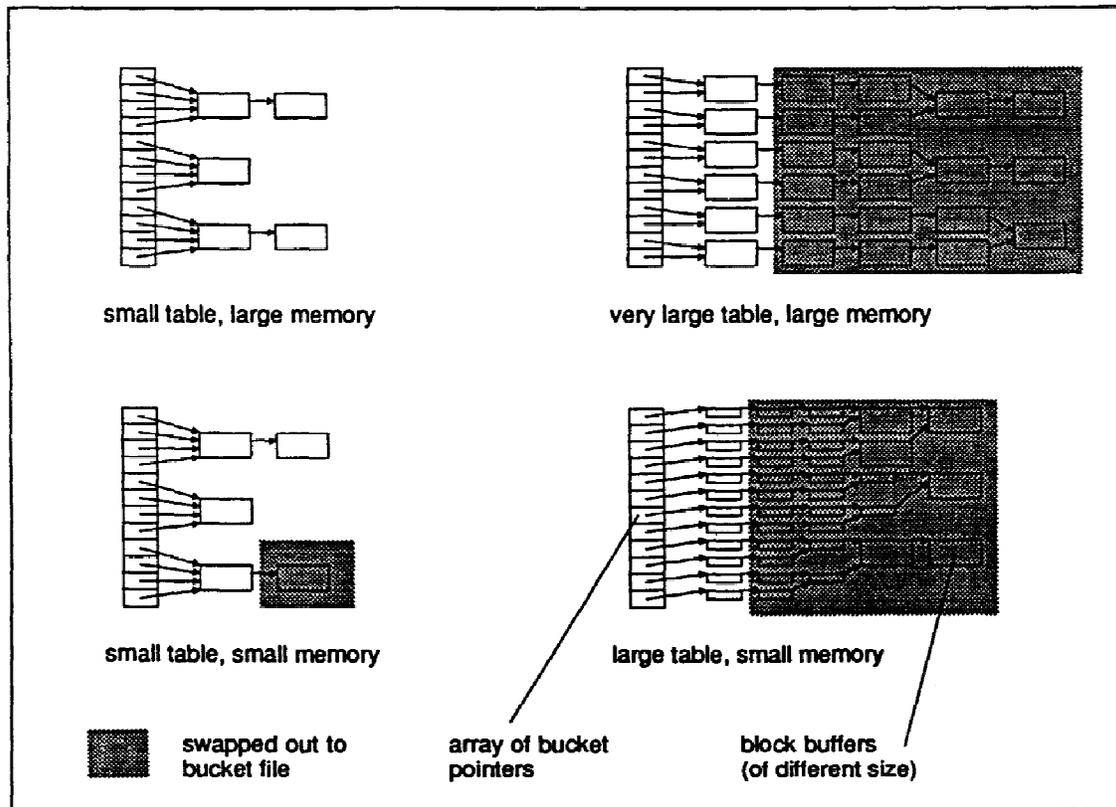


Fig. 4: Adaptive Configurations

An important question is whether an Adaptive Hash Join performs with linear cost as long as \sqrt{b} buffers in memory are available, if the inner relation fits in b buffers. This can be easily answered with 'yes' if the large buffer size that is actually used in the algorithm is also used to compute the value of b , and if it is assumed that no cluster splits take place and the amount of available memory remains constant. Even if the buffer size B is set to its minimum (depending on the actual disk sector size), linear performance will result because step 1 of the algorithm will already decrease the buffer size to its minimum in this case. Only then is it possible to allocate all the output buffers. This proves that the very important property of linear performance has not been sacrificed to get a more flexible overflow mechanism or to use larger disk blocks.

On the other hand, an Adaptive Hash Join will also be able to perform with only two buffers of main memory. In this case, the number of buckets is set to one in step 1 and the algorithm will perform like a nested loop join, with the exception that the inner relation is unnecessarily scanned and written to a bucket file in the first phase. This is certainly a case that will have bad performance like all other algorithms, if the table is large, but adding very small amounts of memory will show big performance

improvements: three buffers will already allow two buckets and therefore let the join perform four times faster.

4. 1. Performance Measurements

Some preliminary performance evaluations have been done with an implementation of the adaptive hash join algorithm in NonStop SQL, a DBMS running on Tandem computers. Figure 5 shows the results. The query measured is a join between two Wisconsin-style tables on the unclustered, non-indexed column *unique1*. Each table is 7 MB large and contains 30,000 rows. The parallel query execution feature of NonStop SQL is not used. With a default buffer size of 28KB, and an initial allocation of 21 clusters by the query optimizer, the algorithm needs somewhat less than 700 KB to allocate all buffers with their full length. In this case, no buckets are kept memory-resident after the inner table has been read.

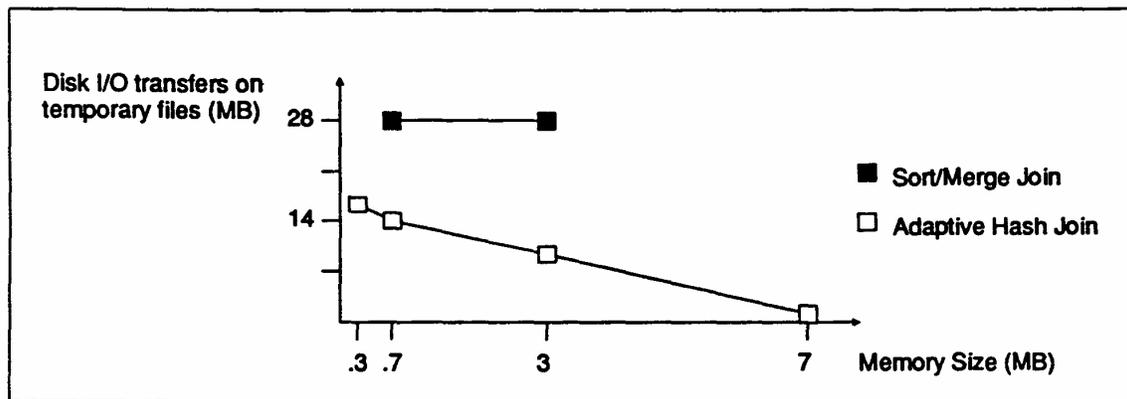


Fig. 5: Execution Cost of the Adaptive Hash Join

In the configuration with only .3 MB of memory, the buffers need to be shortened to 8 KB and cluster splits occur. The graphical display shows the amount of data transferred to and from the disks. In the case where only .3 MB of memory is available, the block size decreases by 1/3 so that three times more transfers are necessary. Despite this fact, the adaptive join still runs faster in terms of elapsed time than the merge join (791 vs. 1260 sec). Figure 5 also shows a reference point with a sort/merge join.

CPU time consumption is nearly constant in all cases, as it is predicted in the first part of this paper. The adaptive hash join consumes on average 62 percent of the CPU time of a sort/merge join. Figure 5 does not show the I/O operations necessary to read the input tables.

5. Conclusions

Hash Joins have linear cost, as long as a minimum amount of memory is available. This minimum is proportional to the square root of the size of the inner relation and proportional to the buffer size used for transfers to the hybrid file. We described an algorithm that will use large transfers to disks, as long as enough main memory is available, but will not reduce the maximum number of tuples, which can be joined in nearly linear cost. To be able to process relations of any size with acceptable performance, a set of adaptive overflow techniques was introduced. To make it possible to run this algorithm in a multiuser environment without major interference to other users, the amount of memory being assigned to a join process can be adjusted dynamically while the join is executing. This is especially important if large join queries are performed concurrently with interactive (OLTP) applications.

6. References

- [Brat 84] K. Bratbergsengen, *Hashing Methods and Relational Algebra Operations*, Proc. 10th VLDB, Aug. 1984, pp. 323-332
- [DeWi84] D. J. DeWitt., R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, D. Wood, *Implementation Techniques for Main Memory Database Systems*, Proc. ACM SIGMOD Conference, 1984, pp. 1-8
- [DeWi85] D. J. DeWitt, R. H. Gerber, *Multiprocessor Hash-Based Join Algorithms*, Proc. 1985 VLDB, pp. 151-164
- [Gerb86] R. H. Gerber, *Dataflow Query Processing Using Multiprocessor Hash-Partitioned Algorithms*, Dissertation, University of Wisconsin-Madison, Computer Sciences Technical Report #672, Oct. 86
- [Kits83] M. Kitsuregawa, H. Tanaka, T. Moto-oka, *Application of Hash to Data Base Machine and Its Architecture*, New Generation Computing 1, 1983, pp. 63-74
- [Naka88] M. Nakayama, M. Kitsuregawa, M. Takagi, *Hash-Partitioned Join Method Using Dynamic Destaging Strategy*, Proc. 14th VLDB (1988), pp. 468-478
- [Pong88] M. Pong, *NonStop SQL Optimizer: Query Optimization and User Influence*, Tandem Systems Review 4,2 (1988), pp. 22-38, Tandem Computers, Part. No. 13693)
- [Schn89] D. A. Schneider, D. J. DeWitt, *A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment*, Proc. ACM SIGMOD Conference 1989, Portland, Oregon, pp. 110-121
- [Shap 86] L. D. Shapiro, *Join Processing in Large Database Systems with Large Main Memory*, ACM TODS 11,3 (Sept. 86), pp. 239-264
- [Ston89] M. Stonebraker et.al., *Parallelism in XPRS*, UC Berkeley, Electronics Research Laboratory, Report M89/16, February, 1989
- (Tand87) Tandem Database Group, *NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL*, Proc. 2nd Int. Workshop on High Performance Transaction Systems, Asilomar, CA (Lecture Notes in Computer Science 359, Springer-Verlag, Ed.: D. Gawlick, A. Reuter, M. Haynie)
- [Vald84] Patrick Valduriez, Georges Gardarin, *Join and Semijoin Algorithms for a Multiprocessor Database Machine*, ACM TODS 9,1 (1984), pp. 134-161)
- [Zell89] H. Zeller, *Parallelisierung von Anfragen auf komplexen Objekten durch Hash Joins*, Proc. GI/SI Fachtagung: Datenbanksysteme in Büro, Technik und Wissenschaft, IFB 204, Springer-Verlag (in German).