Hash-Partitioned Join Method Using Dynamic Destaging Strategy

Masaya NAKAYAMA Masaru KITSUREGAWA
Mikio TAKAGI

Institute of Industrial Science, The University of Tokyo
7-22-1 Roppongi, Minato-ku, Tokyo 106, Japan

Abstract

In this paper we propose a new hash-partitioned join method using a dynamic destaging strategy for large scale databases.

The traditional hash-partitioned join methods such as the Hybrid Hash Join Method assume that the size of each bucket can be controlled by selecting a split function, and the characteristics of the buckets are statically specified. For materializing this assumption, we have to collect information about distributions of the join attribute value before processing a job. In general, however, join operations are applied to relations in which some restrictions are applied, and so it is not easy to collect that information before processing. If we cannot collect the information, the tuple distributions of each bucket may differ from the estimation. An example shows that the processing time in such a case becomes 1.4 times worse than the ideal one.

In this paper we propose a strategy in which the destaging buckets are selected dynamically, instead of a static decision of them during the split phase. Using this strategy, we don't have to collect information before processing and this method can be applied in many cases, which are unsuited to traditional methods. When we apply this method to that example which we mentioned, we can get the same performance as the ideal one.

1 Introduction

Eighteen years have passed since Dr. Codd suggested the relational data model in 1970. Many database systems, including commercial ones, use this data model now. In these systems, relational algebra operations, such as *select*, *project* and *join*, are basic operations for processing tuples of databases. Especially, it is well known that *join* is an expensive operation and many join methods have been proposed so far [Sto76,Kit83,DeW85].

Among these join methods, hash-partitioned join methods present good performance as shown in [Kit83,DeW84,Sha86]. In [DeW85], it is also shown that the Nest Loop Join Method is not suitable for large databases and the Hybrid Hash Join Method — one of the hash-partitioned join methods — presents the best behavior among the all of the shown algorithms in its figure 3. We may agree with that result when we have to handle large sized databases. But we think that we had better use a non-split based join method, the Hash Loop Join Method in this paper, when we handle medium sized databases. [1] This phenomenon was shown in [DeW84]. In Section 2, we reconsider this phenomenon by using the I/O cost formula and show our experimental results.

Second, in [Sha86], it is shown that the Hybrid Hash Join Method uses a split function which partitions the source relations into the minimum number of buckets and each bucket becomes differently sized. This partitioning is optimal when the split function can separate the source relations as it assumes. But it doesn't seem easy to find such a split function which can be applicable in all the situations. When the split function doesn't suit for the data distributions of the source relations, bucket overflows can easily occur. [2] On the other hand,

[1] It means that the medium sized databases are between the same size of the available memory and five times larger.

[2] When the bucket overflows occur, the performance of this

as shown in [Kit83], if we use a split function which divides the source relations into a large number of small sized buckets, the bucket overflows are reduced. And we partition the source relations into the same sized buckets. Using this manner, we can select the initially processed bucket, called R_1 in this paper, dynamically. In section 3, we show these two kinds of splitting strategies and discuss the situations in which they are applicable.

In Section 4, we show the algorithm of our proposed hash-partitioned join strategy. In this strategy, the source relations are partitioned into a large number of buckets during the split phase in order to reduce the possibility of bucket overflows as mentioned before. And also, the destaging buckets are dynamically selected in order to minimize the total I/O cost of the join processing. This strategy gives flexibility for selecting a split function and is applicable for most situations.

The Hybrid Hash Join Method uses a split function whose split range is as small as possible [Sha86]. In many cases, however, a join operation has some restriction conditions. So the split function may not partition the source relations as expected. In such cases, our proposed method results to be superior to the Hybrid Hash Join Method. We show an performance evaluation of unbalanced distributions of buckets in section 5.

In Section 6, we give the conclusions of this paper.

2 Is a non-split based join method really so bad ?

In [DeW84,DeW85], they said that the *Hybrid Hash Join Method* is superior to all the other kinds of join methods including non-split based join methods at most cases. In our experiments, when medium sized relations are processed, non-split based join methods give the better performance results than split based join methods. Here, the medium size means that the size of the relation to be joined is the same as or at most five times greater than that of available main memory. This phenomenon was shown in [DeW84]. In this paper, we reconsider the detailed behavior of join processing in medium sized relations.

In this section, we present the I/O cost formula of each method and show that the non-split based join method is superior to the split based join method for medium sized relations. In addition to these analytical formula, we show our experimental results here.

strategy becomes worse. So we have to avoid them as possible as we can.

2.1 The notations of our environment

In this paper, we assume the join operation of two source relations, named R and S, and that relation R is smaller than relation S.

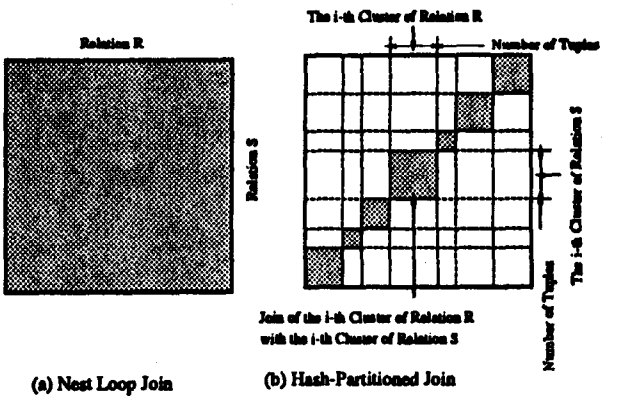


Figure 1: Processing cost of nest loop join methods and hash-partitioned join methods

When we treat relations which size are smaller than the available main memory, the relations may be processed by hash-partitioned join method (Figure 1). We call this phase as *'probe phase'*. On the other hand, when we treat relations larger than the available main memory, we cannot stage all tuples of the relation, and we have to separate it in sets of subrelations during the join processing. This separation phase is called as *'split phase'*.

If we divide these relations into a number of disjoint subrelations before processing the join operation, we call it a *'split based join method'*. On the other hand, if we don't divide them before processing, we call it a *'non-split based join method'*. For example, "GRACE Join Method" and "Hybrid Hash Join Method" described in [DeW85], are split based join methods and "Hash Loop Join Method" is a non-split based join method.

As shown in Figure 2, we use two kinds of hash functions in a split based join method. In this paper, we distinguish the coarse hash function from the fine hash function by calling them *split function* and *hash function* respectively. The split function is used when we partition the source relations (or subrelations) into a number of subrelations, called *buckets* (or *sub-buckets*). These partitions are based on page size because the split phase includes the I/O operations in it. In contrast to this, the hash function is used for reducing the

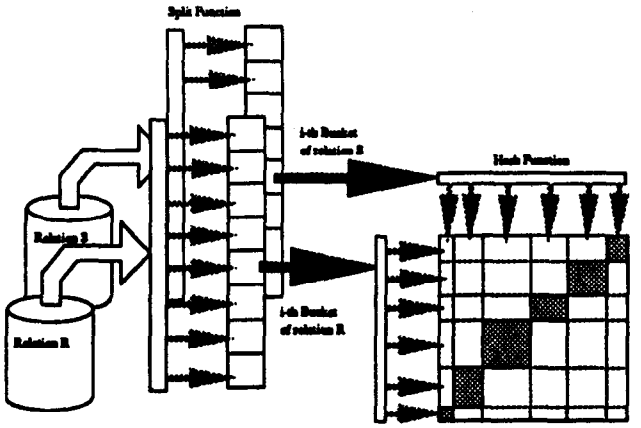


Figure 2: Two kinds of hash functions used by split based join methods

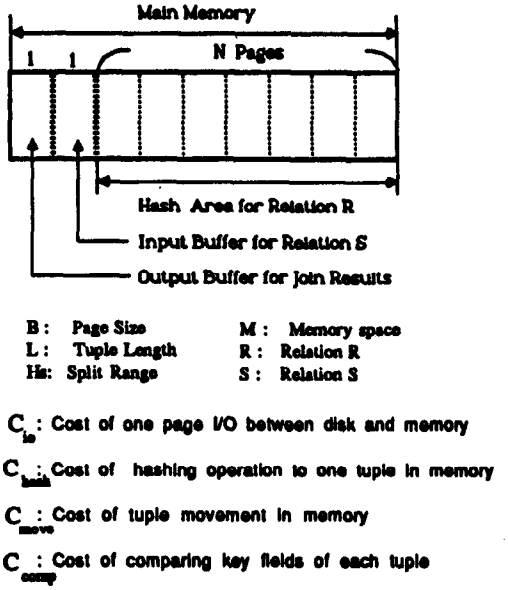


Figure 3: System parameters and memory usage in our experiments

processing cost of join operation, as shown in Figure 1. This partition is based on a tuple size and there are as many partitions as possible.

As shown in Figure 3, main memory contains N pages for staging the tuples of the relation R and 2 pages for input and output buffers. Totally, $N+2$ pages are available for processing a join operation. Here we describe the page size as B bytes. (In our experiments it is 2K bytes.)

H_s and H_h mean *'split range'* and *'hash range'* respectively. In split phase, we partition the relations into H_s buckets. In probe phase, we make a hash table which contains H_h entries and process a join operation by using this table.

In the formula below, $|R|$ means the number of *pages* of relation R. And $\{R\}$ means the number of *tuples* of relation R. Each relation is composed of L bytes long tuples, so the relations between $|R|$ and $\{R\}$ are represented by the following formula:

$$|R| = \left\lceil \frac{\{R\} * L}{B} \right\rceil \quad (1)$$

For relation S, we use the same notations. We use the notation M, meaning the available staging memory, in the formula. This means that $|M|$ is identical with N.

2.2 The cost formula of each join method

In this subsection, we show the algorithms of each join method and express the cost formula.

2.2.1 The cost of non-split based join method

As shown in [DeW85], the Hash Loop Join Method processes the relations as follows.

1. At first, part of relation R is staged into the available memory space, $|M|$ pages in our experiment. The hash table is built at the same time.

2. Second, each page of relation S is staged into the input buffer one by one. We apply the hash function to the tuples of each page of relation S in the input buffer and probe the joinability with the partially staged relation R.

3. These operations are iteratively processed until all the pages of relation R are staged in the memory.

The total cost of this method, C_{ns}, is expressed by following expression:

$$\begin{aligned} C_{ns} = & [|M| * C_{io} + \{M\} * C_{hash} \\ & + \{S\} * (C_{hash} + C_{comp}) + |S| * C_{io}] \\ & * \left\lceil \frac{|R|}{|M|} \right\rceil + (result) * C_{io} \end{aligned} \quad (2)$$

Cost parameters, C_{io}, C_{hash} and C_{comp}, are shown in Figure 3.

2.2.2 The cost of split based join method

We use a simple type of split based join method to formalize it here. The whole of the source relation is split into a number of buckets at *split phase*, and the joinability of each tuple is probed at *probe phase*. The algorithm of this method is as follows:

1. The relation R is staged into the memory and split into distinct H_s buckets. When the buffer page of a bucket gets filled, it is written back to the disk as a temporal relation. At the end, when the whole relation R has been staged and split, the part of the buckets remaining in the memory are flushed out to that temporal relation.

2. The relation S is scanned and split into the same number of buckets as relation R. As described before, all pages are destaged into the temporal relation during the split phase, and all the pages remaining in the memory are flushed out to disk at the end of this step.

3. All pages of each bucket R_i which were written back to the temporal relation, are now staged into the memory and a hash table is made. (In this algorithm, we assume that each bucket is smaller than available memory).

4. And each page of the corresponding bucket S_i is staged into the input buffer separately. We apply the hash function to each tuple and probe the joinability with bucket R_i. Unlike non-split based join method, we don't have to read and probe the pages of the bucket $S_j (j \neq i)$.

5. Step 3 and step 4 are iteratively processed until the rest of the temporal relation is empty.

Step 1 and step 2 correspond to the split phase, and the others correspond to the probe phase. The total cost of this method, C_s, is given in the following expression:

$$\begin{aligned} C_s &= 2 * |R| * C_{io} \\ &\quad + \{R\} * (C_{hash} + C_{move}) \\ &\qquad \text{(split phase of relation R)} \\ &\quad + 2 * |S| * C_{io} \\ &\quad + \{S\} * (C_{hash} + C_{move}) \\ &\qquad \text{(split phase of relation S)} \\ &\quad + \sum_{i=1}^{H_s} [|R_i| * C_{io} + \{R_i\} * C_{hash} \\ &\quad + |S_i| * C_{io} + \{S_i\} * (C_{hash} + C_{comp})] \\ &\quad + (result) * C_{io} \\ &\qquad \text{(probe phase)} \end{aligned} \tag{3}$$

2.3 The I/O cost analysis for processing join of the equal sized relations

The processing cost of each join method is described in the last subsection. Here, we show the I/O cost formula when applied to one of the Wisconsin benchmark environment [Bit83]. In this environment, we use equal sized relations for processing, and the join field value has unique value in each relation.

It is easy for the non-split based join method to estimate these situations from expression (2).

$$\begin{aligned} C_{ns} &\approx |R| * C_{io} + \{R\} * C_{hash} \\ &\quad + (\{R\} * C_{hash} + \{R\} * C_{comp} \\ &\quad + |R| * C_{io}) * \frac{|R|}{|M|} \\ &\quad + (result) * C_{io} \end{aligned} \tag{4}$$

For the split based join method, we assume that the buckets have the same number of tuples. From expression (3), we can derive the cost as follows.

$$\begin{aligned} C_s &\approx 4 * |R| * C_{io} \\ &\quad + 2 * \{R\} * C_{hash} + 2 * \{R\} * C_{move} \\ &\quad + (2 * |R_i| * C_{io} + 2 * \{R_i\} * C_{hash} \\ &\quad + \{R_i\} * C_{comp}) * H_s + (result) * C_{io} \\ &\approx 6 * |R| * C_{io} + 4 * \{R\} * C_{hash} \\ &\quad + 2 * \{R\} * C_{move} + \{R\} * C_{comp} \\ &\quad + (result) * C_{io} \end{aligned} \tag{5}$$

Instead of analyzing the total cost of each join method, we only analyze the I/O cost here. Since when medium sized relations are treated, the processing cost may be small enough compared with the I/O cost.

Here, we show the differences of each join method.

$$\begin{aligned} C_{ns} - C_s &\approx |R| * \frac{|R|}{|M|} * C_{io} - 5 * |R| * C_{io} \\ &\approx \left(\frac{|R|}{|M|} - 5 \right) * |R| * C_{io} \end{aligned} \tag{6}$$

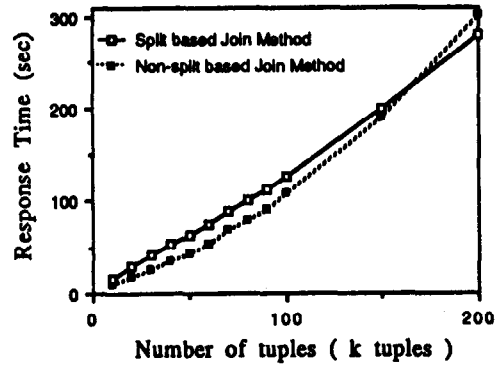
When the number of pages of relation R is less than $5 * |M|$, the I/O cost of non-split based join method

becomes lower than that of split join method. This formula suggests that it is better to use non-split based join method when we treat medium sized relations.

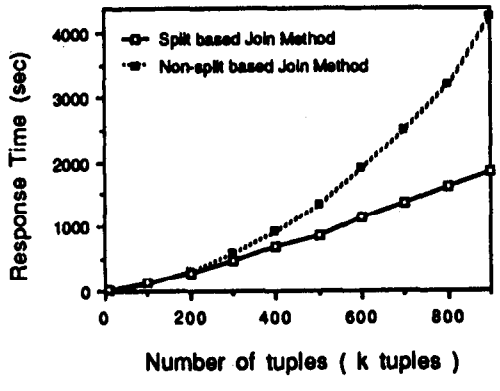
2.4 The experimental results of performance evaluation

We construct an experimental system on SUN3/260 to confirm our conclusion presented in the last subsection. In our experiment, we use relations with 64 bytes long tuples. and memory with 2M bytes for staging the relations during a join operation. It means that 1000 pages are available for staging buffers.

The split range is determined to minimize the number of buckets.



(a) Performance results of medium sized relaions



(b) Performance results of large sized relaions

Figure 4: Performance results of two kinds of join methods

In Fig 4, we show the results of join performance. Fig 4 (a) shows the performance results of medium sized relations, and a part of the results of Fig 4 (b). Fig 4 (b) shows the results of performance results of large sized relations.

A non-split based join method can give superior performance when treating medium sized relations. These results show the fact that medium sized relation should be processed by a non-split based join method. In our experiments, available memory contains 32 K tuples and five times larger size becomes 160 K tuples. Our results confirm the analytical conclusions.

3 How should we split the source relations ?

In the Hybrid Hash Join Method, a particular kind of bucket size distribution is assumed during the split phase. Although this policy is very simple, the bucket distribution is too particular and it is difficult to be achieved by using general split functions.

The policy is as follows:

1. If the relation size is larger than that of the available main memory, they must be split into independent and distinct buckets.

2. In such cases, we determine the number of buckets so that the size of each bucket is smaller than the available memory. This condition means that the largest size of a bucket is $|M|$ pages.

3. Generally, we don't have to use all the memory for buffering the splitting buckets. The unused part of the memory is utilized for making a hash table of bucket R_1. This tuning saves the I/O cost of bucket R_1 (and S_1).

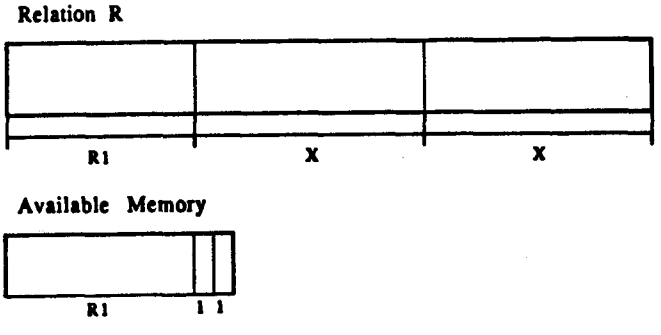


Figure 5: Tuple distributions of each bucket in the Hybrid Hash Join Method

Figure 5 shows the distribution behavior in the Hy-

brid Hash Join Method proposal. Split range H_s is determined by the following formula.

$$H_s - 1 = \left\lceil \frac{|R| - |M|}{|M| - 1} \right\rceil \quad (7)$$

$$|R_1| = |M| - H_s + 1 \quad (8)$$

$$|R_i| = \frac{|R| - |R_1|}{H_s - 1} \quad (2 \leq i \leq H_s) \quad (9)$$

Certainly, this formula gives a guarantee that all pages on memory are used in split phase. But it represents an ideal case for the size of relation R and it is difficult to find actual split function that can partition the relation into such distribution for any type of relations. In [DeW85,Ger86] , it was said that "In GAMMA, we prepare many kinds of split functions and select the suitable one to separate them into such distributions.". This implies that the selection of the split function for any case is not easy.

For this reason, we abandon this kind of bucket distributions, and use more easier distributions of buckets. As shown in Figure 6, we use a split function which only makes equal sized buckets. The *mod* function is a representative function which satisfies the conditions, when the data distribution of the relations is flat.

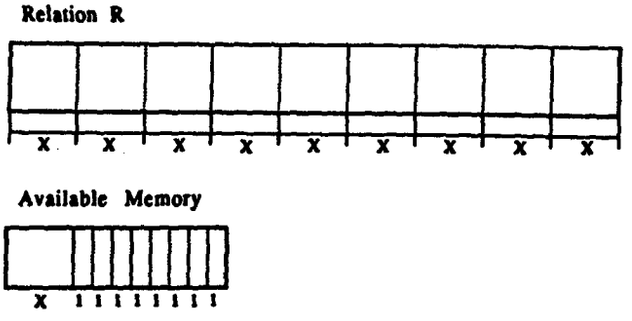


Figure 6: Tuple distributions of each bucket in our join method

In this situation, the size of the relations in which this method is applicable is reduced, because the size of each bucket must satisfy the following constraints.

$$\{R_i\} = \frac{\{R\}}{H_s} \quad (1 \leq i \leq H_s) \quad (10)$$

$$|R_i| \leq |M| - H_s + 1 \quad (1 \leq i \leq H_s) \quad (11)$$

These two relationships lead to the following constraint.

$$H_s^2 - (|M| + 1) * H_s + |R| \leq 0 \quad (12)$$

This constraint shows that the maximum applicable size of the relations with this bucket distribution becomes $\{(|M| + 1)/2\}^2$ when $H_s = (|M| + 1)/2$. [3] As H_s becomes larger, the applicable size of the relation becomes larger for $H_s \leq (|M| + 1)/2$, however for $(|M| + 1)/2 < H_s \leq |M|$, each bucket size becomes smaller because the rest of available memory for making a hash table of R_i is reduced. The total size of applicable relations becomes small in such cases.

When the source relation R is given, [4] how does one decide the size of the split range H_s ? Like the Hybrid Hash Join Method, should it be determined as the minimum one ?

By equation (3) or (5), it seems that there is no relation between the split range H_s and the cost of the join operations. But actually, the split range H_s reflects the total processing time of join operation. Figure 7 shows the response time varying according as the split range H_s change.

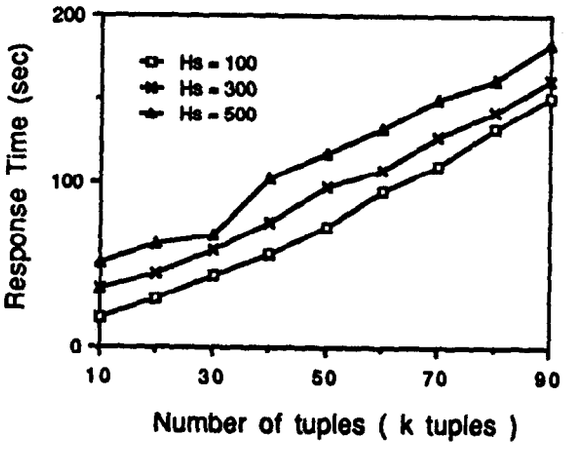


Figure 7: Performance results of changing the split range H_s

This result comes from the fact that the initialization cost of the hash table cannot be ignored. To avoid this overhead, we can process a set of buckets at once. If we split the relations into a large number of buckets, each

[3] The maximum applicable size of relations with Hybrid Hash Join Method becomes $|M|^2 - |M| + 1$ when $H_s = |M|$.

[4] In the following discussion, we assume that the size of the relation is applicable with our method.

bucket may have fewer tuples than $\{M\}$. Therefor they can be grouped in a set to be processed. This technique is used in our split based join method. It is originally suggested in [Kit83], where it is called *"bucket tuning"*. Using a large number of split range, we can easily reduce the *"bucket overflow"*s shown in [Kit83,DeW84].

In addition to this, we can dynamically determine the staged bucket by using fair bucket distributions. For the Hybrid Hash Join Method or other split based join methods discussed in [DeW84,DeW85], the staging buckets are statically decided, that is, they are decided before the split phase.

On the other hand, we determine the staging bucket dynamically during the split phase. The detailed strategy is shown in the next section.

We think that the selection of split functions is not easy for any situation. From this point of view, it is better to partition the relation into as many buckets as possible. When we use this kind of split range H_s, the possibility of bucket overflow becomes the lowest, and we can select the staging buckets dynamically instead of statically. If we select the destaging bucket as the one containing the largest number of tuples when we need an extra page, we can guarantee that the other buckets are kept on the available memory. It gives the system the flexibility for the selection of split function.

4 A new type of split based join method using the dynamic selection strategy of paged out bucket

In this section, we show a new type of split based join method. In this method, the paging out bucket is chosen dynamically during the split phase, and we collect a set of buckets which can be processed at the same time during the split phase. Following in this section, we show the algorithms of this method. At first, we show the staging strategy of relation R in split phase. And the end of staging of the relation R, we schedule the cluster for all buckets and make the hash table for staged buckets on memory. These buckets on memory are treated as R_1 in Hybrid Hash join Method. After this staging, the relation S is partitioned into the same number of buckets and probe the joinability of R_1 buckets. Now, split phase is finished. In probe phase, each cluster of relation R is staged into memory, and probe the joinability with the cluster of relation S respectively. When the size of the i-th cluster exceeds the size of available memory, this cluster is treated as original relation and is applied this strategy recursively.

4.1 Dynamic selection strategy of paged out bucket

As shown in Figure 3, the available memory is composed of $N+2$ pages, and N pages are used as staging area for relation R during the split phase. Concerning the other two pages, one is used as the input buffer of initial staging of both relations, and the other is not used during the split phase. All pages used for staging area, N pages, are initially linked to the free pages list.

At first, relation R is scanned and their pages are staged into the input buffer. The split function is applied on the join key fields for each tuple in this buffer. According to the returned value of this split function, each tuple is copied into the buffer of the appropriate bucket. If there is no room to in this buffer, a new page is allocated from the free pages list. When the free pages list becomes empty, dynamic destaging strategy works as follows.

1. Scan the split table and search for a bucket which has already been paged out. If found, go to step 2. And if there are no paged out buckets, go to step 3.

2. Check the number of tuples staged on memory for this bucket. If the number exceeds one page, mark it as the bucket to be paged out again and go to step 4. Otherwise go back to step1. We choose the bucket to be paged out again as the one whose number of pages on the memory exceeds ones, to avoid the writing and reading of useless area of a page. These extra accesses would lead to a worse performance, and should be avoided if possible.

3. When there are no paged out buckets, the largest bucket is preferably selected as the paged out bucket, because it has, generally, the highest possibility of bucket overflow among all the buckets.

4. When the paging out bucket is determined, all of its filled pages are destaged into the temporal relation. As described in step 2, we don't page out the partial filled pages.

If the distributions of tuples are known before the staging is processed, we can determine the paged out bucket optimally. But usually, join operation follows some restriction operations, and it is impossible to know these distributions, so we choose a strategy which can delay the selection of the paged out bucket as later as possible. Using this strategy, we can get an optimal paged out bucket at each paging out time. In Figure 8, we show some examples of this strategies.

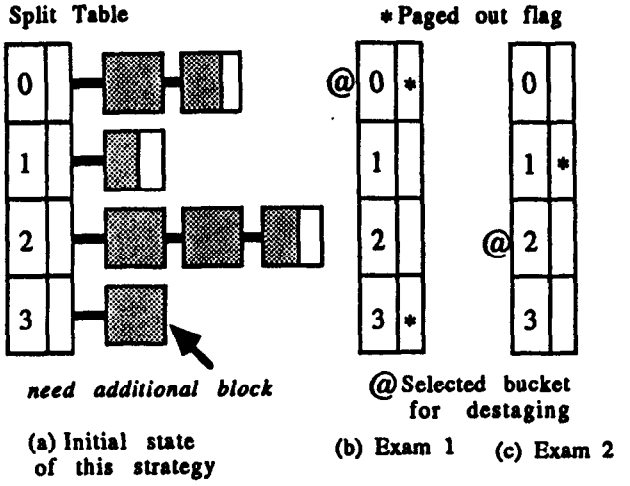


Figure 8: Example of dynamic destaging strategy

4.2 The scheduling of the processing buckets

When the relation R is scanned, we can classify all the buckets into two kinds: one in which all tuples are staged on the memory and the other in which some/all tuples are destaged into the temporal relation.

The bucket R_1 in the Hybrid Hash Join Method belongs to the former kind. And each tuple belongs to this kind of buckets is applied hash function to make the hash table. As shown in Figure 7, the initializing cost of the hash table cannot be ignored. It is the reason why we choose this kind of collection. So All buckets staged on the memory are processed at the same time in our join method.

For the buckets which contain paged out tuples, we make a schedule of their processing sequence and gather these buckets. We call a set of buckets, '*cluster*'. For the total cost optimization, we have to check the whole combination of buckets. It is easily shown that the complexity of this schedule becomes N-P complete. And when the total number of buckets becomes larger and larger, it becomes more difficult to search the best scheduling. To reduce the scheduling cost, we use the simple bucket tuning method described below.

At first, check the number of tuples of each cluster. If the difference from $\{M\}$ is more than the tuple number of the scheduled bucket, add that bucket as a member of the cluster. When no cluster can hold that bucket, make a new cluster and add that bucket as the member of the cluster.

4.3 Overlap the probing of the R_1 buckets with staging the relation S

As mentioned in the last two subsections, we split the relation R into H_s buckets, and make a schedule of the clusters. After that, we split the other relation S into H_s buckets.

Scan the relation S and stage each page into the input buffer. For each tuple in this buffer, the same split function used for the relation R is applied. If the tuple is a member of the staged buckets on the memory, probe the joinability immediately. Otherwise, that tuple is copyed into the staging buffer of each bucket in the same way as was done to relation R. When the whole relation S is scanned, the split phase is finished.

4.4 Processing strategy for the overflow buckets

Now we have a set of clusters in temporal relations. Usually each cluster is a set of buckets where total size fits in the available main memory. When the distribution of each bucket does not fit to this situation, some buckets may overflow the available memory. For example, if all tuples in the relation R have the same split value, we cannot guarantee the safety of this method. In such a case, we select the following strategy.

As shown in section 2, it is better to use the non-split based join method when the size of the bucket is between the size of the available main memory and five times larger.

Otherwise, we apply our split based join method recursively. In this case, another split function is used during the recursive split phase. During this recursive phase, most of the overflow cases can be solved.

Even though we apply these strategies, they cannot be solved all the situations. As mentioned before, when all of the tuples have the same value in the join field, we cannot separate them by any kind of split functions. So, in our method, this recursion finishes when the recursive nests reach a level which is defined before we process the join operation. In such cases, we will change the strategy to the non-split based join method.

Using this strategy, higher performance than that of the Hybrid Hash Join Method can be obtained, in many cases. The next section shows an example in which our method works better than the Hybrid Hash Join Method. This example is only a sample case, but it is believed that in many cases these situations occur.

5 An example of unbalanced bucket distribution

In this section, we show an example situation in which the Hybrid Hash Join Method does not perform so well.

As described in section 3, the Hybrid Hash Join Method uses a strategy in which the number of buckets becomes as small as possible. It is assumed that a suitable split function for the given relations can be found. In this section, we consider a situation where this suitable split functions cannot be found. In general, it is an usual case because join operations follow some restriction operations and we cannot guess the distributions of each bucket.

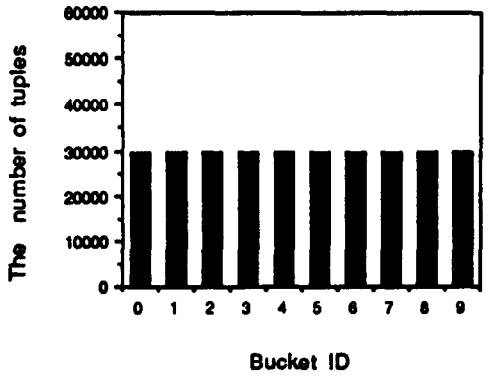
For example, consider the following situations. The relation R has two attributes, Name and Address. Here, assume that Name field is the primary key of the relation R, and we can keep the information about total tuple distributions. In this situation, when we want to join all tuples in the relation R with another relation S, we can split it as we expected by using such information. However, when we want to join *the person who lives in Tokyo* with another relation S, we cannot split it as we expected. Because the tuple distribution which satisfies the condition is not identical with that of whole relation.

We apply the join operation for such relations in this section. We prepare the source relations which have unbalanced distributions of buckets before any restrictions are given. It is to simplify the situation.

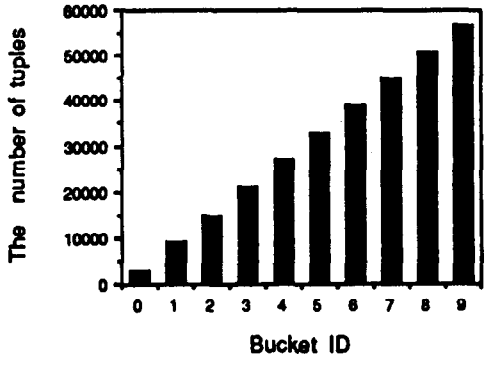
Here we show two relations which have the same characteristics. Each relation is made of 64 bytes long tuples, [5] and the join attribute of our experiments is a 4 bytes integer field. The value of each tuple is unique in the relation. One relation P has the sequential value in the join attribute, and the other relation Q has a particular distributions in the join attribute. In our experiments we use a simple *mod* functions to split the relation and to probe the joinability. Each hash function sees the different bit fields of join attribute and returns the modulus value. We can separate each bucket uniformly, when we process the relation P. On the other hand, if we process the relation Q using this function, the distribution of each bucket becomes unbalanced. Figure 9 shows this result of 300 K tuples. In this situation, the split range H_s becomes 10 by using the Hybrid Hash Join Method. However, by using our proposed join method it becomes 500, determined by $(|N|+1)/2$.

In such situations, we cannot determine the split

[5] Practically, the tuple length or the number of attributes are not a problem in general. Only the value and distributions of join attributes are a matter of concern.



(a) balanced tuple distribution relation P



(b) unbalanced tuple distribution relation Q

Figure 9: The tuple distributions of each relation

functions and the split range H_s so easily. Though the Hybrid Hash Join Method has advantages when the split function is suited to the data distributions of target relation, the total cost becomes worse when it isn't suited to the distributions.

On the other hand, our proposed join method uses additional buckets for adjusting their sizes. This may avoid the bucket overflows in many cases as mentioned before, however it needs additional buffers for those extra buckets during split phase.

Here, we will consider this overhead by using an example. When we treat 300 K tuples long relations, the I/O number for scanning each relation becomes 9375 pages. If we can apply the optimal split function to those relations, the split range H_s becomes 10, as described before. To join them, including the output of the result, we have to read or write them four times at least. It means that, even though in an optimal case, $9375 * 4 * 2 - 991 * 2 * 2 = 71036$ pages I/O occurs. In

such a case, our join method needs 490 additional I/O's for each scanning. The 490 * 2 * 2 pages I/O is merely 2.8 % of the total I/O cost.

Figure 10 shows our experimental results varying according to the size of relations. This figure shows that our method does not reduce the performance for unbalanced bucket distributions. Thus, this method is more flexible than the Hybrid Hash Join Method.

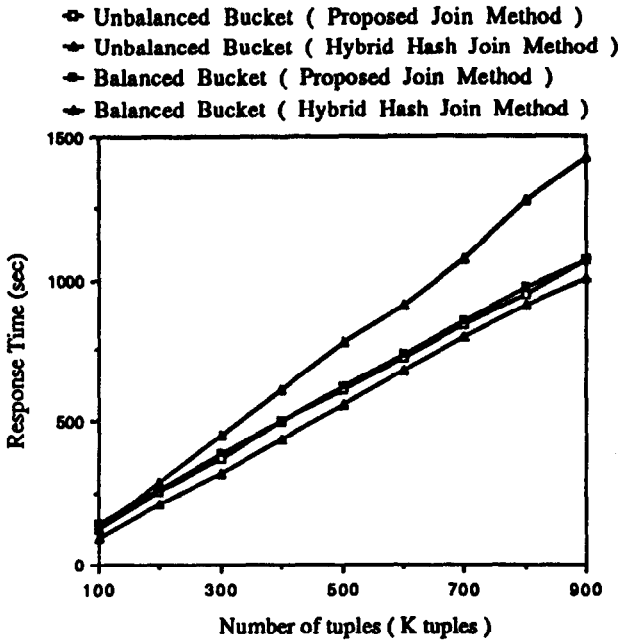


Figure 10: Join performance of unbalanced tuple distributions

This figure also shows that the processing cost of additional bucket is mere, and the processing cost of applying the recursive split phase cannot be ignored. It means that it is better to use a large number of buckets during split phase.

6 Conclusions

In this paper, we pointed out the disadvantages of the traditional split based join method, concentrating our analysis on the Hybrid Hash Join Method.

In most of the split based join methods, the number of buckets is statically determined by a split function, assuming that the result bucket size is uniformly distributed.

However, we think that such cases are actually rare because join operations generally follow some restrictions and there is no guarantee that the split function is suitable for any kind of data distribution of the source relation.

Especially in the Hybrid Hash Join Method, the split function has to deal with two kinds of bucket distributions: one is for the first staged hash bucket and the other is for the destaging buckets. In both cases, unbalanced distributions of tuples in the buckets lead to a worse performance than the estimated one.

We proposed a more flexible method, suited for many kinds of data distribution. In our proposed method, the source relations are split into a larger number of buckets. Whenever necessary, a dynamical selection strategy for the determination of the page out bucket is performed. This dynamical strategy decreases the possibility of overflows or underflows of the first staging hash bucket. Aiming at increasing the efficiency of the memory space and decreasing the overhead due to the hash table initialization, the buckets are gathered in clusters before processing. The clusters are determined so as to fill up the memory.

This strategy was shown to be efficient for large sized relations. For medium sized relations, instead of applying this strategy, we use a non-split based join method, whose estimated I/O cost shows to improve the join operation performance.

Our experimental results show that the method proposed in this paper presents higher performance than the Hybrid Hash Join Method in many cases.

References

[Bit83] Bitton, D., DeWitt, D. J. and Turbyfill, C. Benchmarking database systems - a systematic approach. In *Proceedings of VLDB '83*, 1983.

[DeW84] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R. and Wood, D. Implementation techniques for main memory database systems. In *ACM SIGMOD '84*, pages 1–8, ACM, 1984.

[DeW85] DeWitt, D. J. and Gerber, R. Multiprocessor hash-based join algorithms. In *Proceedings of VLDB '85*, pages 151–164, 1985.

[Ger86] Gerber, R. H. *Dataflow Query Processing Using Multiprocessor Hash-partitioned Algorithms.* Technical Report 672, University of Wisconsin, 1986.

[Kit83] Kitsuregawa, M., Tanaka, H. and Moto-oka, T. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1):66–74, 1983.

[Sha86] Shapiro, L. D. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239–264, 1986.

[Sto76] Stonebraker, M., Wong, E., Kreps, P. and Held, G. The design and implementation of ingres. *ACM Transactions on Database Systems*, 1(3):189–222, 1976.