The VLDB Journal (1996) 5: 64–84

The VLDB Journal
© Springer-Verlag 1996

Join algorithm costs revisited

Evan P. Harris*, **Kotagiri Ramamohanarao****

Department of Computer Science, The University of Melbourne, Parkville VIC 3052, Australia

Edited by Masaru Kitsuregawa. Received April 26, 1993 / Revised March 3, 1994 / Accepted October 13, 1994

Abstract. A method of analysing join algorithms based upon the time required to access, transfer and perform the relevant CPU-based operations on a disk page is proposed. The costs of variations of several of the standard join algorithms, including nested block, sort-merge, GRACE hash and hybrid hash, are presented. For a given total buffer size, the cost of these join algorithms depends on the parts of the buffer allocated for each purpose. For example, when joining two relations using the nested block join algorithm, the amount of buffer space allocated for the outer and inner relations can significantly affect the cost of the join. Analysis of expected and experimental results of various join algorithms show that a combination of the optimal nested block and optimal GRACE hash join algorithms usually provide the greatest cost benefit, unless the relation size is a small multiple of the memory size. Algorithms to quickly determine a buffer allocation producing the minimal cost for each of these algorithms are presented. When the relation size is a small multiple of the amount of main memory available (typically up to three to six times), the hybrid hash join algorithm is preferable.

Key words: Join algorithms – Minimisation – Optimal buffer allocation

1 Introduction

In the past, the analysis of join algorithms has primarily consisted of counting the number of disk pages transferred during the join operation, as this has been perceived as the dominant cost of the join algorithm. However, the difference between the time taken to locate a single disk page and transfer a single disk page is significant. The time taken to locate the page dominates. Hence, the difference between transferring two consecutive disk pages and two random disk pages is quite significant.

For example, if it takes 25 ms to locate the average disk page and 5 ms to transfer it from disk to memory, then reading two consecutive pages takes 35 ms, whereas reading two random pages takes 60 ms. When the cost of a join operation is calculated, the difference in this time should be taken into account.

The CPU cost of a join should be taken into account. Experience with the Aditi deductive database (Vaghani et al. 1994) has shown that the disk access and transfer times correspond to 10–20% of the time taken to perform a join. Thus, the CPU time is an important factor when determining the most efficient method to perform any given join.

There have been a number of join algorithms described in the past, none of which are optimal under all circumstances. We provide analyses of some of the more common of these. The nested block (nested loop) algorithm and the sort-merge algorithm are the algorithms used in most current database implementations. The nested block algorithm is used when a small relation is joined to another, and the sort-merge algorithm is used for larger relations (Blasgen and Eswaran 1977). For a while it was believed that the sort-merge join was the best possible algorithm (Merrett 1981); however, the description of join algorithms based on hashing (Kitsuregawa et al. 1983; DeWitt et al. 1984) indicated that this is not necessarily true. DeWitt et al. compared the sort-merge, simple hash, GRACE hash and hybrid hash join algorithms and concluded that hybrid hash has the lowest cost. Some researchers are still unsure. For example, Cheng et al. (1991) claimed that when memory is large the sort-merge and hybrid hash join algorithms have similar disk I/O performance. However, the hybrid hash algorithm is regarded as one of the best algorithms for performing the join. A survey of join algorithms appears in Mishra and Eich (1992).

Each of the afore-mentioned articles typically uses the number of pages transferred in its description of the cost of disk operations. This assumes that the cost of transferring a number of consecutive pages at once is the same as transferring them individually from random parts of the disk. Hagmann (1986) argued that, for current disk drive technology, when a small number of pages is transferred, the cost of locating the pages is much greater than the cost of transferring them. He analysed the nested loop algorithm counting only

e-mail: * evan@cs.mu.oz.au
** rao@cs.mu.oz.au
Correspondence to: E.P. Harris

the number of disk accesses. When minimised, he showed that half the number of pages of the memory buffer should be devoted to each relation. Under the previous cost model, the inner relation is provided with one page of memory, and the remaining memory is devoted to the outer relation. The original cost model is still widely used, for example, in Omiecinski (1991), Omiecinski and Lin (1992), Walton et al. (1991) and Hua and Lee (1991). Our analysis is a generalisation of these two cost models and allows the relative, or absolute, cost of each disk and CPU operation to be specified.

Others have used a similar cost model to ours when evaluating their algorithms, for example, Pang et al. (1993). However, they do not attempt to optimise the buffer usage based on this information and often read a page at a time from disk during each I/O operation. Graefe (1993) noted the importance of reading and writing clusters of pages. He stated that, for sorting, "the optimal cluster size and fan-in basically do not depend on the input size." This implies that the cluster size should be a small multiple of the page size. In his example, a cluster size of 10 pages was optimal, while a cluster size of 7 pages produced a similar result to the optimal cluster size. In Sect. 5, we show experimentally, using the GRACE hash join algorithm, that a similar cluster size produces results which are not close to optimal. We believe that a minimal buffer allocation should be calculated rather than using a single ad hoc cluster size for all joins.

In this article, we present algorithms to reduce the cost of performing a join by searching for an optimal buffer size. We refer to a set of buffer sizes as a buffer allocation. When we refer to a minimal buffer allocation we are referring to a local minima. However, when we use the word "optimal", we are referring to the global minimum. In Sect. 4.5 we provide results in which, for all the tests we performed, the minimal buffer allocation was the optimal buffer allocation. However, we have no proof that this will always be the case.

The use of an extent-based file system, even under UNIX (McVoy and Kleiman 1991), will provide greater support for our technique than standard file systems which do not guarantee that consecutive pages are even on the same part of the disk. Although standard file systems do typically try to cluster contiguous pages, extent-based file systems achieve this to a greater degree. We will show that simply using one of these file systems alone does not produce optimal results.

Using our cost model, we will demonstrate that the cost of calculating a minimal buffer allocation and then performing the join using the GRACE hash join algorithm is significantly superior to the standard version of the hybrid hash join algorithm. It is usually superior to the same operations using the hybrid hash join algorithm with a minimised buffer allocation when the amount of main memory not large. The hybrid hash join algorithm is generally regarded as having the lowest cost of all join algorithms when the relation sizes are larger than the memory in which the join is to take place. When the time taken to calculate the minimal buffer allocation is taken into account this is not usually the case, unless the relation size is a small multiple of the size of main memory (typically up to three to six times).

In the next section we present four join algorithms, the nested block, sort-merge, GRACE hash and hybrid hash algorithms. Each join algorithm is described and analysed. In Sect. 3 a generalisation of the two hash methods is described and we show how to maximise the buffer usage to reduce the number of disk seeks. Minimisation algorithms are described for some of the join algorithms in this section. In Sect. 4 an analysis of expected results is presented and in Sect. 5 some experimental results are reported. In Sect. 6 we discuss how non-uniform data distributions may be handled, in Sect. 7 we discuss multiple joins, in Sect. 8 we discuss parallelism, and in the final section we present our conclusions.

2 Join algorithms

In the following analyses of the nested block, sort-merge, GRACE hash and hybrid hash join algorithms we make a number of assumptions. In common with the articles mentioned previously, we assume that the distribution of records to partitions is uniform for the join algorithms based on hashing. In Sect. 6 we describe a method which will work when the data is not uniformly distributed.

We assume that a small amount of memory is available, in addition to that provided for buffering the pages from disk. For example, we allow an algorithm to require a pointer or two for each page of memory. This additional memory will typically be thousands of times smaller than the size of the buffer.

The notation used in the analysis of the cost of each algorithm is given in Table 1. A join operation consists of taking two relations, R_1 and R_2, and producing a result relation, R_R. We denote the number of pages of a relation R_r as V_r. We assume, without loss of generality, that $V_1 \leq V_2$.

We denote the total number of pages in memory available for performing the join as B. Each join operation divides this memory up into different numbers of pages for performing different parts of the operation. For example, the nested block join requires part of the memory for each of the three relations. These are denoted B_1, B_2 and B_R, and their sum is usually the total number of pages available $B_1 + B_2 + B_R = B$. Similarly, the partitioning phase of the hybrid hash algorithm divides the memory into blocks of pages for reading a relation into B_I, and writing a number P of partitions out through B_P, while using some memory B_H for a hash table to join the records.

We denote the time taken to perform an operation x as T_x. Each operation is a part of one of the join algorithms, such as transferring a page from disk to memory, or partitioning the contents of a page. Table 1 contains the default values used to calculate the results below. The disk times, T_K and T_T, were based on a disk drive with 8 KB pages, an average seek time of 16 ms, and which rotates at 3600 RPM. The CPU times, and sorting constant, were based on the operations which would be performed on a Sun SPARCstation 10/30.

We denote the total cost of an operation x as C_x. These operations are the cost of a join algorithm, such as the cost of the nested block algorithm, C_{NB}, or a significant part of a join algorithm, such as the cost of the partitioning phase of the GRACE hash join algorithm, $C_{Partition}$.

The cost of locating a page on disk, T_K, would typically be the sum of the average seek time and the average latency time. However, the maximum seek and latency times could

Table 1. Significant notation used in cost formulae

Symbol	Description	Value
V_1	Number of pages in relation R_1	
V_2	Number of pages in relation R_2	
V_R	Number of pages in result of joining relations R_1 and R_2	
B	Number of pages available in memory for use in buffers	
B_1	Number of pages in memory for relation R_1	
B_2	Number of pages in memory for relation R_2	
B_R	Number of pages in memory for result	
B_H	Number of pages in memory used for a hash table	
B_I	Number of pages in memory used for an input buffer	
B_P	Number of pages in memory used for each partition	
P	Number of partitions created on each pass	
ρ	Number of passes during the partitioning phase	
T_C	Cost of constructing a hash table per page in place in memory	(0.015)
T_J	Cost of joining a page with a hash table in memory	(0.015)
T_K	Cost of moving the disk head to a page on disk	(0.0243)
T_M	Cost of merging a page with another in the sort-merge algorithm	(0.0025)
T_P	Cost of partitioning a page in memory	(0.0018)
T_S	Cost of sorting a page in memory	(0.013)
T_T	Cost of transferring a page from disk to memory	(0.00494)
k_S	Sorting constant	(0.00144)
C_{NB}	Cost of the nested block join algorithm	
C_{SM}	Cost of the sort-merge join algorithm	
C_{GH}	Cost of the GRACE hash join algorithm	
C_{HH}	Cost of the hybrid hash join algorithm	

be used if desired, giving an upper bound on the cost of each operation.

We assume that the cost of a disk operation, transferring a set of V_x disk pages from disk to memory, or from memory to disk, can be given by

$$C_x = T_K + V_x T_T. \tag{1}$$

The cost of n disk operations, each transferring V_x disk pages, is nC_x. We further assume that the disk head is repositioned between consecutive reads and writes.

Equation 1 appears to assume that the data is stored contiguously on disk. This is not the case when the size of the data file is large. Additional seeking may occur within the file, providing that the chances of this occurring is the same throughout the file. That is, if there are n additional seeks in V_x pages, then there are $2n$ additional seeks in $2V_x$ pages. If this is the case, then T_T can be composed of the time taken to transfer a page plus the average seeking cost between consecutive pages within the file. Although extra seeks may be required, the time taken to do this is usually very small (smaller than the average seek time for the disk), due to better storage allocation by the underlying file system.

Invalidating the assumption that the disk head is not repositioned between reads and writes would result in a lower cost, because the number of seeks (or the time taken by each one) would be reduced. Removing this assumption requires knowledge of how the disk will be used by the join algorithm and other processes which may be running on the machine. As this is often not feasible, we use the average seek and latency times as a basis for our calculations.

Equation 1 is a generalisation of the commonly used cost model that each disk operation consists of transferring a single page. We can model this by setting $T_K = 0$, $T_T = 1$ and $V_x = 1$. Equation 1 also generalises the cost model that any number of pages can be transferred at the same cost. This can be modelled by setting $T_K = 1$ and $T_T = 0$.

Using Eq. 1, we can derive the cost of transferring a set of V_x disk pages from disk to memory, or from memory to disk, through a buffer of size B_x. It is given by

$$C_{I/O}(V_x, B_x) = \left\lceil \frac{V_x}{B_x} \right\rceil T_K + V_x T_T. \tag{2}$$

The cost in Eq. 2 is used in all the join costs given below.

2.1 Nested block

The nested block join algorithm is a more efficient version of the nested loop join algorithm. It is used in a paged disk environment. The nested loop algorithm works by reading one record from one relation, the outer relation, and passing over each record of the other relation, the inner relation, joining the record of the outer relation with all the appropriate records of the inner relation. The next record from the outer relation is then read and the whole of the inner relation is again scanned, and so on.

The nested block algorithm works by reading a block of records from the outer relation and passing over each record of the inner relation (also read in blocks), joining the records of the outer relation with those of the inner relation. Historically, as much of the outer relation is read as possible on each occasion. If there are B pages in memory, $B-2$ pages are usually allocated to the outer relation, one to the inner relation, and one to the result relation. In Hagmann's analysis (Hagmann 1986) half the available memory is devoted to pages from the inner relation, and half to the outer relation.

The performance of this algorithm can be improved by *rocking* backwards and forwards across the inner relation. That is, for the first block of the outer relation the inner relation is read forwards and for the second block of the outer relation the inner relation is read backwards. This eliminates the need for reading one set of blocks of the inner relation from disk at the start of each pass, except the first, because

the blocks will already be in memory. We use this version of the algorithm in our analysis.

We assume that the memory based part of the join is based on hashing. That is, a hash table is created from the pages of the outer relation, and the records of the inner relation are joined by hashing against this table to find records to join with.

As described above, the total available memory, B pages, is divided into a set of pages for each relation, B_1, B_2 and B_R. The general constraints that must be satisfied are:

- The sum of the three buffer areas must not be greater than the available memory: $B_1 + B_2 + B_R \leq B$.
- The amount of memory allocated to relation R_1 should not exceed the size of relation R_1: $1 \leq B_1 \leq V_1$.
- The amount of memory allocated to relation R_2 should not exceed the size of relation R_2: $1 \leq B_2 \leq V_2$.
- Some memory must be allocated to the result: $B_R \geq 1$ if $V_R \geq 1$.

As described above, $V_1 \leq V_2$, therefore relation R_1 is the outer relation. It is read precisely once, B_1 pages at a time, in $\lceil V_1/B_1 \rceil$ I/O operations. Thus, relation R_2 will be read $\lceil V_1/B_1 \rceil$ times, B_2 pages at a time. Each pass over relation R_2, except the first, reads $V_2 - B_2$ pages due to the rocking over the relation.

The cost of the nested block join algorithm is given by

$$
\begin{aligned}
C_{\textit{Read } R_1} &= C_{\textit{I/O}}(V_1, B_1) \\
C_{\textit{Create}} &= V_1 T_C \\
C_{\textit{Read } R_2 \textit{ initial}} &= C_{\textit{I/O}}(V_2, B_2) \\
C_{\textit{Join initial}} &= V_2 T_J \\
C_{\textit{Read } R_2 \textit{ other}} &= \left(\left\lceil \frac{V_1}{B_1} \right\rceil - 1 \right) C_{\textit{I/O}}(V_2 - B_2, B_2) \\
C_{\textit{Join other}} &= \left(\left\lceil \frac{V_1}{B_1} \right\rceil - 1 \right) V_2 T_J \\
C_{\textit{Write } R_R} &= C_{\textit{I/O}}(V_R, B_R) \\
C_{\textit{NB}} &= C_{\textit{Read } R_1} + C_{\textit{Create}} \\
&\quad + C_{\textit{Read } R_2 \textit{ initial}} + C_{\textit{Join initial}} \\
&\quad + C_{\textit{Read } R_2 \textit{ other}} \\
&\quad + C_{\textit{Join other}} + C_{\textit{Write } R_R}.
\end{aligned}
$$

2.2 Sort-merge

The sort-merge join algorithm works in two phases. In the first (sorting) phase, each relation is sorted on the join attributes. In the second (merging) phase, a record is read from each relation and they are joined if possible, otherwise the record with the smaller sort order is discarded and the next record read from that relation. In this way, each relation is only read once after it has been sorted.

The variant of the sort-merge algorithm, whose cost we present below, is similar to that used in the Aditi deductive database system (Vaghani et al. 1994). Instead of completely sorting each relation, each relation is divided into sorted partitions, the size of which is the size of the memory buffer. This is performed by reading B pages from a relation, sorting the pages, and writing the pages out. The next B pages are then read. This is repeated on both relations. This maximises the size of the sorted partitions without requiring each relation to be read more than once, during the sorting phase.

During the merge phase, the partitions of each relation are merged together joining the records from each relation. The final pass simultaneously merges the partitions of each relation and joins the two relations. During the merging phase, B_R pages are reserved for writing the output relation. As each partition requires at least one input page, a maximum of $B - B_R$ partitions of both relations may be created prior to the merge phase, if the merging phase is to be performed with a single pass over each partition.

During the sorting phase, the whole of the available memory B is used to sort the relations. During the merging phase, the available memory is divided into sets of pages for each partition of each relation. We assume that these are the same size for each partition of a relation, B_1 for the $\lceil V_1/B \rceil$ partitions of relation R_1, and B_2 for the $\lceil V_2/B \rceil$ partitions of relation R_2. The constraints that these variables must satisfy are:

- The sum of the buffer areas must not be greater than the available memory: $\lceil V_1/B \rceil B_1 + \lceil V_2/B \rceil B_2 + B_R \leq B$.
- Some memory must be allocated to each partition and the result: $B_1 \geq 1$, $B_2 \geq 1$ and $B_R \geq 1$ if $V_R \geq 1$.

If the time taken to sort a page of x records in memory is given by $T_S = kx \lg x$ and $k_S = kx$, then the time taken to sort n pages in memory is given by

$$
\begin{aligned}
T &= knx \lg(nx) \\
&= knx(\lg n + \lg x) \\
&= n(k_S \lg n + T_S).
\end{aligned}
$$

Thus, the time taken to sort B pages can be given by $T = B(k_S \lg B + T_S)$.

The cost of the sort-merge join algorithm is given by

$$
\begin{aligned}
C_{\textit{Sort } R_1 \textit{ I/O}} &= 2\, C_{\textit{I/O}}(V_1, B) \\
C_{\textit{Sort } R_1 \textit{ CPU}} &= \left\lceil \frac{V_1}{B} \right\rceil B(k_S \lg B + T_S) \\
C_{\textit{Sort } R_2 \textit{ I/O}} &= 2\, C_{\textit{I/O}}(V_2, B) \\
C_{\textit{Sort } R_2 \textit{ CPU}} &= \left\lceil \frac{V_2}{B} \right\rceil B(k_S \lg B + T_S) \\
C_{\textit{Merge Read}} &= \left(\left\lceil \frac{V_1}{B} \right\rceil \left\lceil \frac{B}{B_1} \right\rceil + \left\lceil \frac{V_2}{B} \right\rceil \left\lceil \frac{B}{B_2} \right\rceil \right) T_K \\
&\quad + (V_1 + V_2) T_T \\
C_{\textit{Merge CPU}} &= (V_1 + V_2) T_M \\
C_{\textit{Write } R_R} &= C_{\textit{I/O}}(V_R, B_R) \\
C_{\textit{SM}} &= C_{\textit{Sort } R_1 \textit{ I/O}} + C_{\textit{Sort } R_1 \textit{ CPU}} \\
&\quad + C_{\textit{Sort } R_2 \textit{ I/O}} + C_{\textit{Sort } R_2 \textit{ CPU}} \\
&\quad + C_{\textit{Merge Read}} + C_{\textit{Merge CPU}} \\
&\quad + C_{\textit{Write } R_R}.
\end{aligned}
$$

This analysis assumes that, at most, $B - B_R$ partitions are created during the sort phase. For large amounts of memory this is likely to be true under normal circumstances, and is certainly enough to compare the sort-merge join algorithm with the other join algorithms presented in this article.

For example, consider a memory buffer of size 16 MB, ignoring the final output buffer. If we assume 8 KB pages, this means that a maximum of $16\,384/8 = 2048$ partitions may be created which are to be merged together on the final pass, thus there are 1024 partitions for each relation. Each partition will be 16 MB in size, thus the maximum size of each relation is 16 GB if only one sorting and one merging pass is permitted. A similar analysis shows that if 64 MB of memory is available, the maximum size of each relation is 256 GB. Thus, if a large amount of main memory is available, one sorting and merging pass is likely to be sufficient.

2.3 GRACE hash

Like the sort-merge join algorithm, the GRACE hash join algorithm (Kitsuregawa et al. 1983) works in two phases. In the first (partitioning) phase, each relation is partitioned such that the partitions of one of the relations can be contained within memory. It may take a number of passes over the relations to achieve this. In the second (merging) phase, each partition of the outer relation is read into memory in turn, the corresponding partition of the inner relation is scanned and the appropriate records joined. The second phase effectively consists of a number of invocations of the nested block algorithm.

The partitioning is performed by hashing each record using the values of its join attributes and placing the record in one of the output partitions. The output partition is determined by the hash value. The same hash function is used for each relation, guaranteeing that all the records of one relation which join with any given record of the other relation are in the corresponding partition of the other relation.

In the cost formulae given below we assume that, during the partitioning phase, records are read into a buffer of size B_I and then distributed between P output buffers of size B_P. While we set the number of partitions created P to be a single value, it could vary for each of the ρ passes. If a large amount of main memory is available, one pass will typically be enough. During the merging phase, the memory buffer is divided in the same way as in the nested block join algorithm. The general constraints which must be satisfied are:

- The sum of the input and output buffer areas during the partitioning phase must not be greater than the available memory: $PB_P + B_I \leq B$.
- Some memory must be allocated as an input area: $B_I \geq 1$.
- Some memory must be allocated to each of the output partitions: $B_P \geq 1$.
- The sum of the three buffer areas during the merging phase must not be greater than the available memory: $B_1 + B_2 + B_R \leq B$.
- The amount of memory allocated to relation R_1 during the merging phase should not exceed the size of relation R_1: $1 \leq B_1 \leq V_1$.
- The amount of memory allocated to relation R_2 during the merging phase should not exceed the size of relation R_2: $1 \leq B_2 \leq V_2$.
- Some memory must be allocated to the result during the merging phase: $B_R \geq 1$ if $V_R \geq 1$.

The cost of the GRACE hash join algorithm is given by

$$
\begin{aligned}
C_{\textit{Part:Read } R_1} &= \sum_{i=0}^{\rho-1} P^i C_{\textit{I/O}}\left(\left\lceil \frac{V_1}{P^i} \right\rceil, B_I\right) \\
C_{\textit{Part:Write } R_1} &= \sum_{i=1}^{\rho} P^i C_{\textit{I/O}}\left(\left\lceil \frac{V_1}{P^i} \right\rceil, B_P\right) \\
C_{\textit{Part:Partition } R_1} &= \sum_{i=0}^{\rho-1} P^i \left\lceil \frac{V_1}{P^i} \right\rceil T_P \\
C_{\textit{Part:Read } R_2} &= \sum_{i=0}^{\rho-1} P^i C_{\textit{I/O}}\left(\left\lceil \frac{V_2}{P^i} \right\rceil, B_I\right) \\
C_{\textit{Part:Write } R_2} &= \sum_{i=1}^{\rho} P^i C_{\textit{I/O}}\left(\left\lceil \frac{V_2}{P^i} \right\rceil, B_P\right) \\
C_{\textit{Part:Partition } R_2} &= \sum_{i=0}^{\rho-1} P^i \left\lceil \frac{V_2}{P^i} \right\rceil T_P \\
C_{\textit{Merge:Read } R_1} &= P^\rho C_{\textit{I/O}}\left(\left\lceil \frac{V_1}{P^\rho} \right\rceil, B_1\right) \\
C_{\textit{Merge:Create}} &= P^\rho \left\lceil \frac{V_1}{P^\rho} \right\rceil T_C \\
C_{\textit{Merge:Read } R_2 \textit{ initial}} &= P^\rho C_{\textit{I/O}}\left(\left\lceil \frac{V_2}{P^\rho} \right\rceil, B_2\right) \\
C_{\textit{Merge:Join initial}} &= P^\rho \left\lceil \frac{V_2}{P^\rho} \right\rceil T_J \\
C_{\textit{Merge:Read } R_2 \textit{ other}} &= P^\rho \left(\left\lceil \frac{\left\lceil \frac{V_1}{P^\rho} \right\rceil}{B_1} \right\rceil - 1\right) \\
&\quad \times C_{\textit{I/O}}\left(\left\lceil \frac{V_2}{P^\rho} \right\rceil - B_2, B_2\right) \\
C_{\textit{Merge:Join other}} &= P^\rho \left(\left\lceil \frac{\left\lceil \frac{V_1}{P^\rho} \right\rceil}{B_1} \right\rceil - 1\right) \\
&\quad \times \left\lceil \frac{V_2}{P^\rho} \right\rceil T_J \\
C_{\textit{Write } R_R} &= C_{\textit{I/O}}(V_R, B_R) \\
C_{\textit{GH}} &= C_{\textit{Part:Read } R_1} \\
&\quad + C_{\textit{Part:Write } R_1} \\
&\quad + C_{\textit{Part:Partition } R_1} \\
&\quad + C_{\textit{Part:Read } R_2} \\
&\quad + C_{\textit{Part:Write } R_2} \\
&\quad + C_{\textit{Part:Partition } R_2} \\
&\quad + C_{\textit{Merge:Read } R_1} \\
&\quad + C_{\textit{Merge:Create}} \\
&\quad + C_{\textit{Merge:Read } R_2 \textit{ initial}} \\
&\quad + C_{\textit{Merge:Join initial}} \\
&\quad + C_{\textit{Merge:Read } R_2 \textit{ other}}
\end{aligned}
$$

$$+C_{Merge:Join\ other}$$
$$+C_{Write\ R_R}.$$

Like the sort-merge join algorithm, this algorithm is likely to only require one pass over each relation during the partitioning phase. For example, consider the same memory buffer of size 16 MB used in the example in the previous section. One partitioning pass means that $\rho = 1$. The largest size of the outer relation will occur when $P = B - 1$, thus $V_1 = (B - 1)B_1$. If we assume that most of the 2048 pages (16 MB) are allocated to B_1, the size of the smaller relation, relation R_1, will be just under 32 GB. The other relation may be much larger. By a similar analysis, if 64 MB (8192 pages) of memory is available, the maximum size of the smaller relation will be just under 512 GB. In practice, a lower cost may be found by making two passes to partition the data when the relations are this large because partitioning relations of this size in one pass requires that $B_P = 1$. This is usually not optimal, as we discuss in Sect. 3.2.1.

2.4 Hybrid hash

The hybrid hash join algorithm (DeWitt et al. 1984; Shapiro 1986) is very similar to the GRACE hash join algorithm. The primary difference is that the hybrid hash join algorithm reserves an area of memory to join records in during the partitioning phase. Instead of hashing each record into one of P partitions during the partitioning phase, each record is hashed into one of $P+1$ partitions. During the partitioning of relation R_1, records which hash into the extra partition are not written out to disk, but are stored in a hash table in the reserved area in memory. When relation R_2 is partitioned, records which hash into the extra partition are joined with the records of relation R_1 which are stored in memory. The amount of memory reserved for this extra partition need not be the same as the expected sizes of the other partitions, providing that the extra partition does not overflow during one pass of relation R_1.

The basis of the cost of the hybrid hash join algorithm is similar to the GRACE hash join algorithm, with the addition of the hash table in memory, which has size B_H, during the partition phase. The constraints and costs for the hybrid hash algorithm are given in Appendix A.

3 Minimising costs

The equations in Sect. 2 describe the cost of each join algorithm. There are a number of methods which could be used to minimise these algorithms, including combinatorial and heuristic techniques. We now describe how we determine a minimal buffer allocation for the nested block algorithm and the join algorithms based on hashing.

3.1 Nested block

To minimise the cost of the nested block join algorithm we minimise C_{NB} in the presence of two variables, B_1 and B_2. We then set $B_R = B - B_1 - B_2$. For a minimisation method to be useful in practice, its running time must be short, relative to the time taken by the join. To determine how to find the minimum, we have plotted the cost of the join versus B_1 in Fig. 1. Similar graphs are produced for any values of V_1, V_2, V_R and B in which the nested block join is used. In Fig. 1, the value of B_R is a constant so that the behaviour of the cost variation may be easily observed. However, the shape of the graph is the same if B_R is varied. The values of T_K and T_T are based on the Wren 6 disk drive, and T_C and T_J were based on the operations which may be performed on a SPARCstation 10/30 using an 8 KB page size. These values were shown in Table 1.

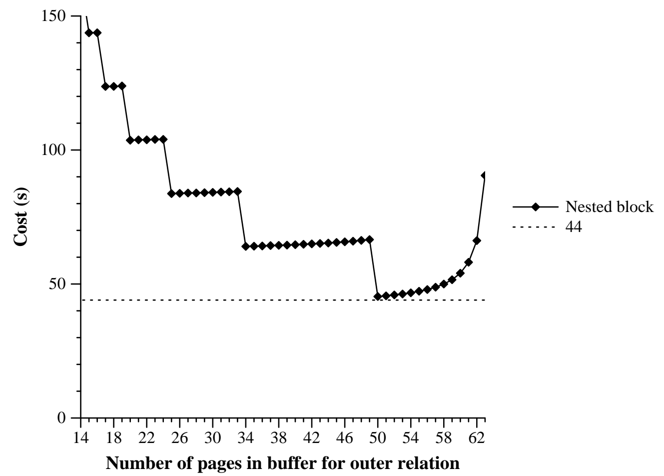

Fig. 1. Cost of nested block join algorithm as B_1 and B_2 vary. $V_1 = 100$, $V_2 = 1000$, $V_R = B_R = 1$, $B = 65$

The minimum value of C_{NB} is likely to occur when memory is well utilised. Memory is better utilised when the values of the variables B_1, B_2 and B_R are such that V_1/B_1, V_2/B_2, and V_R/B_R are integers. That is, the size of the buffer allocated to a relation exactly divides the size of the relation. This ensures that no pages in the buffer are wasted as a relation is read in. For example, consider a relation of size 100 pages. Assume that the size of the buffer allocated to this relation may be 10 or 11 pages. To read the relation will take $\lceil 100/10 \rceil = 10$ or $\lceil 100/11 \rceil = 10$ read operations, respectively. Clearly, the cost of the disk operations will be the same. However, if the relation is allocated 11 pages, 10 pages are not used during the final read operation. It is more efficient for the relation to be allocated 10 pages and allow the other page to be allocated to the other relation, or to the result relation.

Our minimisation algorithm, shown in Fig. 2, works by stepping down from the largest values of B_1 and B_2 until the cost is greater than the minimum cost found by a certain factor, δ. It initially sets $B_1 = \lceil V_1/i \rceil$, where i is the smallest integer such that $B_1 \leq B - 2$, and $B_2 = \lceil V_2/j \rceil$, where j is the smallest value such that $B_1 + B_2 \leq B - 1$. It sets $B_R = B - B_1 - B_2$ and calculates the cost, then increments j and re-calculates B_2, B_R and the cost. This process continues while the cost is less than δ multiplied by the minimum cost found for this value of B_1. The final cost is saved, i is incremented, and B_1 is re-calculated. This process continues while the best cost for each value of B_1 is less than δ multiplied by the minimum cost.

```
function minimiseNB(V_1, V_2, V_R)
    mincost ← ∞, B'_1 ← 0, i ← 1
    while B'_1 ≠ 1 do
        B_1 ← ⌈V_1/i⌉                                   # find the largest integer (almost) dividing V_1
        if B_1 ≤ B − 2 ∧ B'_1 ≠ B_1 then
            runcost ← ∞, B'_2 ← 0, j ← 1                # a fixed value for B_1 is a "run"
            while B'_2 ≠ 1 do
                B_2 ← ⌈V_2/j⌉                           # find the largest integer (almost) dividing V_2
                if B_2 ≤ B − B_1 − 1 ∧ B'_2 ≠ B_2 then
                    B_R ← B − B_1 − B_2
                    cost ← C_NB(V_1, V_2, V_R, B_1, B_2, B_R)
                    B'_2 ← B_2                          # save B_2 to ensure we don't try it twice
                    if cost < runcost then
                        (B+_1, B+_2, B+_R, runcost) ← (B_1, B_2, B_R, cost)     # save run best
                    else if cost > δ× runcost then
                        break                           # this cost is much worse, so end this run
                    end if
                end if
                j ← j + 1                               # prepare for next value of B_2
            end while
            B'_1 ← B_1                                  # save B_1 to ensure we don't try it twice
            if runcost < mincost then
                (B*_1, B*_2, B*_R, mincost) ← (B+_1, B+_2, B+_R, runcost)       # save overall best
            else if runcost > δ× mincost then
                break                                   # this cost is much worse, so end
            end if
        end if
        i ← i + 1                                       # prepare for next value of B_1
    end while
    return mincost, B*_1, B*_2, B*_R
end function
```

Fig. 2. Function for minimising the cost of the nested block join algorithm

It can be shown that the worst case complexity of this algorithm is $O(B^{3/2})$, assuming that $V_1 \leq \alpha B$, where α is a small constant which is usually less than four. If this condition does not hold, the nested block algorithm will perform worse than the other algorithms, such as the GRACE and hybrid hash joins, and should not be used. We describe how the minimisation algorithm performs in Sect. 4. The results show that the computation time required to find the minimal buffer allocation is insignificant. In our tests, it always ran in less than 0.05% of the execution time of the join.

3.2 A generalised hash join algorithm

If the constraints on the hybrid hash join algorithm are relaxed so that $B_H = 0$ is permitted, and by removing the result buffer B_R during the partitioning phase when $B_H = 0$, the hybrid hash join algorithm can be generalised to include the GRACE hash join algorithm. Both of these algorithms, as described in Sect. 2, have separate input and output buffer areas during the partitioning phase. These two areas can be combined without affecting the cost equations C_{GH} (Sect. 2.3) and C_{HH} (Appendix A) after altering the constraints on each algorithm.

Figure 3 diagrammatically represents our scheme. A set of $2P - 1$ one page buffers is reserved along with the B_R and B_H pages. The remaining PB_P pages are used as both the input and output buffers during the partitioning phase. A relation is read into the PB_P pages, then it is partitioned across the $PB_P + 2P - 1$ pages. It effectively tries to partition in place using the PB_P pages. Pages which are not completely filled are moved to one of the $2P-1$ spare pages based upon its partition number. On the next pass, the contents of each spare page are added to each partition after partitioning but before the full pages are written out. Incompletely filled pages are again saved within the $2P - 1$ spare pages. Thus, only complete pages are written out, except after final read when all pages must be written.

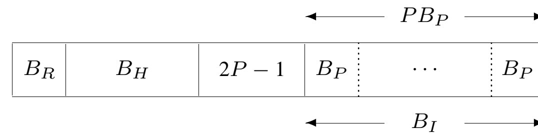

Fig. 3. Buffer structure of modified hash join algorithm during partitioning

The number of spare pages which are required is, at most, $2P - 1$.

- Just before reading each set of pages from disk, at most P pages can contain records. This is because each output partition has only one partially filled page active at any point in time.
  All the other pages are considered empty, because pages which were considered full have been just written out, and so can be considered to be empty. Therefore, P spare pages must be available in addition to the ones used for buffering the relation after a read operation.

– Any set of N pages can be partitioned in place into P partitions using $N + P - 1$ pages.
  Each of the records must be moved into one of the P partitions. This requires P output pages. However, records are never added to one of the P partitions faster than they are taken out of the N pages. Therefore, one of the N pages can be used as an output partition, and at most only $P - 1$ extra pages are required.

The number of pages read from disk during each read operation is PB_P. Combining the previous two points, we can see that the number of spare pages which are required is $2P - 1$.

As the number of pages transferred under this scheme is the same as the algorithms described in Sect. 2, the number of seeks will be the same if the constraints on P, B_P, B_I, B_H and B_R are modified appropriately. The cost functions of Sect. 2.3 and Appendix A are still valid. The constraints and minimisation algorithm for the modified GRACE hash algorithm are presented in the next section, and the constraints for the modified hybrid hash algorithm are discussed in Appendix A.

3.2.1 Modified GRACE hash

The buffer arrangement of our modified GRACE hash join algorithm during the partitioning phase is the same as that described in Fig. 3, if B_R and B_H are ignored. For the cost C_{GH} to be valid, the constraints that change are:

– $PB_P + 2P - 1 \leq B$ instead of $PB_P + B_I \leq B$.
– $B_I = PB_P$ instead of $B_I \geq 1$.

The other constraints remain valid.

We now consider the maximum practical size of the relations under this scheme before two passes must be made over the data to partition it. If we assume a 16 MB buffer area, an 8 KB page size, and CPU times as given in Table 1, the largest relation which will be partitioned in one pass is around 15.9 GB in size. For relations larger than this, it is better to make two passes over the relation in which two pages are allocated to each output partition, than it is to make only one pass where each output partition is only allocated one page. Note that this is similar to the capacity of the sort-merge algorithm when only one pass is permitted.

To minimise the cost of the modified GRACE hash join algorithm, we must minimise C_{GH} in the presence of four variables, B_1, B_2, P and ρ. We set $B_R = B - B_1 - B_2$, $B_P = \lfloor (B - (2P - 1))/P \rfloor$ and $B_I = PB_P$. To determine how to find the minimum we have plotted the cost of the join versus B_1 in Fig. 4, in the same way that we did with the nested block algorithm. The resulting graph is very similar to the graphs produced using the nested block algorithm, as shown in Fig. 1.

We use a minimisation algorithm for the GRACE hash join method similar to the one which minimises the nested block join algorithm. The primary difference is that the value of B_1 is derived from the value of P and the number of passes. Instead of changing B_1 directly, the values of P and ρ are changed. The minimisation algorithm is presented in Fig. 5.

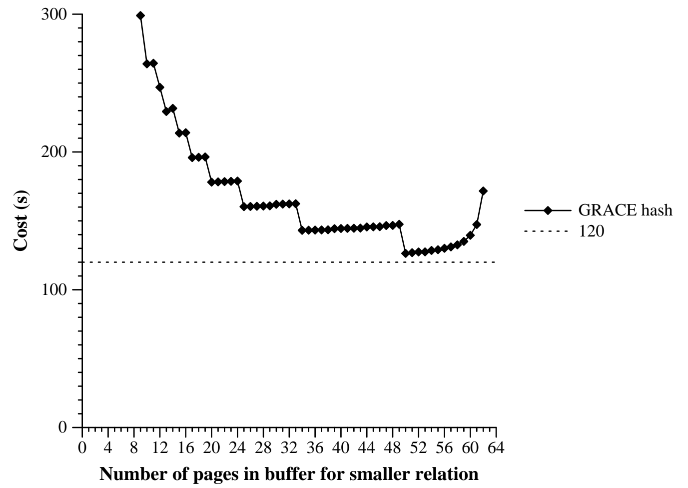

Fig. 4. Cost of GRACE hash join algorithm as B_1 and B_2 vary. $V_1 = V_2 = 1000$, $B_R = V_R = 1$, $B = 64$, $P = 10$, $B_P = 5$, $\rho = 1$

It can be shown that the upper bound on the complexity of this minimisation algorithm is $O(V_2)$, assuming that at most one partitioning pass is made over each relation. Therefore, the size of the two relations is of the form $V_1 = \alpha B^2$ and $V_2 = \beta B^2$, where $\alpha \leq \beta$ and $\alpha < 1$. The performance of this algorithm is discussed in Sect. 4. Like the nested block minimisation algorithm, the results show that the computation time required to find the minimal buffer allocation is insignificant. It is more expensive than the nested block minimisation algorithm, but the running times of the joins for which it finds minimal buffer allocations are longer. In our tests, it always ran in less than 0.05% of the execution time of the join.

4 Computational results

In this section we provide examples of the expected performance of the join algorithms under the cost model described in Sect. 2. We compare the nested block join algorithm, the GRACE hash join algorithm, and the hybrid hash join algorithm with the respective version of each algorithm commonly described in the literature. We then compare the expected performance of all four join algorithms described in Sect. 2 on a variety of joins on relations of different sizes. In Sect. 5 we report on our experimental results.

In the remaining sections, when we refer to the optimised or minimised GRACE hash (hybrid hash) algorithm we mean the modified GRACE hash (hybrid hash) algorithm described in Sect. 3.2.1 (Appendix B). When we refer to the standard versions of both algorithms, we mean the original versions of both algorithms with the original buffer allocations.

4.1 Nested block

Using the buffer arrangement which results in a minimal value of C_{NB} is obviously desirable; determining it requires some computational cost. We would like to know whether the improvement in performance makes this additional cost worthwhile. To this end, we have plotted the cost of each

```
function minimiseGH(V_1, V_2, V_R, B)
    (mincost, B_1^+, B_2^+, B_R^+) ← minimiseNB(V_1, V_2, V_R, B)
    P^+ ← 0, ρ^+ ← 0
    runcost ← ∞, ρ ← 1
    do
        prevcost ← runcost
        for i ← 2 to 1 do
            runcost ← ∞
            for P ← ⌈ ρ√(V_1/i(B+2)) ⌉ to ⌊B/3⌋ do
                B_1 ← ⌈V_1/iP^ρ⌉
                if B_1 ≤ B − 2 then
                    (cost, B_2, B_R) ← minimiseB_2(V_1, V_2, V_R, B, B_1, P, ρ)
                    if cost < mincost then
                        (mincost, runcost, B_1^+, B_2^+, B_R^+, P^+, ρ^+) ← (cost, cost, B_1, B_2, B_R, P, ρ)
                    else if cost < runcost then
                        runcost ← cost
                    else if cost > δ × runcost then
                        break
                    end if
                end if
            end for
        end for
        ρ ← ρ + 1
    until prevcost < runcost
    return mincost, B_1^+, B_2^+, B_R^+, P^+, ρ^+
end function

function minimiseB_2(V_1, V_2, V_R, B, B_1, P, ρ)
    mincost ← ∞, B_2' ← 0, i ← 1
    while B_2' ≠ 1 do
        B_2 ← ⌈ ⌈V_2/P^ρ⌉ / i ⌉          # find largest integer (almost) dividing relation size
        if B_2 ≤ B − B_1 − 1 ∧ B_2' ≠ B_2 then
            B_R ← B − B_1 − B_2
            cost ← C_GH(V_1, V_2, V_R, B_1, B_2, B_R, P, ρ)
            if cost < mincost then
                (B_2^+, B_R^+, mincost) ← (B_2, B_R, cost)      # save best values
            else if cost > δ× mincost then
                break                        # this cost is much worse, so finish
            end if
            B_2' ← B_2                       # save B_2 to ensure we don't try it twice
        end if
        i ← i + 1                            # prepare for next value of B_2
    end while
    return mincost, B_2^+, B_R^+
end function
```

Fig. 5. Functions for minimising the cost of the GRACE hash join algorithm

algorithm for a range of values of V_1 for fixed values of V_2, V_R and B. This is shown in Fig. 6.

If we assume a page size of 8 KB, $V_2 = 100\,000$ approximately corresponds to a 781 MB relation, $V_R = 10\,000$ to a 78 MB result relation, and $B = 4096$ to 32 MB of main memory used during the join operation. Figure 6 is a representative comparison of the nested block algorithm for all values of V_2, V_R and B we tested. For example, a graph of identical shape is produced when $V_2 = 1000$, $V_R = 100$ and $B = 64$.

The "Nested block (opt)" line corresponds to the minimal value of C_{NB} calculated using the minimisation algorithm in Fig. 2. Table 2 shows a number of the minimal values for B_1, B_2 and B_R for various values of V_1. The "Nested block (std)" line corresponds to the standard version of the nested block algorithm commonly described in the literature, in which $B_1 = B - 2$, $B_2 = B_R = 1$. The "Nested block (Hag)" line corresponds to the version proposed by Hagmann (1986), in which $B_1 = B_2 = (B - 1)/2$, $B_R = 1$. Note that while Hagmann's version is faster than the standard version for larger relations, it is approximately twice as slow for most of the smaller relations in which $B/2 < V_1 < B$. This is in marked contrast to the results Hagmann reported, and is due entirely to our more realistic cost model.

Our results indicate that the shape of the graphs are general and independent of values of V_2, V_R and B. The standard nested block algorithm increases its cost substantially each time the size of V_1 increases by $B - 2$. This is because an additional pass over the second relation is required at each of these points. Similarly, Hagmann's version increases its cost as the size of V_1 increases by $(B - 1)/2$, for the same reason. The minimal version continually ad-

Table 2. Minimal buffer allocation of B_1, B_2 and B_R for the nested block join algorithm when V_2 = 100 000 (781 MB), V_R = 10 000 (78 MB), B = 4096 (32 MB).

V_1	B_1	B_2	B_R
1	1	3226	869
2048	2048	1613	435
4000	4000	73	23
4096	2048	1852	196
8000	4000	79	17
8192	2731	1235	130
100000	4000	90	6

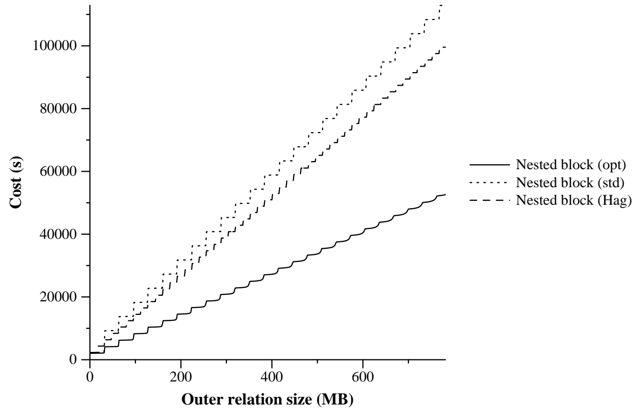

Fig. 6. Cost of the nested block join algorithm as V_1 varies. V_2 = 100 000 (781 MB), V_R = 10 000 (78 MB), B = 4096 (32 MB)

justs the size of each of the buffers B_1, B_2 and B_R for each relation to minimise the need to perform additional passes.

The results show that a significant improvement in performance is gained by using the buffer arrangement which results in the minimal value of C_{NB}. For example, when V_1 = 8000, the minimal version takes 46% of the time of the standard version and 51% of the time of Hagmann's version. When relation R_1 can be contained within the memory buffer, an improvement is still achieved by selecting better values for B_2 and B_R. In addition, the minimisation algorithm is very fast. This is discussed further in Sect. 4.5.

4.2 GRACE hash

As with the nested block join algorithm, calculating the minimal value of C_{GH} requires some computational cost. We would like to know if the improvement in performance which can be achieved makes calculating the minimal arrangement worthwhile.

We have plotted the cost of each algorithm for a range of values of V_1 for the same fixed values of V_2, V_R and B used in the previous section. The result of this is shown in Fig. 7. The "GRACE hash (opt)" line corresponds to the minimal value of C_{GH} calculated using the minimisation algorithm. The "GRACE hash (std)" line corresponds to the standard version of the GRACE hash algorithm, in which $B_1 = B - 2$, $B_2 = B_R = 1$, and $P = B - 1$.

As with the nested block algorithm, the results show that a significant improvement in performance is gained by using the buffer arrangement providing the minimal value of C_{GH}. For example, when V_1 = 12 000 (94 MB), the minimal version takes 30% of the time of the standard version, and

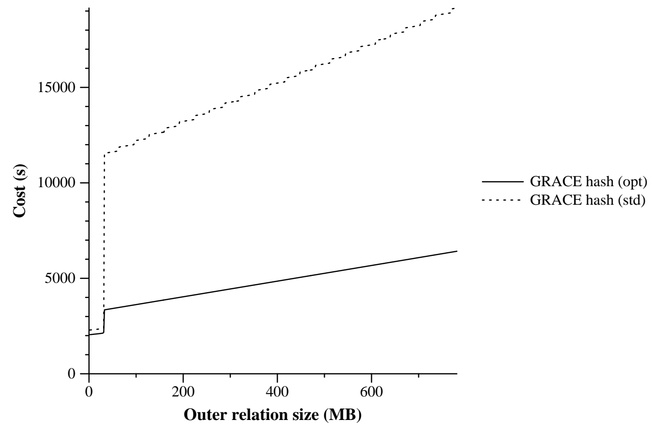

Fig. 7. Cost of the GRACE hash join algorithm as V_1 varies. V_2 = 100 000 (781 MB), V_R = 10 000 (78 MB), B = 4096 (32 MB)

when V_1 = 100 000 (781 MB) it takes 34% of the time of the standard version. Like the nested block minimisation algorithm, the time taken by the GRACE hash minimisation algorithm is small. This is discussed further in Sect. 4.5.

4.3 Hybrid hash and simulated annealing

While using the buffer arrangement which results in a minimal value of C_{HH} is desirable, determining it using a minimisation algorithm similar to the GRACE hash minimisation algorithm requires a huge computational cost, often much longer than performing the join. To attempt to reduce this, we used simulated annealing (Aarts and Korst 1989) to find a good buffer arrangement in a much shorter period of time. The parameters to the simulated annealing algorithm were chosen so that it would terminate in around 10 s, although all our tests times actually varied between 4 s and 21 s. This is discussed further in Sects. 4.4 and 4.5.

We have plotted the cost of each algorithm for a range of values of V_1 for the same fixed values of V_2, V_R and B as the nested block and GRACE hash algorithms. The result of this is shown in Fig. 8. The "Hybrid hash (opt)" line corresponds to a good value of C_{HH} calculated using simulated annealing. Simulated annealing does not guarantee to determine the minimal value at each point. However, the shape of the graph indicates that it performs well. The "Hybrid hash (std)" line corresponds to the standard version of the hybrid hash algorithm, in which $B_1 = B - 2$, $B_2 = B_R = B_P = B_I = 1$, $P = \lceil (V_1 - (B-2))/((B-2)-1) \rceil + 1$ and $B_H = B - P - B_I - B_R$.

As with the nested block and GRACE hash join algorithms, the results show that a significant improvement in performance is gained by using the buffer arrangement providing a good value of C_{HH}. For example, when V_1 = 12 000 (94 MB) the minimised version takes 34% of the time of the standard version, and when V_1 = 100 000 (781 MB) it takes 35% of the time of the standard version.

4.4 Join Algorithm Comparison

We have seen that using a more realistic cost model for determining memory buffer usage for the nested block, GRACE

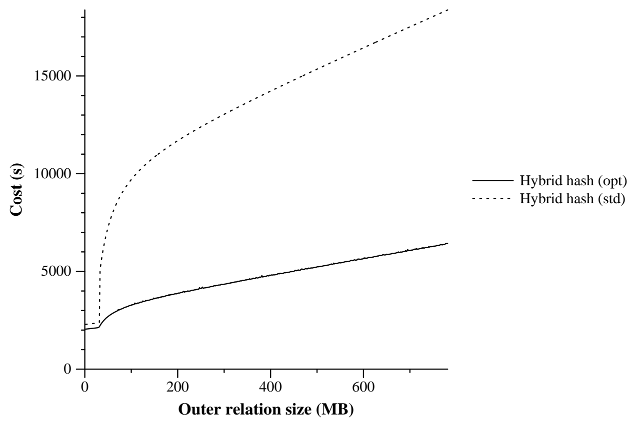

Fig. 8. Cost of the hybrid hash join algorithm as V_1 varies. V_2 = 100 000 (781 MB), V_R = 10 000 (78 MB), B = 4096 (32 MB)

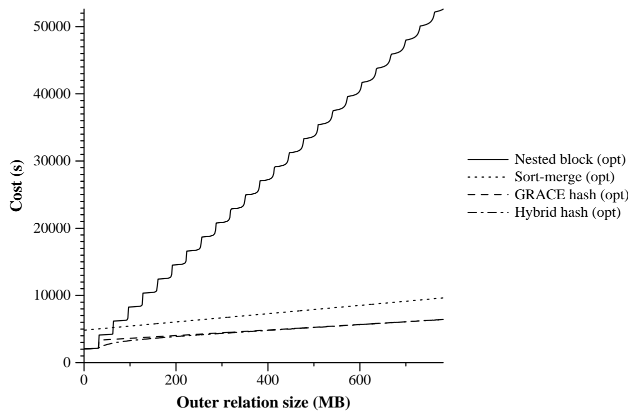

Fig. 9. Join algorithm comparison as V_1 varies. V_2 = 100 000 (781 MB), V_R = 10 000 (78 MB), B = 4096 (32 MB)

hash and hybrid hash join algorithms can result in a significant improvement in the performance of the join algorithm. We now compare the four join algorithms, using the same example as above. The result of this is shown in Figs. 9 and 10. Figure 10 contains an enlarged version of part of Fig. 9.

Using the standard cost model, others (including Blasgen and Eswaran 1977; DeWitt et al. 1984; DeWitt and Gerber 1985; Shapiro 1986) reported that *when the outer relation may be contained within main memory, the nested block algorithm performs the best*. Figures 9 and 10 show that this is still the case under our cost model. Note that our definitions of the GRACE and hybrid hash algorithms presented in Sect. 2 reduce to the nested block algorithm when no partitioning passes are made over the data; hence the costs are the same. As the size of the outer relation gets larger, the other join algorithms all perform better than the nested block algorithm.

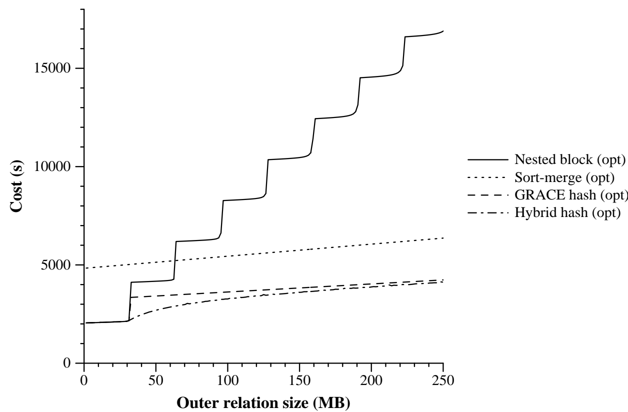

Fig. 10. Join algorithm comparison as V_1 varies. V_2 = 100 000 (781 MB), V_R = 10 000 (78 MB), B = 4096 (32 MB)

Table 3. Range of random values for V_1, V_2, V_R in pages

Variable	Minimum		Maximum	
V_1	97	(0.8 MB)	199 627	(1560 MB)
V_2	11 926	(93 MB)	399 450	(3121 MB)
V_R	60	(0.5 MB)	256 119	(2001 MB)

The result that is different to that reported in the past using the standard cost model, such as by DeWitt et al. (1984), is that, in general, *the GRACE hash algorithm performs as well as the hybrid hash algorithm for large relations relative to the size of main memory*. The size of a large relation is defined to be some small multiple of the size of main memory (for example, three or ten times). When the relation size is larger than the buffer size but still small, the hybrid hash algorithm performs better than all the other algorithms. This is because a large percentage of the relations may be joined during the first pass over the data. Thus, the amount of data which does not have to be written to disk, read back in, and then joined, will be large enough to ensure that the cost of the hybrid hash algorithm is significantly smaller than that of the other methods.

Using the standard cost model, as reported in DeWitt (1984), the sort-merge algorithm does not perform as well as either hash-based algorithm, despite the fact that our version of the sort-merge algorithm has a lower cost than the version reported.

4.5 Join algorithm comparison: minimisation times

In the previous four sections, we have shown that the minimal versions of each of the algorithms will perform as well or better than the standard versions. Therefore, whether the minimal versions are the best to use in practice will depend on the time taken to find the minimal buffer allocation in each case. The sum of the time taken to determine the minimal buffer allocation and then to execute the join, must be faster than simply using the standard version of each algorithm for this scheme to be worthwhile.

To determine the likely relative performance of each join and minimisation algorithm, we generated 1000 random join queries. For each query, the value of B was randomly chosen to be one of 128, 256, 512, 1024, 2048, 4096 or 8192. If we assume pages are 8 KB, this tests main memory sizes from 1 MB to 64 MB. The values of V_1, V_2 and V_R were chosen randomly such that $V_1 \leq V_2$. The extreme values for these variables are shown in Table 3.

For each of these queries, the minimisation algorithm for the nested block and GRACE hash join algorithms were used to determine the minimal values. Simulated annealing was used for the extended version of the modified hybrid hash algorithm which generalises both the hybrid hash and the

Table 4. Number of minimal buffer allocations for each join algorithm. NB, nested block; SM, sort-merge; GH, GRACE hash; HH, hybrid hash

Memory size B	Total number of joins	Number of minimal costs for each join algorithm Excluding minimisation time NB	SM	GH & HH	GH	HH	Including minimisation time NB	SM	GH & HH	GH	HH
128	139	0	0	6	127	6	0	0	0	136	3
256	136	0	0	2	131	3	0	0	0	133	3
512	131	0	0	2	124	5	0	0	0	127	4
1024	142	1	0	0	123	18	1	0	0	123	18
2048	132	4	0	0	94	34	4	0	0	100	28
4096	156	2	0	0	59	95	2	0	0	68	86
8192	164	5	0	0	6	153	5	0	0	7	152

GRACE hash algorithms. This was done so that the GRACE and hybrid hash algorithms could be compared equally, and to see if simulated annealing was, in fact, determining a minimal buffer allocation. An exhaustive search was also performed for the sort-merge algorithm, and for $B = 128$ and $B = 256$ for the nested block and GRACE hash algorithms to determine the optimal values.

The performance of the nested block and GRACE hash minimisation algorithms was very good. The nested block and GRACE hash minimisation algorithms always found the optimal value for the buffer sizes $B = 128$ and $B = 256$, and always ran in less than 0.05% of the time to perform the join. The buffer allocation found for each of the 1000 joins was different from the standard buffer allocation in all cases. While it is possible that the standard buffer allocation may be optimal for some joins, we believe that there are very few joins for which this is the case. Thus, we believe that these minimisation algorithms should be used in practice when executing joins, particularly given the improvements demonstrated in Sects. 4.1 and 4.2.

The running time of the simulated annealing minimisation of the extended hybrid hash algorithm was also small. It took up to 3% of the time it would take to both use simulated annealing to determine the buffer allocation and then to perform the join. Unfortunately, simulated annealing did not always find the minimal buffer allocation. We know this, because, for a number of the queries the cost of the hybrid hash algorithm using the buffer allocations determined using simulated annealing was higher than the cost of the buffer allocations found GRACE hash minimisation algorithm. As the simulated annealing algorithm was minimising a cost function which generalised the GRACE hash join method in addition to the hybrid hash join method, the minimal cost should be less than, or equal to, that found by the GRACE hash minimisation algorithm.

Table 4 summarises the performance of each of the minimisation algorithms. In theory, either the nested block or hybrid hash join algorithms should provide the minimal buffer allocation for all joins. However, our results differ from this expectation. We found that of the 1000 joins, the nested block join algorithm was the best algorithm to use in 12 of the joins based upon execution time alone, and was still the best in those 12 cases when the time taken to determine the minimal buffer allocation was also considered. The GRACE and hybrid hash algorithms determined minimal buffer allocations with the same cost for 10 queries, and differed for the remaining 978 queries. The results in Table 4 show that the simulated annealing minimisation algorithm for the extended hybrid hash join algorithm, which encompasses both the hybrid hash join method and the GRACE hash join method, needs to be improved significantly before it will be generally useful across all memory buffer sizes. The GRACE hash minimisation algorithm often performs better than the hybrid hash join algorithm minimised using the simulated annealing algorithm, particularly for smaller buffer sizes.

Interestingly, for 258 of the 988 join queries in which algorithms other than the nested block algorithm produce the minimal cost, the hybrid hash algorithm minimised using simulated annealing produced a minimal value such that $B_H = 0$. That is, the GRACE hash method resulted in a lower cost than the traditional hybrid hash method in which $B_H > 1$ is a constraint. Although the minimal hybrid hash algorithm reduced to the GRACE hash algorithm, it is possible for the simulated annealing algorithm to produce a different buffer allocation. This was observed on 250 of the 258 occasions. In all these cases we found that simulated annealing produced a buffer allocation with a higher cost. The difference in cost was usually small; often less than 0.1% of the cost. However, differences up to 6.3% were observed. On the remaining 8 of the 258 occasions, the buffer allocation determined by the simulated annealing algorithm produced identical buffer allocations with the same cost as that of the GRACE hash minimisation algorithm. This indicates that although the simulated annealing algorithm produces good results, particularly for smaller values of V_1, there is scope for developing a better algorithm for minimising the cost of the hybrid hash join.

The expected performance of the sort-merge algorithm reinforced the results shown in Fig. 9. Due to the restricted version of the sort-merge algorithm we examined, only a limited number of joins were appropriate. Of the 1000 joins, 343 were too big to sort the relations in one pass. In another 12 joins the nested block algorithm produced the minimal value. The sort-merge algorithm had a higher cost than the GRACE hash algorithm for all the remaining joins. The magnitude of the additional cost varied between 24% and 153% of the cost of the GRACE hash algorithm, with an average of 52%. The primary advantage of the sort-merge algorithm is that it avoids the problem of uneven partition sizes which can affect the performance of the hash join algorithms. We discuss how this problem can be reduced for the hash join algorithms in Sect. 6.

In 294 of the 1000 join queries, the hybrid hash algorithm minimised using simulated annealing performed the best. Table 5 summarises the improvement of the hybrid hash algorithm over the GRACE hash algorithm. Figure 4

Table 5. Percentage improvement of hybrid hash over GRACE hash when hybrid hash has a lower cost, including minimisation time

Memory size, B	Total joins	Percentage improvement Min	Median	Mean	Max
128	3	0.04	0.08	0.61	1.70
256	3	0.76	4.08	3.60	5.96
512	4	2.10	5.67	8.01	20.38
1024	18	0.11	1.46	3.81	15.39
2048	28	0.00	0.94	4.62	31.42
4096	86	0.02	1.27	3.81	28.66
8192	152	0.11	3.68	15.15	84.14

shows that when $B \leq 2048$, the GRACE hash minimisation algorithm is much more likely to produce a better buffer allocation than the hybrid hash minimisation algorithm.

In Table 5 we can see that for large main memories ($B = 8192$) and large relations it is desirable to use simulated annealing and the extended hybrid hash algorithm. Conversely, when the amount of memory is smaller, we believe that the minimisation algorithm for the GRACE hash join algorithm should be used to determine the minimal buffer allocation for any join. The results in Tables 4 and 5 do not contradict the results in Fig. 9. The reason that the hybrid hash algorithm performs so much better for larger memory sizes is that the range of sizes of the relations being joined does not change, so the ratio of relation size to main memory size decreases for larger main memory sizes. Figure 9 shows that even when the outer relation is 20 times greater than main memory, an improvement is possible, albeit small. This is reflected in the experiments shown in Table 5. We found that the cost improvement of the hybrid hash algorithm over the GRACE hash algorithm across all 1000 joins was less than 2.2% in 50% of the cases, less than 5.1% in 70% of the cases, and less than 30% in 90% of the cases.

Instead of using a random starting point for the simulated annealing algorithm, we tried using the buffer allocation produced by the GRACE hash minimisation algorithm in a single simulated annealing trial. This method proved cost effective for 881 of the 1000 join queries. That is, the improvement in the cost was greater than the time taken for the single run of the simulated annealing algorithm (which was all less than 1 s of CPU time) for 881 join queries, with an average improvement of 8.6% across those 881 queries.

Of the 881 join queries for which a lower cost was found, the cost determined was lower than that determined by the simulated annealing algorithm described above for 807 of the joins, and greater for the other 74. Thus we believe that seeded simulated annealing is likely to be a better method to use than the normal simulated annealing algorithm with random starting points when minimisation time is significant.

Another possible method of determining the optimal buffer allocation for the hybrid hash algorithm would be a combination of storing pre-computed optimal buffer allocations for various input and output relation sizes, in conjunction with a (small) search around that optimal buffer allocation for given relation sizes. Storing all possible combinations of relation sizes is unlikely to be possible, unless the sizes of potential relations is severely constrained. Additionally, in a multi-user system, the amount of memory available as buffer space is likely to vary depending on the system load. Thus, sets of relation and buffer sizes would have to be stored for each different amount of memory available. This is likely to be impractical.

The storage of only unique optimal buffer allocations is also likely to be impractical. With four distinct variables (B_1, B_R, B_H and P, deriving B_2 and B_P), the number of possible buffer allocations is $O(B^4)$. If the buffer size can also vary, it clearly becomes impractical to store buffer allocations for many values of B. An open problem remains to determine how many buffer allocations to store and how to efficiently derive an optimal allocation from them.

In conclusion, we have seen that a combination of the nested block and GRACE hash join algorithms, and their respective minimisation algorithms, provide a significant improvement over the standard versions of all the join algorithms we have examined. In all the cases in which the optimal buffer allocation was calculated, the minimal buffer allocations produced by the nested block and GRACE hash minimisation algorithms were optimal. In addition, all the minimised versions of the GRACE hash join, hybrid hash join and sort-merge join provide a significant improvement over the best of the standard versions of the algorithms, namely the standard hybrid hash join. We believe that these join algorithms should be implemented. To determine a good buffer allocation for the hybrid hash join method, the seeded simulated annealing algorithm should also be implemented. For smaller main memory sizes, it should only be used if the size of the outer relation is less than a small multiple of the size of main memory (typically up to three to six times). For larger main memories and large relations, it should always be used if the size of the outer relation is larger than main memory because the minimisation cost will be insignificant when compared to the running time of the join.

4.6 Stability of minimal allocation: varying seek and transfer times

It is important to know the effect of the seek and transfer times, T_K and T_T, on the minimal buffer arrangement. These times are used in the calculation of the cost of each join operation, and different hardware typically has different values for each of these times. If a small variation in the relationship between these values has a significant impact on the stability of the minimal buffer arrangement then the characteristics for each disk drive would have to be known. This would make the method of determining minimal buffer arrangements much less useful, and difficult to determine.

Table 6 shows possible values for V_1 in which each algorithm will be used while V_2, V_R, B, T_C, T_J and T_P are constant. Let $C(\overline{B}, N)$ be the cost of the join algorithm using buffer arrangement $\overline{B}$ and $N = T_K/T_T$. To calculate the cost ratio for a given value of T_K/T_T, N', we first find the minimal buffer arrangement, $\overline{B}$, for $T_K/T_T = 5$. We set $C_1 = C(\overline{B}, N')$. Now we find the minimal buffer arrangement, $\overline{B_1}$, for $T_K/T_T = N'$. We set $C_2 = C(\overline{B_1}, N')$ and the cost ratio is given by C_1/C_2.

Table 6 shows that the relationship between T_K and T_T does not have an enormous impact on the cost of the minimal buffer arrangement. We believe that the number of occasions in which an extremely accurate estimation of the relation-

Table 6. Buffer size changes as the relationship between T_K and T_T is varied, V_2 = 100 000 (781 MB), V_R = 10 000 (78 MB), B = 4096 (32 MB), $T_C = T_J = 3T_T$, $T_P = 0.4T_T$

	Nested block V_1 = 4093 (32 MB)			Cost	GRACE hash V_1 = 100 000 (781 MB)				Cost
T_K/T_T	B_1	B_2	B_R	ratio	B_1	B_2	B_R	P	Ratio
1	4093	2	1	1.00	3125	782	189	32	1.00
2	4093	2	1	1.00	3226	646	224	31	1.00
3	4093	2	1	1.00	3226	646	224	31	1.00
4	4093	2	1	1.00	3226	646	224	31	1.00
5	4093	2	1	1.00	3449	493	154	29	1.00
6	4093	2	1	1.00	3449	493	154	29	1.00
7	2047	1819	230	1.03	3449	493	154	29	1.00
8	2047	1819	230	1.10	3449	493	154	29	1.00

Table 7. Buffer size changes when the relationship between T_J and T_T is varied, V_2 = 100 000 (781 MB), V_R = 10 000 (78 MB), B = 4096 (32 MB), $T_K = 5T_T$, $T_C = T_J$, $T_P = T_J/8$

	Nested block V_1 = 4093 (32 MB)			Cost	GRACE hash V_1 = 100 000 (781 MB)				Cost
T_J/T_T	B_1	B_2	B_R	Ratio	B_1	B_2	B_R	P	Ratio
1.5	2047	1852	197	1.10	3449	493	154	29	1.00
2	2047	1852	197	1.00	3449	493	154	29	1.00
2.5	4093	2	1	1.00	3449	493	154	29	1.00
3	4093	2	1	1.00	3449	493	154	29	1.00
4	4093	2	1	1.00	3226	646	224	31	1.00
5	4093	2	1	1.00	3226	646	224	31	1.00

ship between the seek and transfer times is required will be relatively rare. This result also gives us confidence that the impact of additional seeks within large data files will be very small.

4.7 Stability of minimal allocation: varying CPU and disk times

It is also important to know the effect of CPU times compared with the disk seek and transfer times on the minimal buffer arrangement. As with the relationship between seek and transfer times, these times are used in the calculation of the cost of each join operation. Not only does different hardware have different values for each of these times, but the operating system and software used also affect the values of these times. If a small variation in the relationship between these values has a significant impact on the stability of the minimal buffer arrangement, then the relationship between the values would have to be known. This would make the method of determining minimal buffer arrangements much less useful, and more difficult to determine.

Table 7 shows possible values for V_1 in which each algorithm will be used while V_2, V_R and B are constant. The ratio of T_J and T_T is varied, while the other values are set so that $T_K = 5T_T$, $T_C = T_J$ and $T_P = T_J/8$.

We use a similar analysis to the previous section to derive the cost ratio. Let $C(\overline{B}, N)$ be the cost of the join algorithm using buffer arrangement $\overline{B}$ and $N = T_J/T_T$. To calculate the cost ratio for a given value of T_J/T_T, N', we first find the minimal buffer arrangement, $\overline{B}$, for $T_J/T_T = 3$. We set $C_1 = C(\overline{B}, N')$. Now we find the minimal buffer arrangement, $\overline{B_1}$, for $T_J/T_T = N'$. We set $C_2 = C(\overline{B_1}, N')$ and the cost ratio is given by C_1/C_2.

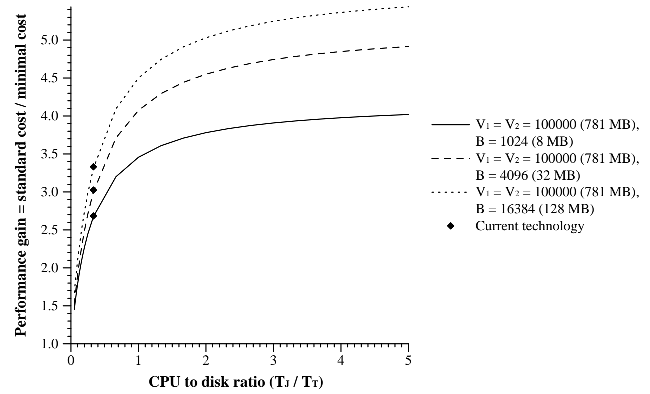

Fig. 11. Relative cost of minimal and standard buffer allocations when the ratio T_J/T_T varies. V_R = 10 000 (78 MB), $T_K = 5T_T$, $T_C = T_J$, $T_P = T_J/8$

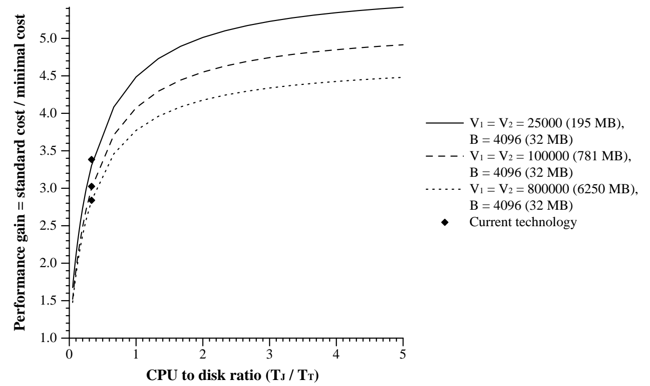

Fig. 12. Relative cost of minimal and standard buffer allocations when the ratio T_J/T_T varies. V_R = 10 000 (78 MB), $T_K = 5T_T$, $T_C = T_J$, $T_P = T_J/8$

Table 7 shows that the relationship between T_J and T_T does not have an enormous impact on the cost of the minimal buffer arrangement.

4.8 Benefits of minimal allocation: varying CPU and disk times

The current trend in hardware technology is for CPU speeds to increase at a rate greater than that of the disk drive technology (seek and transfer rates). As the cost of memory is decreasing, it is also likely that in the future more memory will be available to be used for buffers. In the future, will it be more or less beneficial to use the minimal buffer allocation?

Figures 11 and 12 show how the ratio between the CPU time constants (T_C, T_J and T_P) and disk time constants (T_K and T_T) affects the performance of the GRACE hash join algorithm. The ratio of the cost of the standard buffer allocation to the minimal buffer allocation is compared for a number of different buffer and relation sizes. These figures show that as the CPU speed gets faster, relative to the disk seek and transfer speeds, the minimal buffer allocations perform better than the standard buffer allocations.

In addition, Fig. 11 demonstrates the effect of varying the total buffer size. As the amount of memory available for

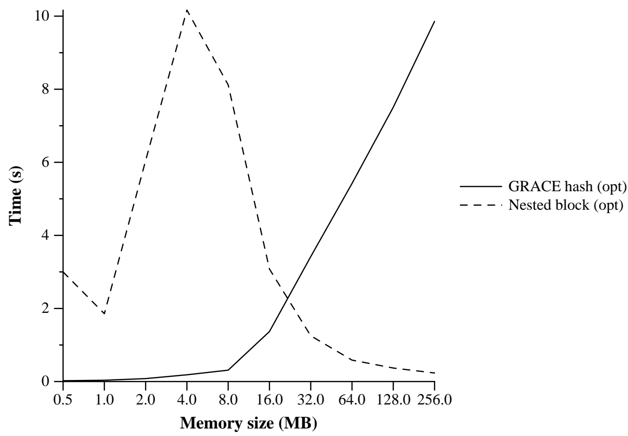

Fig. 13. Time taken by the nested block and GRACE hash minimisation algorithms. V_1 = 500 000 (3906 MB), V_2 = 1 000 000 (7813 MB), V_R = 100 000 (781 MB)

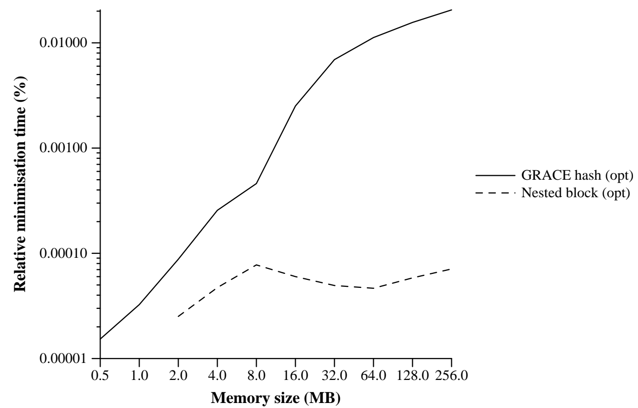

Fig. 14. Relative time taken by the nested block and GRACE hash minimisation algorithms. V_1 = 500 000 (3906 MB), V_2 = 1 000 000 (7813 MB), V_R = 100 000 (781 MB)

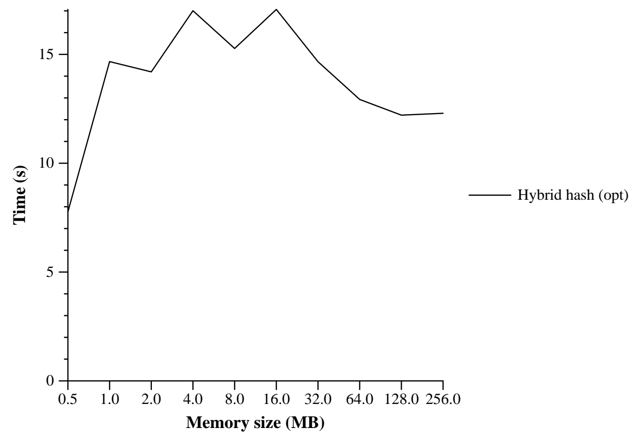

Fig. 15. Time taken by the simulated annealing algorithm. V_1 = 500 000 (3906 MB), V_2 = 1 000 000 (7813 MB), V_R = 100 000 (781 MB)

buffers increases, the performance of the minimal allocation over the standard allocation increases. Figure 12 demonstrates the effect of varying the relation sizes. It shows that smaller relations exhibit greater performance improvement than larger ones using the minimal buffer allocation. However, the difference is not as great as the impact of using different buffer sizes.

4.9 Minimisation performance as buffer sizes vary

In Sect. 4.5 we reported that the time taken by the minimisation algorithms is very small compared with the running time of the join. In Fig. 13 we present the time taken to minimise a typical join of two relations for a large number of different buffer sizes, ranging from 64 pages (512 KB) to 32 768 pages (256 MB). In Fig. 14 we present the relative time taken to minimise the join compared with the running time of the join. Note that the scale of the axis denoting the number of pages of memory is logarithmic. The results again show that the time taken by the minimisation algorithms are very small compared with the running time of the join algorithm. They also show that the running time of the GRACE hash minimisation algorithm is approximately linear in the amount of main memory available.

The time taken by the simulated annealing algorithm can be controlled by its parameters. In Fig. 15 we present the time taken to minimise the same join using simulated annealing. Note that the time taken is longer than the GRACE hash minimisation algorithm. However, as the time taken does not increase significantly as more memory is made available, there will be a point at which it becomes more cost effective to use simulated annealing than the GRACE hash minimisation algorithm (in this case, approximately 1 GB of main memory).

5 Experimental results

In order to validate our analysis and the results obtained in the previous section, we conducted a series of experiments. Programs were written which performed the appropriate I/O and CPU operations for the nested block and GRACE hash join algorithms when provided with input files. They were implemented on a Sun SPARCstation IPX running SunOS 5.3 and relied on the UNIX file system for file management. Thus, we had no control over the disk accesses required to retrieve the data.

The SunOS file system (McVoy and Kleiman 1991) attempts to keep consecutive pages together in cylinder groups whenever possible, as well as buffering disk pages and reading disk pages ahead. It also reads seven 8 KB pages at a time, so that reading one page at a time through a file does not result in a disk access for every read. To combat this we set the page size to be 56 KB. The large page size is likely to have the effect of moving the cost of the standard buffer allocation closer to the optimal buffer allocation because the one page allocated to the inner relation is, in fact, seven "normal" disk pages. This produces the same effect as the clusters of Graefe (1993).

We varied the amount of memory reserved for the memory buffer between 100 and 310 of the 56 KB pages; thus the total of amount of memory used varied between 5.5 and 17 MB. To minimise the effect of the buffer cache, we ensured that the total amount of memory allocated to our program was 17 MB, regardless of how much was used as the memory buffer. The system call mlock() was used to en-

Table 8. Timing parameter values for experimental results

Parameter	Value (s)
T_K	0.0233
T_T	0.0356
T_P	0.00881
T_C	0.00220
T_J	0.00317

Table 9. Relation sizes (in 56 KB pages) for experimental results

Join algorithm	V_1	V_2	V_R			
			join attribute			
			5	4	3	2
Nested block	256 (14 MB)	512 (28 MB)	1	1	9	808
GRACE hash	512 (28 MB)	1024 (56 MB)	1	3	33	—

sure that this address space was fixed in physical memory, and not swapped out, as our programs ran.

The disk drive used was an Elite-2. The values for the parameters T_K, T_T, T_P, T_C and T_J were calculated using initial diagnostic data provided by our program. We ensured that the UNIX buffer cache did not contain any of our data files as we collected the diagnostic data; therefore the costs estimated using these parameters should not be greater than the time taken by the join experiments. These values are shown in Table 8.

In the following results, all times reported are that of the total elapsed time of the algorithm. Thus, these results were susceptible to any other activity on the machine. To attempt to minimise and identify this, the experiments were performed when there was no other user active on the machine and each join was performed ten times. However, we did not remove network or other operating-system-related activities from the machine.

The data files used in the experiments consisted of 184 byte records, similar to those of the Wisconsin benchmark (Bitton 1983). Each record consisted of a unique identifier attribute, six integer attributes and three string attributes of length 52 bytes. The values of each of the integer attributes was chosen from a different domain, so that the result relations would be different sizes, depending on which attribute was used for the join. All experiments were performed on the integer attributes. Table 9 shows the sizes of the relations joined, and the size of the result relation for the attributes which were involved in a join.

Representative examples of the results of these experiments are shown in Figs. 16–19. The points on the graphs denote the mean and standard deviation of time taken to perform each join. Note that the times shown for the experiments are the *elapsed* times, which are susceptible to other activity on the machine. We would anticipate that in a more highly controlled environment the variation in the results would be much lower.

Figure 16 shows the cost of the experiments for six buffer sizes using the nested block join, and the expected cost calculated using the values in Table 8. The experimental cost is lower than the calculated cost. This is due to the presence of the buffer cache and the limited disk space in which we performed the experiments. These factors combine to result in the fact that it is difficult to experimentally determine the values of the constants with great accuracy. However, the trend for the cost to decrease, as the amount of memory which is used increases, is consistent with the expected cost. We would anticipate that in an environment in which the buffer cache was not available, the difference between the experimental and calculated costs would be much lower.

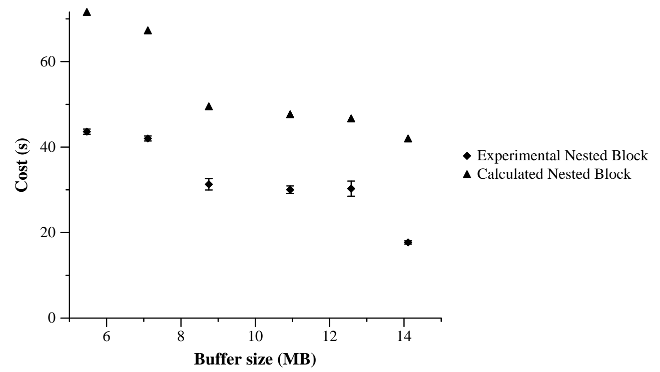

Fig. 16. Experimental cost and expected cost of performing nested block join versus main memory buffer size. V_1 = 256 (14 MB), V_2 = 512 (28 MB), V_R = 9 (504 KB)

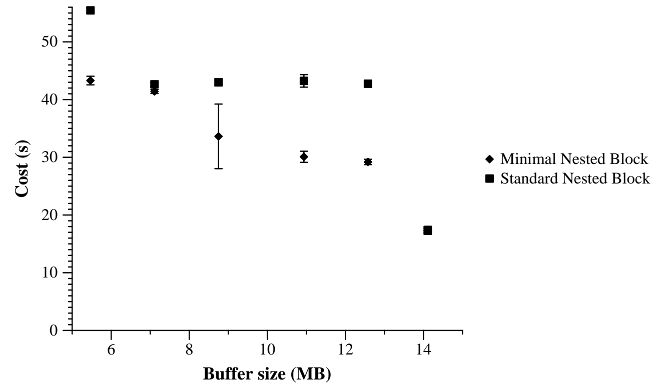

Fig. 17. Experimental cost of performing minimal and standard nested block join versus main memory buffer size. V_1 = 256 (14 MB), V_2 = 512 (28 MB), V_R = 1 (56 KB)

Figure 17 compares the performance of the minimal and standard versions of the nested block join algorithm. It is clear that a substantial improvement in performance can be achieved, particularly for the intermediate buffer sizes. The largest buffer size was chosen such that the minimal buffer allocation was the same as the standard buffer allocation.

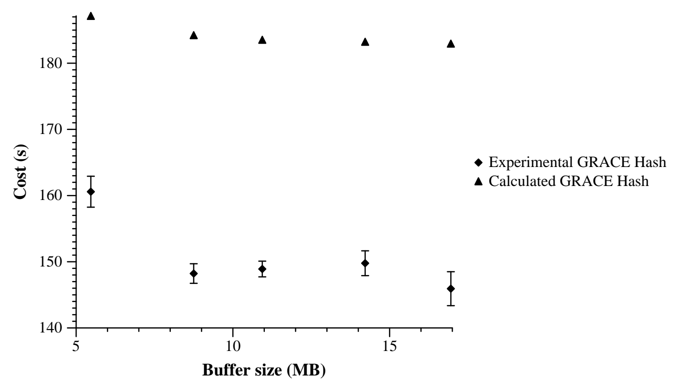

Fig. 18. Experimental cost and expected cost of performing GRACE hash join versus main memory buffer size. V_1 = 512 (28 MB), V_2 = 1024 (56 MB), V_R = 1 (56 KB)

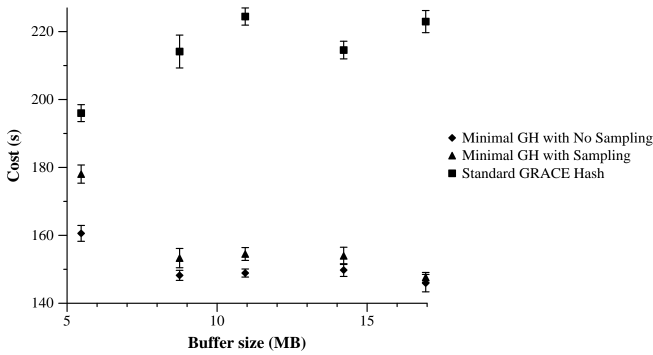

Fig. 19. Experimental cost of performing minimal and standard GRACE hash join versus main memory buffer size. V_1 = 512 (28 MB), V_2 = 1024 (56 MB), V_R = 1 (56 KB)

The outer file size was such that $V_1 = 256$, so the amount of memory chosen was such that $B = 258$, so $B_1 = 256$, $B_2 = 1$ and $B_R = 1$. Therefore, the results of the minimal and standard algorithms were expected to be the same, which was the case. For the second smallest buffer size, the results are again close. This is because the number of passes performed over the relations is different. The standard algorithm performs two passes, with $B_1 = 128$, while the minimal algorithm performs three passes, setting $B_1 = 86$. Therefore, even though the minimal algorithm performs an additional pass over the inner relation, its cost is still slightly lower than the standard algorithm.

Note that the file system and buffer cache reduces the cost of the inefficient standard buffer allocation more than it does for the more efficient minimal buffer allocation. For example, a file system prefetching a disk page has a much greater impact if one page is read at during each I/O operation than if a large number of pages are read during each I/O operation. In an environment in which a buffer cache is not available, a minimal buffer allocation would show a greater improvement over the standard buffer allocation.

Our experimental implementation of the GRACE hash join algorithm is slightly extended from the version described in the previous sections. The minimal buffer allocation is determined initially prior to partitioning. However, during the merging phase, the minimal buffer allocation is determined for each partition of the relations. This helps address the problem of unequal buffer sizes.

Figure 18 compares the cost of experiments for five buffer sizes using the GRACE hash join algorithm with the expected cost calculated using the values in Table 8. In this case the smallest buffer size creates eight partitions from each of the input relations, the middle three buffer sizes create four partitions from each of the input relations, and the largest buffer size creates two partitions from each of the input relations. The results are similar to that of the nested block join algorithm in Fig. 16 in that the trend as the amount of available memory increases is consistent with the expected cost, but the actual values of the constants used do not accurately provide the exact cost.

Figure 19 compares the performance of the minimal and standard versions of the GRACE hash join algorithm. It also contains results of a sampled version of the algorithm which is discussed in Sect. 6. The results show that a large improvement over the standard version of the algorithm is achieved. Across all the results, the average improvement varied from at least 15% for the smallest amounts of available memory to at least 30% for the larger amounts of available memory.

These results indicate that using a minimal buffer allocation, instead of the standard one, can result in the improvements that we have earlier shown to be theoretically possible. If the buffer cache had not been available, and the memory used by the buffer cache had been used by the join algorithm, an even greater improvement would have been realised. This is because the join algorithm can choose better pages to keep in memory than the UNIX buffer cache algorithm.

6 Non-uniform data distributions

The problem of non-uniform data distributions is common amongst all hash-based techniques. To address this problem Nakayama et al. (1988) proposed an extension to the hybrid hash method, called the Dynamic Hybrid GRACE Hash join method (DHGH). Their method dynamically determines which partitions will be stored in memory and which will be stored on disk. This depends upon the distribution of data. Kitsuregawa et al. (1989) provide an analysis of DHGH and show the effect of varying partition sizes. They show that a large number of small partitions is the best method to handle non-uniform distributions. These small partitions are combined for the merging phase to minimise the join cost. As with almost all other analyses of join algorithms, they count the number of disk pages transferred in their cost model.

The DHGH assumes that there is one input page and all other pages are for the partitions. As we have seen, this is generally not optimal when disk page access time is taken into account. Additionally, their method does not support partitioning of the data in place, which has significant benefits for the extended GRACE and hybrid hash methods.

Our proposal is to use sampling. It has been shown to produce good results for query optimisation (Lipton et al. 1990; Haas and Swami 1992). In our method, a sample of each relation is read. Each record is hashed into a range that is a number of times the size of the desired number of partitions. A table of the frequency of each of the hash values is constructed. Finally, each hash value is assigned to a partition such that the partitions of the relations are as close to equal size as possible. Each record in each relation is placed in the appropriate partition by looking up its assigned partition based on its hash value. The additional overhead of this method, compared with the methods described in the previous sections, is likely to be very small compared to the running time of the algorithm. We call this method the Minimal GRACE Hash algorithm for Non-Uniform distributions (MGHNU), and the original method the Minimal GRACE Hash Algorithm for Uniform distributions (MGHU). A similar extension can easily be made to the hybrid hash join algorithm.

The MGHNU method assumes that the sample of the first relation is representative of the whole relation. This may be achieved in two ways. A small sample of randomly chosen pages from the first relation may be read. This would effectively require $T_K + T_T$ time for each page, so only a small

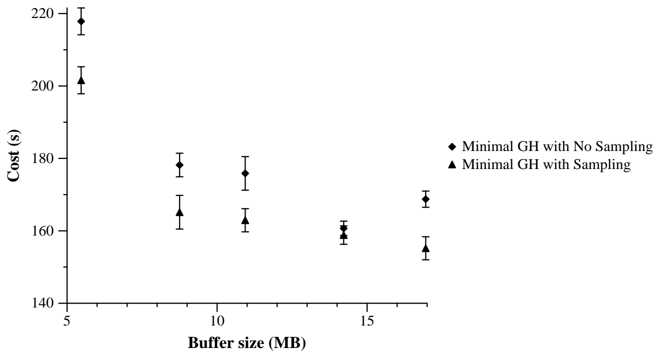

Fig. 20. Experimental cost of performing MGHU and MGHNU versus main memory buffer size. V_1 = 512 (28 MB), V_2 = 1024 (56 MB), V_R = 3 (168 KB)

number of pages should be read. The other way would be to read the first B_I pages, as the GRACE hash algorithm would normally do. As in DHGH, MGHNU will take more CPU time due to the construction of the table and the grouping of partitions. Therefore, the MGHNU method will be a little more expensive than the MGHU method when applied to a uniform data distribution. However, the MGHNU method would be much more efficient than the MGHU method for non-uniform data distributions. Therefore, we believe that the MGHNU method is the best method to use as a general method which tolerates non-uniform data distributions.

6.1 Experimental results

As mentioned in Sect. 5, Fig. 19 contains the results for a sampled version of the GRACE hash join algorithm. This is, in fact, an implementation of MGHNU. We were expecting that for a relatively uniform distribution, it would result in a higher cost than the MGHU method.

Figure 19 shows that for a relatively uniform data distribution, the additional cost of sampling the relations prior to performing the join did result in a small increase in the cost of the join. The number of pages transferred during sampling was less than 0.2% of the total number of pages transferred; however, reading each page resulted in a disk head seek.

The additional cost of sampling was greater when less main memory was available. A smaller amount of main memory means that the first disk I/O operation during the partitioning of each relation will request fewer pages than if a large amount of main memory was used. We expect that all of the pages which were sampled would be in the buffer cache prior to the first read of the partitioning phase. Therefore, the cost can be reduced by requesting as many of these as possible. More will be requested if the amount of main memory is larger, so we expect that the cost of sampling will be higher with a smaller main memory buffer, if there is a buffer cache available.

In addition to the uniform distribution, we performed experiments using relations in which the values of the attributes formed a Zipf distribution, as described, for example, in Knuth (1973). A representative example of the results is shown in Fig. 20.

In Fig. 20 we compare the MGHU ("No Sampling") and MGHNU ("Sampling") GRACE hash join algorithms. Note that, as described above, both methods determine the minimal buffer allocations after partitioning, thereby utilising the best buffer allocation after partitioning even if the data is non-uniform. Figure 20 shows that using the MGHNU method does result in a clear reduction in the cost of performing the join.

7 Multiple joins

Many queries asked of database systems are composed of multiple join operations. There are a number of methods of implementing multiple joins, from writing temporary files to disk at the end of each join, to performing each operation in a pipeline. By appropriately extending our minimisation algorithms, any of these methods could be used. Our primary result is independent of these implementations. That is, whatever method is used, the query can be executed faster by using a buffer allocation which takes into account all of the costs involved.

The algorithms we have described can be used without change if a query of multiple joins is implemented by writing temporary relations to disk as the result of each join. Each of the minimisation algorithms take into account B_R, the buffer size of the result relation, through which the temporary relation is written to disk. If the size of the temporary relation, as given by V_R, is accurate, the results give us confidence that a good buffer size will be chosen.

A method of implementing a query of multiple joins which can be more efficient is to buffer at least part of the result in memory. This can easily be supported using our cost model. The cost functions need only be changed so that the number of writing operations for the result relation is reduced by one, and the number of reading operations for the temporary relation in the next join is reduced by one. The minimisation algorithms must also be modified, increasing their complexity. If the number of variables becomes such that the minimisation is too slow, simulated annealing can be used to find a minimal buffer allocation, as we did for the hybrid hash join.

8 Parallel joins

In recent years, a large amount of research has taken place into parallel join algorithms, particularly join algorithms based on hashing (Richardson et al. 1987; Schneider and DeWitt 1989; Shatdal and Naughton 1993; Walton et al. 1991). Many of these algorithms are based on existing hash join algorithms, often the hybrid hash join algorithm. We believe that our technique will be just as important in this domain as in the sequential case.

Parallel join algorithms incur network (or shared memory) costs, in addition to the costs associated with the sequential algorithms. Tuples for joining may come both from the network and from a local disk. Network costs do not contain a "seek" factor, and in many algorithms individual tuples are transferred across a network rather than pages. Thus, the network traffic will not have a significant impact on the buffer sizes, and we believe that modified versions of our algorithms can be used to significantly improve the

cost of disk accesses in parallel join algorithms, as they do for the sequential join algorithms.

9 Conclusion

In this article, we have presented an analysis of four common join algorithms – the nested block, sort-merge, GRACE hash and hybrid hash join algorithms – based upon the time required to access and transfer a page from disk and the processing cost of the join. This is a generalisation of both the commonly used method of counting the number of disk pages transferred and a proposed alternative of counting the number of operations in which any number of disk pages may be transferred as a single operation. We have shown that it is very important to consider both the disk seek and transfer times and CPU times involved in the join algorithms to achieve optimal performance.

We have presented cost-effective algorithms to quickly find minimal buffer allocations for the nested block and GRACE hash join algorithms and suggested a mechanism for handling non-uniform data distributions. The time taken by these minimisation algorithms is less than 0.05% of the running time of the join operation. For all the buffer allocations which we verified, the minimal buffer allocation produced by the algorithms was the optimal buffer allocation.

We have reported experimental results which confirm that the performance gains reported in this article can be achieved in practice. With the current disk and CPU technology, we expect performance gains in the order of two to three times when using the join algorithms with minimal buffer allocations, rather than the standard join algorithms. Even if a buffer cache is available on the system, substantial improvements are possible, particularly for large relations in which the GRACE or hybrid hash algorithms must be used. In the future, as the relative speed of the CPU over the disks increases, the use of minimal buffer allocations will become more important.

Our modified version of the GRACE hash join algorithm will usually perform better than the extended hybrid hash join algorithm when the cost of calculating a minimal buffer allocation is taken into account and the amount of main memory is 32 MB or less, and the relations are larger than several times the size of main memory. For large memory sizes, the simulated annealing algorithm for the extended hybrid hash join algorithm is superior. This approach is further improved by using the buffer allocation produced by the GRACE hash minimisation algorithm as a starting point, instead of a random starting point. This approach was cost effective for 88% of the joins tested, resulting in an average improvement of 8.6% across all the joins tested.

Even in an operating system environment in which file system access is not directly under the control of the database, there will be a decrease in the total cost, as a result of a reduction in the number of system calls required to read the data and the CPU time of memory-based parts of the join algorithms.

Further work arising from this paper includes a comparison of how other methods of handling data skew impact on our scheme. For example, a comparison with non-sampling-based methods of determining data distributions, such as in Sun et al. (1993), would be interesting. Other join optimisation schemes which have been proposed which use an older cost model, such as in Harris and Ramamohanarao (1994), should be re-evaluated with the more accurate cost model. Another issue is to improve on the minimisation algorithms presented in this article; particularly to improve the speed and accuracy of the hybrid hash minimisation algorithm.

Appendix A: Hybrid hash constraints and cost

In this section, we describe the costs and constraints associated with the hybrid hash join algorithm which was described in Sect. 2.4.

In the cost formulae below, we assume that, during the partitioning phase, records are read into a buffer of size B_I and then distributed between $P+1$ partitions: P output buffers of size B_P, and a hash table of size B_H. For generality we assume that the hybrid hash join algorithm may have multiple partitioning passes, although this was not the application for which it was originally intended (Shapiro 1986). While we set the number of partitions created P to be a single value, it could vary for each of the ρ passes. Like the GRACE hash join algorithm, if a large amount of main memory is available, one pass will usually be enough. During the merging phase, the memory buffer is divided in exactly the same way as in the nested block algorithm. Therefore, the general constraints that must be satisfied are:

- The sum of the input, output, result and hash table buffer areas during the partitioning phase must not be greater than the available memory: $PB_P + B_I + B_H + B_R \le B$. The result buffer area is required because result tuples will be created as the second relation is partitioned.
- Some memory must be allocated as an input area: $B_I \ge 1$.
- Some memory must be allocated to each of the output partitions: $B_P \ge 1$.
- There must be multiple output partitions: $P > 1$.
- Some memory must be allocated to the in-memory hash table, and it need not be greater than the size of the outer relation: $1 \le B_H \le V_1$.
- The sum of the three buffer areas during the merging phase must not be greater than the available memory: $B_1 + B_2 + B_R \le B$.
- The amount of memory allocated to relation R_1 during the merging phase should not exceed the size of relation R_1: $1 \le B_1 \le V_1$.
- The amount of memory allocated to relation R_2 during the merging phase should not exceed the size of relation R_2: $1 \le B_2 \le V_2$.
- Some memory must be allocated to the result during the merging phase: $B_R \ge 1$ if $V_R \ge 1$.

The cost of the hybrid hash join algorithm is given by

$$V'_{1,i} = \left\lceil \frac{V_1 - B_H \sum_{j=0}^{i-1} P^j}{P^i} \right\rceil$$

$$V'_{2,i} = \left\lceil \frac{\left\lceil \frac{V_2(V_1 - B_H \sum_{j=0}^{i-1} P^j)}{V_1} \right\rceil}{P^i} \right\rceil$$

$$C_{\textit{Part:Read } R_1} = \sum_{i=0}^{\rho-1} P^i C_{I/O}(V'_{1,i}, B_I)$$

$$C_{\textit{Part:Write } R_1} = \sum_{i=1}^{\rho} P^i C_{I/O}(V'_{1,i}, B_P)$$

$$C_{\textit{Part:Partition } R_1} = \sum_{i=0}^{\rho-1} P^i V'_{1,i} T_P$$

$$C_{\textit{Part:Create}} = \sum_{i=0}^{\rho-1} P^i B_H T_C$$

$$C_{\textit{Part:Read } R_2} = \sum_{i=0}^{\rho-1} P^i C_{I/O}(V'_{2,i}, B_I)$$

$$C_{\textit{Part:Write } R_2} = \sum_{i=1}^{\rho} P^i C_{I/O}(V'_{2,i}, B_P)$$

$$C_{\textit{Part:Partition } R_2} = \sum_{i=0}^{\rho-1} P^i V'_{2,i} T_P$$

$$C_{\textit{Part:Join}} = \sum_{i=0}^{\rho-1} P^i \left\lceil \frac{V_2 B_H}{V_1} \right\rceil T_J$$

$$C_{\textit{Merge:Read } R_1} = P^\rho C_{I/O}(V'_{1,\rho}, B_1)$$

$$C_{\textit{Merge:Create}} = P^\rho V'_{1,\rho} T_C$$

$$C_{\textit{Merge:Read } R_2 \textit{ initial}} = P^\rho C_{I/O}(V'_{2,\rho}, B_2)$$

$$C_{\textit{Merge:Join initial}} = P^\rho V'_{2,\rho} T_J$$

$$C_{\textit{Merge:Read } R_2 \textit{ other}} = P^\rho \left(\left\lceil \frac{V'_{1,\rho}}{B_1} \right\rceil - 1 \right) \times C_{I/O}(V'_{2,\rho} - B_2, B_2)$$

$$C_{\textit{Merge:Join other}} = P^\rho \left(\left\lceil \frac{V'_{1,\rho}}{B_1} \right\rceil - 1 \right) \times V'_{2,\rho} T_J$$

$$C_{\textit{Write } R_R} = C_{I/O}(V_R, B_R)$$

$$\begin{aligned} C_{HH} = {} & C_{\textit{Part:Read } R_1} + C_{\textit{Part:Write } R_1} + C_{\textit{Part:Partition } R_1} + C_{\textit{Part:Create}} \\ & + C_{\textit{Part:Read } R_2} + C_{\textit{Part:Write } R_2} + C_{\textit{Part:Partition } R_2} + C_{\textit{Part:Join}} \\ & + C_{\textit{Merge:Read } R_1} + C_{\textit{Merge:Create}} + C_{\textit{Merge:Read } R_2 \textit{ initial}} \\ & + C_{\textit{Merge:Join initial}} + C_{\textit{Merge:Read } R_2 \textit{ other}} + C_{\textit{Merge:Join other}} + C_{\textit{Write } R_R}. \end{aligned}$$

Appendix B: Modified hybrid hash

The buffer arrangement of our modified hybrid hash join algorithm is depicted in Fig. 3. For the cost C_{HH} to be valid, the constraints are now that:

– $PB_P + 2P - 1 + B_H + B_R \leq B$ instead of $PB_P + B_I + B_H + B_R \leq B$.
– $B_I = PB_P$ instead of $B_I \geq 1$.

The other constraints remain valid.

Our algorithm to minimise C_{HH} is similar to that which minimises the GRACE hash join algorithm, except that, instead of minimising the cost using three variables, it must minimise using seven: B_1, B_2, B_R, P, B_P, B_H, ρ. We set $B_I = PB_P$. It takes significantly longer to produce a result than any of the other minimisation algorithms. Additionally, it is not feasible to perform an exhaustive search to determine whether it determines the minimal result or not. As a consequence, we do not present it here.

Acknowledgements. The authors would like to thank the anonymous referees for their comments which significantly improved this article. They would also like to thank Zoltan Somogyi for his valuable comments. The research in this paper was supported by the Australian Research Council, the Collaborative Information Technology Research Institute, the Cooperative Research Centre for Intelligent Decision Systems, and the Key Centre for Knowledge Based Systems. The first author was also supported by an APA scholarship.

References

Aarts E, Korst J (1989) Simulated annealing and Boltzmann machines. Wiley, New York

Bitton D, DeWitt DJ, Turbyfill C (1983) Benchmarking database systems a systematic approach. In: Schkolnick M, Thanos C (eds) Proceedings of the Ninth International Conference on Very Large Databases, Florence, pp. 8–19

Blasgen MW, Eswaran KP (1977) Storage and access in relational data bases. IBM Syst J 16(4):363–373

Cheng J, Haderle D, Hedges R, Iyer BR, Messinger T, Mohan C, Wang, Y (1991) An efficient hybrid join algorithm: a DB2 prototype. In: Proceedings of the Seventh International Conference on Data Engineering, IEEE Computer Society Press,Kobe, Japan, pp. 171–180

DeWitt DJ, Gerber R (1985) Multiprocessor hash-based join algorithms. In: Apers PMG, Wiederhold G (eds) Proceedings of the Eleventh International Conference on Very Large Databases, Stockholm, Sweden, pp. 151–164

DeWitt DJ, Katz RH, Olken F, Shapiro LD, Stonebraker MR, Wood D (1984) Implementation techniques for main memory database systems. In: Yormark B (ed) Proceedings of the 1984 ACM SIGMOD International Conference on the Management of Data, Boston, Mass., pp. 1–8

Graefe G (1993) Query evaluation techniques for large databases. ACM Computing Surv 25:73–170

Haas PJ, Swami AN (1992) Sequential sampling procedures for query size estimation. In: Stonebraker, M. (ed) Proceedings of the 1992 ACM SIGMOD International Conference on the Management of Data, San Diego, Calif, pp. 341–350

Hagmann RB (1986) An observation on database buffering performance metrics. In: Kambayashi Y (ed) Proceedings of the Twelfth International Conference on Very Large Data Bases, Kyoto, Japan, pp. 289–293

Harris EP, Ramamohanarao K (1994) Using optimized multi-attribute hash indexes for hash joins. In: Sacks-Davis R (ed) Proceedings of the Fifth Australasian Database Conference, Global Publications Services, Christchurch, New Zealand, pp. 92–111

Hua KA, Lee C (1991) Handling data skew in multiprocessor database computers using partition tuning. In: Lohman GM, Sernadas A, Camps R (eds) Proceedings of the Seventeenth International Conference on Very Large Data Bases, Barcelona, Spain, pp. 525–535

Kitsuregawa M, Tanaka H, Moto-oka T (1983) Application of hash to data base machine and its architecture. New Generation Comput 1:66–74

Kitsuregawa M, Nakayama M, Takagi M (1989) The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In: Apers, PMG, Wiederhold G (eds) Proceedings of the Fifteenth International Conference on Very Large Data Bases, Amsterdam, pp. 257–266

Knuth DE (1973) Sorting and searching. (The art of computer programming, vol. 3) Addison-Wesley, Reading, Mass.

Lipton RJ, Naughton JF, Schneider DA (1990) Practical selectivity estimation through adaptive sampling. In: Garcia-Molina H, Jagadish HV (eds), Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data, Atlantic City, NJ, pp. 1–11

McVoy LW, Kleiman SR (1991) Extent-like performance from a UNIX file system. In: Proceedings of the USENIX 1991 Winter Conference, Dallas, Texas, pp. 33–43

Merrett TH (1981) Why sort-merge gives the best implementation of the natural join. SIGMOD Rec 13:39–51

Mishra P, Eich MH (1992) Join processing in relational databases. ACM Computing Surv 24:63–113

Nakayama M, Kitsuregawa M, Takagi M (1988) Hash-partitioned join method using dynamic destaging strategy. In: Bancilhon F, DeWitt DJ (eds) Proceedings of the Fifteenth International Conference on Very Large Data Bases, Los Angeles, Calif, pp. 468–478

Omiecinski E (1991) Performance analysis of a load balancing hash-join algorithm for a shared memory multiprocessor. In: Lohman GM, Sernadas A, Camps R. (eds) Proceedings of the Seventeenth International Conference on Very Large Databases, Barcelona, pp. 375–385

Omiecinski E, Lin ET (1992) The adaptive-hash join algorithm for a hypercube multicomputer. IEEE Trans Parallel Distrib Syst 3:334–349

Pang H, Carey MJ, Livny M (1993) Partially preemptible hash joins. In: Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data, Washington, DC, pp. 59–68

Richardson JP, Lu H, Mikkilineni K (1987) Design and evaluation of parallel pipelined join algorithms. In: Dayal U, Traiger I (eds) Proceedings of the 1987 ACM SIGMOD International Conference on the Management of Data, San Francisco, Calif, pp. 399–409

Schneider DA, DeWitt DJ (1989) A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In: Clifford J, Lindsay B, Maier D. (eds) Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Ore, pp. 110–121

Shapiro LD (1986) Join processing in database systems with large main memories. ACM Trans Database Syst 11:239–264

Shatdal A, Naughton JF (1993) Using shared virtual memory for parallel join processing. In: Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data, Washington, DC, pp. 119–128

Sun W, Ling Y, Rishe N, Deng Y (1993) An instant and accurate size estimation method for joins and selection in a retrieval-intensive environment. In: Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data, Washington, DC, pp. 79–88

Vaghani J, Ramamohanarao K, Kemp DB, Somogyi Z, Stuckey PJ, Leask TS, Harland J (1994) The Aditi deductive database system. VLDB J, 3:245–288

Walton CB, Dale AG, Jenevein RM (1991) A taxonomy and performance model of data skew effects in parallel joins. In: Lohman GM, Sernadas A, Camps R. (eds) Proceedings of the Seventeenth International Conference on Very Large Databases, Barcelona, pp. 537–548

Evan Harris (B.Sc.(Hons) is a Postdoctoral Research Fellow at The University of Melbourne (Ph.D.).

Kotagiri Ramamohanarao (Ph.D.) is a Professor at The University of Melbourne.