Performance Evaluation of Functional Disk System with Nonuniform Data Distribution

Masaru KITSUREGAWA, Miyuki NAKANO, Lilian HARADA, Mikio TAKAGI
Institute of Industrial Science
University of Tokyo
22-1, Roppongi 7, Minato-ku, Tokyo
Japan

ABSTRACT

In this paper, we analyze the performance of a Functional Disk System with Relational database engine (FDS-RII) for a nonuniform data distribution. FDS-RII is a relational storage system, designed to accelerate relational algebraic operations, which employs a hash-based algorithm to process relational operations. Basically, in the hash-based algorithm, a relation is first partitioned into several clusters by a split function. Then each cluster is staged onto the main memory and, further, a hash function is applied to each cluster to perform a relational operation. Thus, the nonuniformity of split and hash functions is considered to be resulting from a nonuniform data distribution on the hash-based algorithm. We clarify the effect of nonuniformity of the hash and split functions on the join performance. It is possible to attenuate the effect of the hash function nonuniformity by increasing the number of processors and processing the buckets in parallel. Furthermore, in order to tackle the nonuniformity of split function, we introduce the Combined Hash Algorithm. This algorithm combines the Grace Hash Algorithm with the Nested Loop Algorithm in order to handle the overflown bucket efficiently. Using the Combined Hash Algorithm, we find that the execution time of the nonuniform data distribution is almost equal to that of the uniform data distribution. Thus we can get sufficiently high performance on FDS-RII also for nonuniformly distributed data.

1. Introduction

The join operation is one of the most expensive operations in relational database systems. Recently many researchers have proposed several hash-based algorithms for the join operation, such as the Grace Hash Algorithm and the Hybrid Hash Algorithm[3, 4, 5, 6, 9, 10]. The hash-based algorithms attain much higher performance for very large relations than the conventional nested loop or sort-merge algorithms[11], while providing a natural opportunity for parallel processing, which makes them very attractive for implementation on parallel database machines. Some performance evaluation results of hash-based algorithms on parallel database machines, reported in [2, 7, 11], show that the hash-based algorithms are very efficient for large relational database systems. The performance of these hash-based algorithms, however, is analyzed only under the assumption of uniformly distributed data. There are few reports on the performance of the hash-based algorithms with nonuniform data distribution[8, 11].

In this paper, we examine the effect of data distribution nonuniformity on the Functional Disk System with Relational database engine (FDS-RII), we developed in order to accelerate relational algebraic operations[1]. FDS-RII is not a simple storage system; it introduces some semantics of data processing into the storage system itself and incorporates a large staging buffer memory with a parallel processing mechanism. FDS-RII employs filtering and hash-based dynamic clustering mechanisms as special hardware functions, providing an intelligent data management and an efficient data processing with multiple processors. We have already proposed the processing algorithm on FDS-RII: the "Combined Hash Algorithm" in [2] that chooses the processing strategy from the hashed Nested Loop and the Grace Hash Algorithms at run time by comparing their I/O cost. In order to clarify the efficiency of the Combined Hash Algorithm, the performance of FDS-RII has been evaluated and analyzed with a uniform data distribution. With the expanded version of Wisconsin Benchmark[7], we evaluated the basic performance of FDS-RII, which attained very high performance for very large relations in comparison with Gamma and Teradata[2].

In this paper, the effects of nonuniform data distribution on the performance of hash-based algorithms are discussed. Basically, in the hash-based algorithms, source relations, much larger than the staging buffer memory (or main memory), are partitioned into several clusters by a split function. Each cluster is staged onto the staging buffer memory and a hash function is then applied to each cluster to perform the join operation. The effects of these two types of functions on FDS-RII are analyzed separately. First, we analyze the performance of FDS-RII with nonuniformly distributed data that fit into the staging buffer memory and are clustered into buckets with a hash function. This means that we evaluate only the join phase of the hash-based algorithm. Secondly, we evaluate the efficiency of the Combined Hash Algorithm on FDS-RII. In the Combined Hash Algorithm, the Nested Loop Algorithm is adopted as a run time method, when the relations are not so large. When the benchmark relations are much larger in size than that of the staging buffer memory, a split function is used to partition source relations into clusters equal in size to the staging buffer memory by using the Grace Hash Algorithm. When a cluster overflows the staging buffer memory, the Combined Hash Algorithm is applied in order to choose once again a processing strategy for the overflown clusters, from the hashed Nested Loop or the Grace Hash Algorithms, at run time. We show how the Nested Loop Algorithm is adopted for the overflown cluster. Since it is not necessary for the Nested Loop Algorithm to partition the source relations into clusters, we expect that the performance of the Combined Hash Algorithm is not affected with nonuniform data distribution.

Section 2 briefly introduces the hardware and software configurations of FDS-RII. In Section 3, the effect of nonuniform data distribution on hash-based join algorithms is analyzed in detail. In Section 4, we analyze the nonuniformity of the hash function, describing the data flow of FDS-RII and the query processing strategy for relations whose sizes fit into the staging buffer memory. The performance of FDS-RII is measured and analyzed with the normal distribution by increasing the number of processors. In Section 5 presents an analysis of the split function nonuniformity. When the size of the source relations is larger than that of the staging buffer memory, the Combined Hash Algorithm is employed on FDS-RII. We describe in detail how the appropriate
processing strategy is determined at run time by comparing I/O costs of the two algorithms (Nested Loop and Grace Hash) in the Combined Hash Algorithm. The performance evaluation results of FDS-RII with normal distribution and Zipf-like distributions are reported and analyzed. The last section concludes this paper and discusses future topics for research on FDS-RII.

2. Overview of the Functional Disk System

While the CPU performance has been very much improved during the past decade, the same cannot be stated about the secondary storage, whose performance has been a very serious problem on large data processing. In order to dissolve this "I/O bottleneck" problem of the storage, whose performance has been a very serious problem between CPU and the secondary storage, we have proposed the data interface with the host. In order to verify our approach, we have constructed a pilot system.

2.1. Hardware Configuration of FDS-RII

Fig.1 shows the hardware configuration of FDS-RII. FDS-RII consists of three parts: the processing part, the staging buffer and the disk part.

The processing part consists of four processors. One of them is the master and the other three are the slave processors. Each processor board is a commercial one which contains the MC68020 CPU(16MHz) and 1 MB of local memory. The staging buffer memory consists of 6MB dynamic memory which is shared by the four processors. The processing part, the staging buffer memory and the disk part are connected by Versa Bus, the standard MOTOROLA bus. In order to increase the I/O bandwidth, another special bus called "SO Bus" is added to the system. It connects the processor boards to the disk part and is used only to store the result relations onto the disks. The disk part is composed of 8 inch disks (Fujitsu M2344KS) and the "Intelligent Disk Controller (IDC)". The IDC is the heart of this disk part and is responsible for the control of the disks and for the filtering and hash-based dynamic clustering of the data read from the disks.

2.2. Software Configuration of FDS-RII

Fig.2 shows the configuration of the FDS-RII system software.

The miss-match between a database system and a general purpose operating system deteriorates the performance of database systems[17]. Therefore, in FDS-RII, a dedicated I/O environment is developed. In order to obtain high performance, a ROM based MICROWARE OS-9 is used only as a low level process control. The buffer management, the I/O drivers and the process communication, which are called "FDS Kernel" and are key functions for database systems, are specially developed and optimized on FDS-RII. Since a hash-based algorithm is employed on FDS-RII and filtering and dynamic clustering mechanisms are implemented as special hardware functions, FDS Kernel provides buffer management at cluster level, which supports a flexible space utilization for multiple processors and IDC. The FDS Kernel also provides a high interface for the database system through the I/O drivers, by which applications can access disks at the logical data level, such as a relation or a tuple level. The database system on FDS-RII consists of the QUEL Parser, the Packet Generator, the FDS Runtime Manager and the Packet Handler as shown in Fig.2. The QUEL Parser, the Packet Generator and the FDS Runtime Manager run on the master processor and the Packet Handler runs on each slave processor. The QUEL Parser provides the user interface and supports the QUEL language which is used on Ingres system. The Packet Generator makes a packet which contains the run time information of FDS-RII. The FDS Runtime Manager manages the database resources and controls the system, actually executing a query. The Packet Handlers are controlled by the FDS Runtime Manager and support parallel processing of relational operations.

2.3. Query Execution on FDS-RII

The query execution flow on FDS-RII is shown in Fig.3. When a user issues a query on the master processor, first, the QUEL Parser parses the query and makes the query tree, which contains simple relational operations and resource information. Then the Packet Generator interprets the query tree and generates a query packet as an unit of relational algebraic operation. A query packet consists of an IDC packet, a procedure packet and a system packet. The IDC packet contains information of the source relations and specifications of the filtering and clustering conditions. The procedure packet contains information about the relational operation to be executed. The system packet contains information necessary for parallel processing and staging buffer memory management. Employing system packet information, the FDS Runtime manager initiates the IDC, controls the synchronization of the multiple processors and manages the space of the staging buffer memory through FDS Kernel. Each of the Packet Handler invokes an operation described in the procedure packet sent by the FDS Runtime Manager in parallel. Finally, the Packet Handler gets a procedure packet. IDC is activated by the FDS Runtime Manager, reads relations from disks and generates data clusters in the staging buffer memory on the fly. Once a relation is loaded on the staging buffer memory, each Packet Handler gets clustered tuples from the staging buffer memory to its own local memory and processes the bucket in parallel.
Since we employ the fine-hash algorithm, the relation into buckets, since certain buckets may get more tuples than others. This nonuniformity of bucket size raises several problems, such as bucket overflow. For simplicity, we take the Grace Hash Algorithm in the following discussion. The Grace Hash Algorithm consists of two phases: the data split phase and the join phase. At the split phase, relations are partitioned into many clusters (I/O clusters), smaller in size than the staging buffer memory. The joining phase starts after the data partitioning, with the pair of I/O clusters of both relations being read from the disks and the join operation executed. In this paper, we adopt the fine-hash algorithm for processing I/O clusters. That is, we cannot guarantee the uniformity of a data distribution once we apply a hash function in order to split the relation into buckets, since certain buckets may get more tuples than others. This nonuniformity of bucket size raises several performance degrading problems, such as bucket overflow.

As for hash-based join algorithms, the Grace Hash [3] and the Hybrid Hash Algorithms [4] are well-known examples. The Hybrid Hash Algorithm is the hybrid form of the Simple Hash and the Grace Hash Algorithms, and shows better performance than the Grace Hash Algorithm for small relation sizes. This is because in the Hybrid Hash Algorithm, the writing and reading cost of the first bucket can be eliminated by adopting the Simple Hash algorithm and holding the first bucket on memory until the join phase starts. For large relations, both algorithms exhibit almost the same performance, since almost all buckets are written into the disks and the advantage of holding one bucket on memory is relatively small. For large relations, both algorithms exhibit almost the same performance, since almost all buckets are written into the disks and the advantage of holding one bucket on memory is relatively small.

In this section, we evaluate the effect of data distribution nonuniformity on the performance of hash-based join algorithms. Almost all the performance evaluations of these algorithms have been done so far by using uniformly distributed data. It is assumed that source relations are uniformly partitioned into buckets. However, this assumption may not be satisfied in actual database environments. That is, we cannot guarantee the uniformity of data distribution when we apply a hash function in order to split the relation into buckets, since certain buckets may get more tuples than others. This nonuniformity of bucket size raises several performance degrading problems, such as bucket overflow.

As for hash-based join algorithms, the Grace Hash [3] and the Hybrid Hash Algorithms [4] are well-known examples. The Hybrid Hash Algorithm is the hybrid form of the Simple Hash and the Grace Hash Algorithms, and shows better performance than the Grace Hash Algorithm for small relation sizes. This is because in the Hybrid Hash Algorithm, the writing and reading cost of the first bucket can be eliminated by adopting the Simple Hash algorithm and holding the first bucket on memory until the join phase starts. For large relations, both algorithms exhibit almost the same performance, since almost all buckets are written into the disks and the advantage of holding one bucket on memory is relatively small.

As for hash-based join algorithms, the Grace Hash [3] and the Hybrid Hash Algorithms [4] are well-known examples. The Hybrid Hash Algorithm is the hybrid form of the Simple Hash and the Grace Hash Algorithms, and shows better performance than the Grace Hash Algorithm for small relation sizes. This is because in the Hybrid Hash Algorithm, the writing and reading cost of the first bucket can be eliminated by adopting the Simple Hash algorithm and holding the first bucket on memory until the join phase starts. For large relations, both algorithms exhibit almost the same performance, since almost all buckets are written into the disks and the advantage of holding one bucket on memory is relatively small.

In this section, we evaluate the effect of data distribution nonuniformity on the performance of hash-based join algorithms. Almost all the performance evaluations of these algorithms have been done so far by using uniformly distributed data. It is assumed that source relations are uniformly partitioned into buckets. However, this assumption may not be satisfied in actual database environments. That is, we cannot guarantee the uniformity of data distribution when we apply a hash function in order to split the relation into buckets, since certain buckets may get more tuples than others. This nonuniformity of bucket size raises several performance degrading problems, such as bucket overflow.

As for hash-based join algorithms, the Grace Hash [3] and the Hybrid Hash Algorithms [4] are well-known examples. The Hybrid Hash Algorithm is the hybrid form of the Simple Hash and the Grace Hash Algorithms, and shows better performance than the Grace Hash Algorithm for small relation sizes. This is because in the Hybrid Hash Algorithm, the writing and reading cost of the first bucket can be eliminated by adopting the Simple Hash algorithm and holding the first bucket on memory until the join phase starts. For large relations, both algorithms exhibit almost the same performance, since almost all buckets are written into the disks and the advantage of holding one bucket on memory is relatively small.

As for hash-based join algorithms, the Grace Hash [3] and the Hybrid Hash Algorithms [4] are well-known examples. The Hybrid Hash Algorithm is the hybrid form of the Simple Hash and the Grace Hash Algorithms, and shows better performance than the Grace Hash Algorithm for small relation sizes. This is because in the Hybrid Hash Algorithm, the writing and reading cost of the first bucket can be eliminated by adopting the Simple Hash algorithm and holding the first bucket on memory until the join phase starts. For large relations, both algorithms exhibit almost the same performance, since almost all buckets are written into the disks and the advantage of holding one bucket on memory is relatively small.

In this section, we analyze the performance of FDS-R1I when the source relations are smaller in size than the staging buffer memory and the join attribute values are nonuniformly distributed. This assumption implies that only the join phase of the hash-based join algorithm will be executed. In order to execute a join operation on FDS-R1I, source relations are partitioned into buckets by IDC and each bucket on the staging buffer memory is processed in parallel by multiple processors. We begin by describing the parallel processing of buckets on FDS-R1I and, then, report the measurement results and discuss the efficiency of parallel processing.

4.1 Parallel Processing of Buckets

As for hash-based join algorithms, the Grace Hash [3] and the Hybrid Hash Algorithms [4] are well-known examples. The Hybrid Hash Algorithm is the hybrid form of the Simple Hash and the Grace Hash Algorithms, and shows better performance than the Grace Hash Algorithm for small relation sizes. This is because in the Hybrid Hash Algorithm, the writing and reading cost of the first bucket can be eliminated by adopting the Simple Hash algorithm and holding the first bucket on memory until the join phase starts. For large relations, both algorithms exhibit almost the same performance, since almost all buckets are written into the disks and the advantage of holding one bucket on memory is relatively small.
The method for processing data whose size fits into the staging buffer memory on FDS-RI is illustrated in Fig.4. The execution of a relational operation consists of 2 steps: the staging step and the processing step. At the staging step, IDC reads relations from the disk, hashes them building a hash table, and generates buckets in the staging buffer memory. Thus, relations are partitioned into small buckets whose tuples have the same hash value. When reading a relation, IDC examines each tuple according to specified parameters, filtering some attribute fields and applying a hash function to a target attribute field on the fly. At the processing step, each processor gets a bucket from the staging buffer memory into its own local memory as shown in Fig.4. All processors process buckets on their local memory in parallel and write back the results to the disks.

4.2. Effect of Hash Function Nonuniformity

4.2.1. Measurement Environment

We use the following join query for performance measurement.

\[
\text{range of } e \text{ is relation } R \quad \text{range of } f \text{ is relation } S
\]

\[
\text{retrieve } (e.a0, f.a1) \text{ where } e.a0 = f.a0
\]

In this evaluation, we analyze measurement results on FDS-RI in order to clarify the effect of hash function nonuniformity. In the total cost of processing buckets, the costs of reading the data from disk and writing the results back to disk depend only on the amount of data and are not related to the data distribution. Therefore, we analyze only the cost of processing buckets on the staging buffer memory, and excluded the I/O cost. The attribute \( a0 \) is a 2-byte integer and its value is nonuniformly distributed, as illustrated in Fig.5. Each bucket size is calculated according to the equations below.

\[ h: \text{number of buckets} \]

\[ R: \text{size of relation } R (= \text{relation } S) \]

\[ \sigma: \text{standard deviation} \]

\[ i: \text{index of buckets } (0 < i < h) \]

\[ S(\sigma) = \frac{1}{\sigma \sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{(x-\sigma)^2}{2\sigma^2}} dx \]

\[ \frac{1}{\sigma \sqrt{2\pi}} \int_{-\frac{h}{2}}^{\frac{h}{2}} e^{-\frac{(x-\sigma)^2}{2\sigma^2}} dx \]

\[ \text{Size of the } i\text{th bucket} = \frac{16}{S(\sigma)} \times 2R \]

4.2.2. Performance Analysis

Fig.6 shows measurement results of execution time on FDS-RI with a normal distribution. In this measurement, the relations \( R \) and \( S \) have 1,000 tuples each and the staging buffer memory size is 512 KB. Each tuple is 100 bytes long and the number of buckets is 20. We vary the number of processors from 1 to 3. In Fig.6, the horizontal axis shows the standard deviation and the vertical axis shows the execution time. Since we use a normal distribution, the data are skewed more heavily as the value of the standard deviation gets smaller. In Fig.6, we note that each of the three curves approaches a horizontal line, as the standard deviation increases. Because the data are distributed almost uniformly in the case of a large standard deviation, the horizontal line represents the performance with a uniform data distribution, which is the optimal performance. That is, the performance in these cases is coincident with that of the uniform data distribution as the standard deviation increases. We also note that the performance is very much improved when the number of processors increases. As shown in Fig.6, in the case of three processors, when the standard deviation is 2.0, the execution time is almost equal to that of the uniform distribution. Even when the standard deviation is 1, the performance of three processors is better than the best performance of two processors. This means that parallel processing is very effective despite of the nonuniformity of the hash function. Thus, for further increasing in the number of processors, we can expect the performance with a nonuniform distribution to be almost as good as that of a uniform distribution, even if the standard deviation is small. The effect of the hash function nonuniformity on the performance could thus disappear.

5. Nonuniformity of the Split Function

We reported the effect of the hash function nonuniformity in the previous section. In this section, we analyze the nonuniform distribution performance when the source relations are much larger in size than the staging buffer memory. For large relations, the hash-based algorithms attain high performance if the data distribution is uniform. This is because the hash-based algorithm splits the source relations into small clusters whose sizes fit into the staging buffer memory, by using a split function. Since the join attribute values of each I/O cluster are not overlapping each other, the join cost for two large relations is reduced to the sum of the join cost for I/O clusters. The split function, however, cannot always equalize the I/O clusters sizes, even if the data distribution is uniform.
Once overflown I/O clusters are generated, they have to be partitioned again, which severely degrades the performance. In order to handle nonuniformly distributed data efficiently, we introduce the Combined Hash Algorithm, which expands the Grace Hash Algorithm.

5.1. Combined Hash Algorithm

Table 1 shows the processing flow of the Combined Hash Algorithm. This algorithm consists of 2 phases: Split Phase and Join Phase.

In the Split Phase, a processing method is selected from the Grace Hash and the Nested Loop Algorithms by comparing their I/O costs at run time. We calculate the I/O costs of both algorithms from the source relation and staging buffer memory sizes, and from the filtering factor. The filtering factor is estimated as follows: the first portion of the source relation is read until the staging buffer memory becomes partially full. Then the filtering factor can be estimated from the size of this first portion before filtering, as well as from the size of its corresponding portion that will be stored in the staging buffer memory after filtering. Once the processing algorithm is selected, the parameter needed for the Join Phase is calculated. When the Nested Loop Algorithm is chosen, we determine the number of outerloops and innerloops. When the Grace Hash Algorithm is selected, the number of I/O clusters is determined.

A comparison of I/O cost for both algorithms is made in terms of the execution times of Nested Loop and Grace Hash Algorithms on FEDS-IIII with uniform distribution (Fig.7). From the figure, it is apparent that when relations are small, the execution time is shorter for the Nested Loop Algorithm. This means that, since the total cost of the join operation is almost equal to the I/O cost, the I/O cost is lower for the Nested Loop Algorithm, in comparison with the Grace Hash Algorithm. For example, assume that the size of the smaller relation (relation R) is as large as that of the staging buffer memory and the filtering factor is 1.0. With the Nested Loop Algorithm, the outer relation (relation R) is read only once and the inner relation (relation S) is read twice. On the other hand, with the Grace Hash Algorithm, both source relations are always read and written for making I/O clusters and all I/O clusters are read again. That is, both relations R and S are read twice and written once, which makes the I/O cost of the Nested Loop Algorithm lower when relations are small. In contrast, when the size of relations is much larger than that of the staging buffer memory, we find from Fig.7 that the I/O cost of the Grace Hash Algorithm is much lower. The I/O cost formulas for both algorithms and the selection criteria based on their I/O costs are described in detail in Appendix A. From the following discussion, we note that the Grace Hash Algorithm is employed when the size of relation R is larger than five times that of the staging buffer memory and that the Nested Loop Algorithm is adopted otherwise. (This criterion is conducted from cond(4) in Appendix A.)

At the Split Phase, the difficulty in partitioning the source relations lies on the generation of I/O clusters which fit into the staging buffer memory. Depending on the nonuniformity of the split function, the I/O cluster may get larger than the available staging buffer memory. In this case, the Split Phase is invoked recursively in order to choose an appropriate method for each overflown I/O cluster again. As mentioned above, the Nested Loop Algorithm is effective when the size of the overflown I/O cluster is smaller than five times that of the staging buffer memory. Here we show how the Combined Hash Algorithm chooses the appropriate method for each overflown I/O cluster. For example, assume that the source relation is partitioned into I/O clusters as shown in Fig.8. The horizontal line in Fig.8 indicates just the size of the first I/O cluster is four times as large as that of the staging buffer memory, the Nested Loop Algorithm is chosen. For the second I/O cluster, the Grace Hash Algorithm is selected, since the size of the I/O cluster is five times as large as that of the staging buffer memory. In this way, the appropriate method is chosen for each overflown I/O cluster in the Combined Hash Algorithm.

After the processing method has been determined and the I/O clusters generated, the Join Phase is executed repeatedly, as many times as the number of generated I/O clusters. The Join Phase is executed in parallel as described in section 4.
Algorithm is considered here.

In the Nested Loop Algorithm, source relations are only read from the staging buffer memory, the Nested Loop Algorithm is chosen.

5.2. the disks and nonuniformity is almost attenuated by parallel processing. Therefore, the performance of the Nested Loop Algorithm does not depend on the split function nonuniformity. That is, on the performance of the Nested Loop Algorithm, we consider only the effect of the hash function nonuniformity. Therefore, in the case of the Nested Loop Algorithm, the performance should not be significantly effected by data distribution (uniform or nonuniform).

5.3. Effect of Split Function Nonuniformity

5.3.1. Measurement Environment

We execute the same join query as that of section 3. In this measurement, 512KB of staging buffer memory and 4 processors are used. The filtering factor is 1.0, and the bucket size is 10 tuples on the average.

Our objective is the measurement for processing various sizes of I/O clusters, so we use the normal distribution[11], as in the section 4, as well as the Zipf-like distribution, developed by G.K.Zipf[14] and often used in database performance evaluations[13,15,16]. In Fig.10, we show the normal distribution of I/O clusters of the relation used in this measurement and in Fig.11, the Zipf-like distribution of I/O clusters. Both relations R and S have the same distribution and are the same size. Since the tuple length is 128 bytes and the size of the staging buffer memory is 512 KB, the number of tuples which can be held on the staging buffer memory is 4000. Therefore, in the case of uniform distribution, all I/O clusters of one relation have 2000 tuples. In the case of normally distributed data, as shown in Fig.10, half of the I/O clusters overflows the staging buffer memory and the other half fits into the staging buffer memory. The sizes of I/O clusters are calculated as follows.

\[ S(i) = \frac{1}{\sigma \sqrt{2\pi}} \int_{-\infty}^{\infty} \frac{1}{\sigma^3} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \, dx \]

\[ \frac{1}{\sigma^2 \sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \, dx \]

Size of the i th I/O Cluster = \[ \frac{S(i)}{S(\infty)} \times R \]

In the case of Zipf-like distribution, as shown in Fig.11, data skew more than that of the normal distribution when the decay factor becomes large. The sizes of I/O clusters are calculated as follows.

\[ x : \text{decay factor (When } x = 0, \text{ the distribution is uniform)} \]

\[ k : \text{rank of I/O clusters } (0 < k < h) \]

Size of the k th I/O Cluster = \[ \frac{R}{k^x \sum_{i=1}^{h} \frac{1}{i^x}} \]

In order to analyze the effect of the split function nonuniformity on the Combined Hash Algorithm, we compare the measurement results of the Combined Hash Algorithm with the results measured using only the Grace Hash Algorithm. If we only use the Grace Hash Algorithm, we have to divide the overflown I/O clusters again, even if the size of the overflown I/O cluster is only twice as large as that of the staging buffer memory. With a uniform data distribution, both the Combined Hash Algorithm and the Grace Hash Algorithm attain the same performance.
5.3.2. Performance Analysis with Varying Relation Size

Fig. 12 shows the performance of the Combined Hash Algorithm and the Grace Hash Algorithm with normal distribution. The result with uniformly distributed data, which yields the best performance, is also plotted as shown in Fig. 12. The standard deviation $\sigma$ is actually varied from 1.0 to 5.0. However, only results for the Grace Hash Algorithm (standard deviation $\sigma = 1.0$) and the Combined Hash Algorithm (standard deviation $\sigma = 1.0, 2.0$) are shown in Fig. 12. This is because the results of the Combined Hash Algorithm (standard deviation $\sigma = 3.0, 4.0, 5.0$) are almost equal to that of the uniform distribution, and the results of the Grace Hash Algorithm (standard deviation $\sigma = 2.0, 3.0, 4.0, 5.0$) are almost equal to the results of the Grace Hash Algorithm (standard deviation $\sigma = 1.0$). As shown in Fig. 12, the total execution time of all methods increases as a linear function of the relation size, because the source relations are very large, and the Grace Hash Algorithm is applied to partitioning the source relations in all cases. We find that the execution time of the Grace Hash Algorithm is always larger than that of the Combined Hash Algorithm.

This is because the overflown I/O clusters are read and repartitioned again in the Grace Hash Algorithm and this repartitioning cost accumulates a big overhead. On the other hand, with the Combined Hash Algorithm, the Nested Loop Algorithm is chosen for each overflown I/O cluster. Therefore, the execution time of the Combined Hash Algorithm is always much better than that of the Grace Hash Algorithm. Especially, when the standard deviation is larger than 2.0, the execution time of the Combined Hash Algorithm with a normal distribution is almost equal to the best performance.

Fig. 13 shows the performance of the Combined Hash and the Grace Hash Algorithms with the Zipf-like distribution. The results with uniformly distributed data are also shown in Fig. 13. The decay factor $\gamma$ is 0.5. We find the same result for the Zipf-like distribution as for the normal distribution. That is, since the Nested Loop Algorithm is applied to each overflown I/O cluster at the Combined Hash Algorithm, the execution time of the Combined Hash Algorithm is almost equal to the best performance.
From these two measurements, we can conclude that the Combined Hash Algorithm is efficient for handling nonuniform data distribution, and there is little effect of the split function nonuniformity on the performance of the Combined Hash Algorithm.

5.3.3. Performance Analysis with Varying Standard Deviation

Fig. 14 shows the performance of the Combined Hash Algorithm with a nonuniform data distribution, compared to that of a uniform distribution. In these measurements, we used 20,000 tuple-relations and varied the standard deviation ($\sigma$) from 0.5 to 5.0.

![Combined Hash Algorithm Execution Time for Varying Standard Deviation](chart.png)

**Fig. 14**: Execution Time for Varying Standard Deviation

When the standard deviation ($\sigma$) is larger than 2.0, the execution time is approximately the same as that of the uniform data distribution. Even if the standard deviation is 1, the execution time increases only 10% of the total execution time with the uniform distribution. Therefore, we conclude that when the Combined Hash Algorithm is used, the split function nonuniformity does not considerably affect the total execution time.

6. Conclusions

We have evaluated the performance of hash-based join algorithms with nonuniform data distribution on FDS-R1I. In the Grace Hash Algorithm, relations are partitioned into I/O clusters by a split function. Each I/O cluster is then staged on the staging buffer memory and a further hash function is applied to execute a join operation. In order to evaluate the effect of a nonuniform data distribution, we considered the nonuniformity of the hash function and the nonuniformity of the split function. We measured the performance of these cases on FDS-R1I, analyzed them in detail, and showed that the Combined Hash Algorithm implemented on FDS-R1I is effective.

As regards the nonuniformity of the hash function, we find that its effect can be diminished by increasing the number of processors and processing buckets in parallel. In FDS-R1I, the performance of three processors with normal distribution is almost equal to that with a uniform distribution, even when the standard deviation is small. Regarding the nonuniformity of the split function, the performance of the Combined Hash Algorithm with a normal distribution is almost equal to that with a uniform distribution. Since source relations are not always partitioned into I/O clusters whose sizes fit into the staging buffer memory, the occurrence of overflowed I/O clusters increases the processing cost. On FDS-R1I, the appropriate method is chosen from the Nested Loop and the Grace Hash Algorithms at run time for each I/O cluster. In comparison with the results obtained by using only the Grace Hash Algorithm, we conclude that the Combined Hash Algorithm can handle nonuniformly distributed data very efficiently.

In this paper, we only consider the performance of the join operation. However, the hash-based algorithm is easily applied to the other relational operations such as the aggregation and the projection (duplicate elimination). We intend to measure the performance of these relational operations with the hash-based algorithm and to analyze their performance in detail. We are also planning to implement the Dynamic Hybrid Grace Hash Algorithm [8,12] on FDS-R1I and to analyze its performance with a nonuniform distribution.

[References]


87
Appendix A  I/O cost formulas for Join Operation and Selection Criteria
A.1 I/O cost of the two algorithms
The I/O cost is usually the most heavy component of the execution cost for the processing of large relations. Therefore, in the following, we examine the I/O cost of join operations in FDS-RH excluding the cost of writing the result relation, which is equal in any case.

We use the following parameters to express the I/O cost of FDS-RH:
- \( R, S \) : size of relation, \( R, S \) in tuples
- \( M \) : size of staging buffer memory in tuples
- \( C_r, C_s \) : size of I/O cluster of relation, \( R, S \) in tuples
- \( f_r, f_s \) : filtering factor of relation \( R, S \)
- \( T\text{coh} \) : cost to initialize the hash table of IDC
- \( SF_{oh} \) : cost to specify the filtering parameters on IDC
- \( RR_{oh}, WR_{oh} \) : overhead cost of read and write operations respectively

There are three cases:
1. \( f_r, R + f_s, S < M \)
   Under this condition, the relations fit into the staging buffer memory.
   \( I/O\text{cost} = TC_{oh} + SF_{oh} + RR_{oh} + \frac{1}{n}R \)
   \( + SF_{oh} + m( RR_{oh} + \frac{1}{m}S) \)
   \( = TC_{oh} + 2SF_{oh} + RR_{oh} + 1 \text{R} + 1 + nS \) eq(1)

2. \( \min(f_r, R, f_s, S) < M \) and \( f_r, R + f_s, S > M \)
   Under these conditions, the I/O cost is minimized by using the Nested Loop Algorithm. The number of outer loops is only one, and the number of inner loops for relation \( S \) is given by \( m = \frac{f_s, S}{M - f_r, R} \)
   \( I/O\text{cost} = TC_{oh} + SF_{oh} + RR_{oh} + \frac{1}{n}R \)
   \( + SF_{oh} + m( RR_{oh} + \frac{1}{m}S) \)
   \( = TC_{oh} + 2SF_{oh} + 1R + 1 + nS \) eq(2)

3. \( f_r, R > M \) and \( f_s, S > M \)
   (a) Nested Loop Algorithm
   - We assume \( f_r, R < f_s, S \). The number \( n \) of outer loops for the smaller relation \( R \) is given by \( n = \frac{f_r, R}{M - \alpha} \) \( (\alpha = 1 \text{ page}) \). The larger relation \( S \) is staged over the rest space of the staging buffer and the number \( m \) of inner loops for relation \( S \) is given by
     \( m = \frac{f_s, S}{M - f_r, R} \)
     \( I/O\text{cost} = TC_{oh} + n( SF_{oh} + RR_{oh} + \frac{1}{n}R \)
     \( + SF_{oh} + m( RR_{oh} + \frac{1}{m}S) ) \)
     \( = TC_{oh} + 2nSF_{oh} + n(m+1)RR_{oh} + 1R + 1 + nS \) eq(1)

   There is another method of Nested Loop Algorithm. When the filtering factor of \( S \) is low, the size of the filtered relation \( S \) is much smaller than the original one. Therefore the filtered relation \( S \) is saved into the disk, and at the second reading of the outer loop, the Task Cycle reads data from the saved filtered relation.
   \( I/O\text{cost} = TC_{oh} + SF_{oh} + RR_{oh} + \frac{1}{n}R \)
   \( + SF_{oh} + m( RR_{oh} + \frac{1}{m}S) \)
   \( + (n - 1)( SF_{oh} + RR_{oh} + \frac{1}{n}R \)
   \( + SF_{oh} + m( RR_{oh} + \frac{1}{m}S) ) \)
   \( = TC_{oh} + 2nSF_{oh} + n(m+1)RR_{oh} + mWR_{oh} \)
   \( + 1R + 1 + nS + 1f_s, S \) eq(2)

   (b) Grace Hash Algorithm
   The I/O cost of this method is composed of two other costs: the cost of generating the I/O clusters and the data processing cost. At the Split Phase, the source relations are read from the disk and the I/O cluster is generated by applying a split function. Here we assume that the data distribution is uniform and the size of all I/O clusters is equal. The number \( h \) of I/O clusters of each relation is given by \( h = \frac{f_r, R + f_s, S}{M} \) and each cluster size by \( C_r = \frac{f_r, R}{h} \) and \( C_s = \frac{f_s, S}{h} \). The number \( r \) of loops for generating the I/O cluster of relation \( R \) is given by \( r = \left( \frac{f_r, R}{M} \right) \), and similarly the number \( s \) is given by \( s = \left( \frac{f_s, S}{M} \right) \)
   \( I/O\text{cost} = TC_{oh} + SF_{oh} + r ( RR_{oh} + \frac{1}{r}R \)
   \( + h( WR_{oh} + \frac{f_r, R}{hr} ) \)
   \( + TC_{oh} + SF_{oh} + s ( RR_{oh} + \frac{1}{s}S \)
   \( + h( WR_{oh} + \frac{f_s, S}{hs} ) ) \)
   \( = (h+1)TC_{oh} + 2(h+1)SF_{oh} + (r+s+2h)RR_{oh} \)
   \( + h(r+s)WR_{oh} + 1R + 1 + nS + 2f_s, R + 2f_s, S \) eq(3)

A.2 Selection Criteria for the two algorithms
We evaluate the I/O cost formulas without the overhead times.

The condition under which the I/O cost of the Nested Loop Algorithm is better than that of the Grace Hash Algorithm is given when eq(1) < eq(3), and we can see this condition as follows.
   \( n < 1 + 2f_s, S < 5 \) cond(4)
In the same way, from eq(2) < eq(3), we can see the condition as follows.

\[ n < 2 + 2 \frac{f_s |R|}{f_s |S|} < 4 \]  \hspace{1cm} \text{cond(5)}

From cond(4) and cond(5), we can find that when we use the Nested Loop Algorithm with the filtered relation S and when the source relation size satisfies the condition \( f_s |R| < 4M \) and \( f_s = f_s \), the I/O cost of the Nested Loop Algorithm is lower.

At last, the following condition determines whether the inner relation S is written back at the Nested Loop Algorithm. When the filtering factor of relation S is low, the I/O cost of the Nested Loop Algorithm which reads the filtered relation S is smaller than that of the original Nested Loop Algorithm. Since eq(2) < eq(1), we get

\[ f_s < 1 - \frac{1}{n} \quad (n \geq 2) \]  \hspace{1cm} \text{cond(6)}

Acknowledgement

This research is partially supported by Denki-Tsushin-Fukyu Foundation, the Murata Science Foundation, Inamori Foundation and Tokyo Denryoku Foundation.