JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 36, 171-190 (2020)
DOI: 10.6688/JISE.202001_36(1).0010

Bucket-Sorted Hash Join

HYUNKWANG SHIN [1], BYUNG-WON ON [2,+], INGYU LEE [3]
AND GYU SANG CHOI [1,+]
[1] *Department of Information and Communication Engineering*
Yeungnam University
Gyeongsan, Gyeongbuk, 38541 Korea
E-mail: shg3786@naver.com, castchoi@ynu.ac.kr
[2] *Department of Software Convergence Engineering*
Kunsan National University
Gunsan-si, Jeollabuk-do, 54150 Korea
E-mail: bwon@kunsan.ac.kr
[3] *Sorrell College of Business*
Troy University
Troy, AL 36082, USA
E-mail: inlee@troy.edu

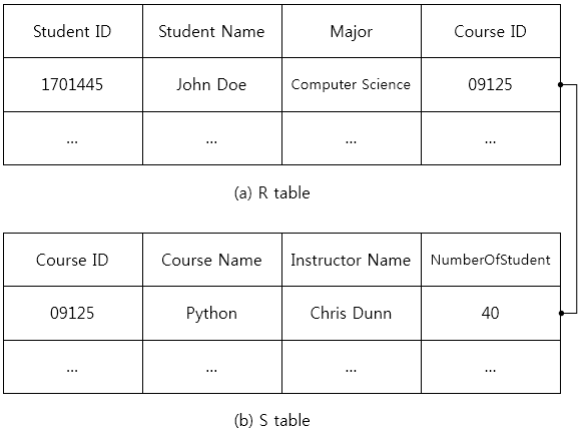
As one of the most important operations in relational database management systems, the join operation is very time-consuming as it needs to merge related records between two tables to produce valuable data. Thus far, several join schemes have been proposed to improve the performance of the join operation, and the hybrid hash-join scheme generally shows the best performance among them. However, this scheme incurs a big overhead during the probing phase as it must scan all records across buckets in the hash table in order to find a corresponding record. In this study, we propose a new hash join scheme, called bucket-sorted hash join, which only maintains records sorted within a bucket. Our proposed scheme can significantly reduce the overhead incurred during the probing phase because all records are sorted within a bucket, and the corresponding records are easily found using a binary search. Our experiments show that the proposed scheme can improve the performance of the join operation by up to 300% in terms of the TPC-H benchmark compared to the hybrid hash join scheme. Thus, the proposed scheme is a viable alternative in hash join operations.

Keywords: database, SQL, hybrid hash join, grace hash join, TPC-H

1. INTRODUCTION

The aim of relational database management systems (RDBMSs) is to store and manage large-scale structured information in a database which consists of relational tables. To *quickly* answer queries in structured query language (SQL) issued by end-users, an RDBMS should support two important operators. The first is the B*–tree–based indexing algorithm [1, 2], and the other is the relational join algorithm. In particular, the join operation is used to connect several tuples r in a table R to other tuples s in a table S if r and s are related to each other. Fig. 1, describes a simple example of a join operation as follows: Given two tables, R and S, assume that the schema data of R are {StudentID, StudentName, Major, CourseID} and those of S are {CourseID, CourseName, InstructorName, NumberOfStudents}. To see the list of all courses that a student, John Doe, signs up for a semester, we can write a simple SQL script program such as "SELECT *R*.StudentName,

Received July 14, 2018; revised January 3 & April 19, 2019; accepted May 27, 2019.
Communicated by Wei-Shinn Ku.
+ The corresponding authors are Gyu Sang Choi and Byung-Won On.



Student ID	Student Name	Major	Course ID
1701445	John Doe	Computer Science	09125
...	...	...	...

(a) R table

Course ID	Course Name	Instructor Name	NumberOfStudent
09125	Python	Chris Dunn	40
...	...	...	...

(b) S table

Fig. 1. Example of joining two tables (R and S).

S.CourseName FROM R, S, WHERE R.StudentName = "John Doe" AND R.CourseID = S.CourseID." In this case, because query responses are usually prepared based on related information from one or more tables in a database, the join operation is one of the most frequently occurring operations in database systems and can largely affect the performance of RDBMSs by consuming a significant amount of system resources, such as CPU cycles, disk bandwidth, and buffer memories.

From the mid-1980s to the early 1990s, basic concepts and concrete algorithms involving join operations were actively developed by the database research community. Currently, rather than conducting new research on fundamental join operation algorithms, most state-of-the-art studies focus on modifying the existing join algorithms that need to be optimized for new computer systems and the latest technology. For example, with the advent of various, next-generation non-volatile memories such as NAND flash memory [3, 4], PCM [5, 6, 7, 8, 9], and STT-MRAM [10, 11], fundamental database algorithms have been re-imagined by numerous researchers and engineers. Existing disk storage has been gradually replaced by NAND flash memory-based solid-state drives (SSDs), instead of HDDs in common computer systems. Because the physical properties of SSDs are completely different from those of HDDs, existing software running on HDDs needs to be modified for SSDs to maximize performance. Furthermore, several recent studies have been conducted on PCM-based join operation algorithms [12, 13]. Efficient join algorithms have been presented for modern multi-core processor systems and even GPU-based computer systems. In the modern big data era, most database researchers are also interested in parallel join algorithms [14] and MapReduce-based join algorithms [15]. Despite the existence of these ongoing studies on new join algorithms suitable for modern systems, there is still room for performance improvement in the traditional join algorithms that have already been widely used in commercial RDBMSs.

In general, regardless of the ANSI-standard or Oracle SQL, there are various types of join operations: INNER JOIN, [LEFT|RIGHT|FULL] OUTER JOIN, CARTESIAN JOIN(EQUI–JOIN), and [COMMON TABLE EXPRESSION|CONNECT–BY] SELF JOIN. In addition, as join algorithms, NESTED LOOP JOIN (NLJ), SORT MERGE JOIN (SMJ), and HASH JOIN (HJ) have been widely employed in commercial RDBMSs. In NLJ, the candidate records are first selected by the WHERE clause of a join query from a driving table R. Then, for each candidate record $r \in R$, a full table scan is performed on an inner table S to match r to $s \in S$. This method is mainly used to join small-sized tables. On

the other hand, both SMJ and HJ are more appropriate for joining large-sized tables. In SMJ, each of two tables, R and S, is first sorted by the WHERE clause of a join query and then a join operation is performed for R and S. However, this method requires additional memory space for the sorting task, and the execution time is insufficient when the join operation cannot be executed immediately. This is because a table with a small number of records must wait for the table with a larger number of records to be sorted first. Similar to SMJ, HJ is typically used when joining large-sized tables. In HJ, given two tables, R and S, R is prepared as a hash table using a hash function if the size of R is smaller than that of S. For each $s \in S$ record, using the same hash function, the relevant records from R are found by scanning the hash table. In most database applications, relational tables are not sorted and equi-join is usually used. Thus, in this work, we focus on the hash join algorithm among the three join algorithms.

In this study, we propose bucket-sorted hash join (BHJ), which is a novel hash join algorithm used to sort records by using a binary search only within a bucket; this considerably reduces its probing time. Existing hybrid hash join algorithms sequentially scan records in a hash table until the corresponding records are matched. Our experimental results show that the proposed hash join algorithm outperforms the existing hash join algorithms by up to 300% because it significantly decreases the probing time in the second phase in HJ.

The rest of this paper is organized as follows: In Section 2, we introduce existing hash join algorithms. In Section 3, we describe the details of the proposed BHJ algorithm to improve the performance of the existing hybrid hash join algorithm. Next, we explain the experimental set-up and discuss the experimental results in Section 4. Finally, we summarize our work and discuss future work in Section 5.

2. BACKGROUND AND RELATED WORK

In this section, we assume that tables R and S will be joined with different schemes, to explain them easily.

2.1 Join Operation

The join operation, which is one of the most important operations in RDBMS, generates valuable data by combining related records between two tables. In addition, this operation is very time-consuming because it needs to compare and merge the records of two tables. Thus, studies have been conducted to improve the join operation [16, 17], and these join algorithms are primarily categorized into three types: (1) nested loop join, (2) merge join, and (3) hash join.

2.2 Nested-loop Join

The nest-loop join algorithm [18] first reads one record from table R and then scans an entire table S to find the corresponding record, as shown in Fig. 2. If the corresponding record is found in table S, the algorithm merges the two records in tables R and S. It repeats this procedure for all records in table R, taking on the form of a nested loop that consists of inner and outer loops. Therefore, the nested-loop scheme generally shows inferior performance compared to other schemes. To improve the performance of the nested-loop scheme, the block nested-loop join scheme had been proposed. It reads multiple records from table R once, and then searches for the corresponding records in table S. This effectively reduces the number of accesses to table S.

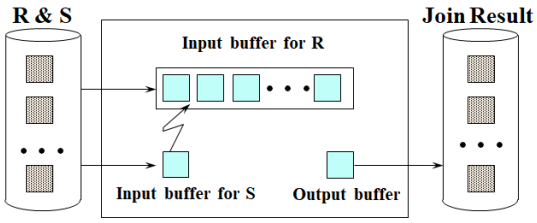


Fig. 2. Block nested-loop join scheme.

2.3 Merge Join

Fig. 3 depicts the merge join operation scheme that first sorts tables R and S before merging them. The scheme reads a record from table R and finds a corresponding record in table S, or vice versa. The algorithm then repeats this operation for all records in tables R or S. The merge join operation will be fast if these tables are indexed by the merged columns in these two tables. However, if the merged columns are not indexed, the scheme must execute an expensive sorting operation on the merged columns. Thus, it can significantly deteriorate the performance of the join operation [19].

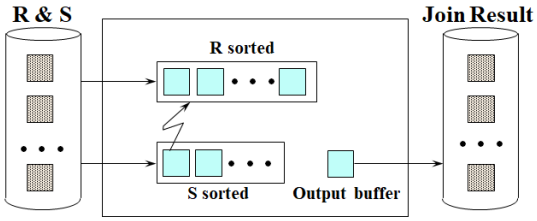


Fig. 3. Merge join scheme.

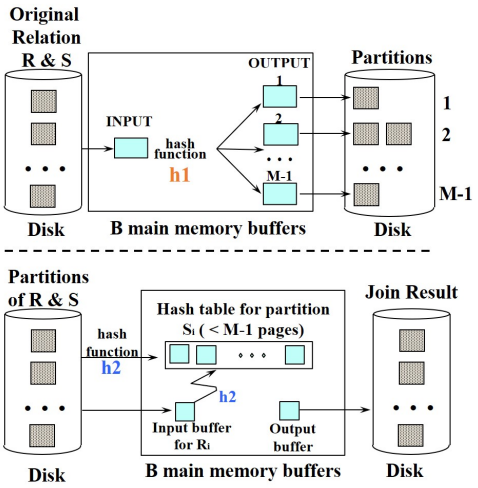


Fig. 4. Hash join scheme.

2.4 Hash Join

The hash join scheme has two steps: build and probe. In the build step, it builds a hash table for the table with the smaller number of tuples. In the probe step, it probes

the constructed hash table using the other table. This can be accomplished with all these operations in memory [20] if the tuples of these tables are small. However, if these tables are too big to fit in the memory, they require expensive disk I/O operations. The early hash join [21] is based on symmetric hash join and uses a single hash table for each input. It also consists of reading and flushing. This join algorithm dynamically customizes the balance between initial performance results and minimum execution time. The grace hash join scheme [22, 23] was proposed to reduce the number of expensive disk I/O operations, as presented in Fig. 4. The scheme consists of two phases. In the first phase, table R is divided into multiple partitions, and each partition is read from the disk and hashed with a hash table. In this paper, a directory (*i.e.*, hash table) consists of multiple directory entries and resides in main memory. Buckets are also allocated in the memory space to hold corresponding records by the hashed value. Each directory entry can have multiple linked buckets, and a hashed record is assigned to each corresponding directory entry. If the allocated memory space for a certain directory entry in the hash table is full, all records of the allocated memory space are written back to the disk and deleted, and the algorithm then continues to process all the tuples in table R. In the second phase, it builds the hash table with a different hash function after reading the tuples in table R from the storage. Then, it probes the built hash table for table R with a tuple of table S, after reading the tuple of table S from the storage. Thus, the grace hash join scheme can significantly reduce the number of disk I/Os because it accesses tables R and S with block I/Os.

The hybrid hash join scheme [24, 25] is a variant of the grace hash join and is intended to utilize memory more efficiently. In hybrid hash join, the first partition of table R is kept in the memory and the hash table is built during the first phase, without writing back to the storage. When hashing the first partition of table S, the algorithm directly probes the hash table of the first partition of table R in the memory. Except for the first partition of tables R and S, other partitions will be processed in exactly the same way as the grace hash join. Thus, the hybrid hash join scheme can eliminate the disk I/Os for the first partition. In this study, we will propose a new approach to improve the performance of the hash join scheme.

3. BUCKET-SORTED HASH JOIN

In this section, we introduce the motivation for our study and propose a novel hash join scheme that sorts records only within the same bucket, instead of sorting all records across multiple buckets

3.1 Motivation

The join operation in SQL produces a new table by combining two or more tables in a RDBMS, and the hash join scheme (as one of the implementations of the join operation) shows high performance compared to other implementations. In addition, there are several schemes that are utilized to implement the hash-join scheme, including the grace hash-join and hybrid hash-join schemes. In this study, we focused on the hybrid hash-join scheme because it will produce higher performance compared to other hash-join schemes.

Next, we measured the execution time of the hybrid hash-join scheme between tables R and S, which comprise the CUSTOMER and ORDERS tables, respectively, in the TPC-H. These tables are shown in Fig. 5. TPC-H, the ad-hoc/decision support benchmark among the benchmarks of the Transaction Processing Performance Council, is extensively used to evaluate database management systems(DBMSs) or computer systems. In this experiment, we generated records for the CUSTOMER and ORDERS tables in TPC-H

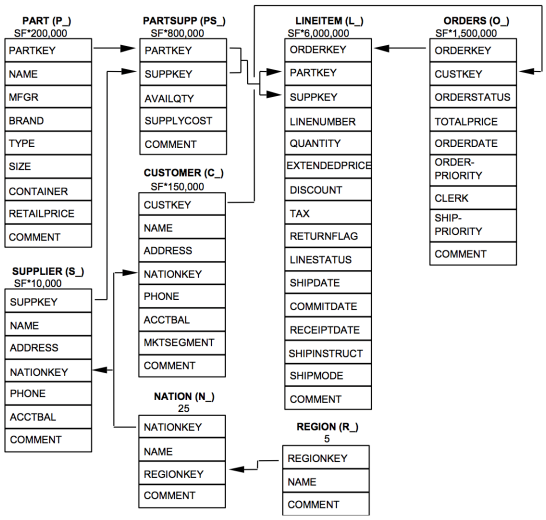


Fig. 5. Schema of tables R and S.

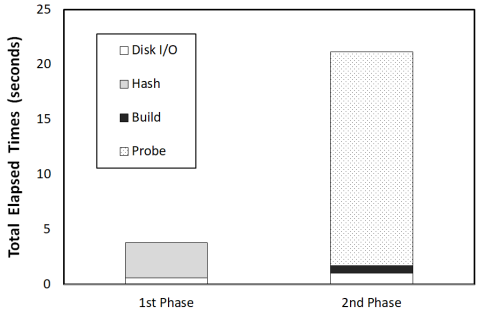


Fig. 6. The breakdown execution time of hybrid hash-join scheme.

by normal distribution. The numbers of records in these tables are 450K and 2250K, respectively. This means that the number of records in table S is 10 times larger than that in table R. In addition, we set the bucket size to 4 Kbytes. All values of the CUSTKEY attribute as the primary key are unique in table R, and all tuples in table R are to be sorted by the primary key. The CUSTKEY attribute in table S can be duplicated, and all tuples in table S are sorted using the ORDERKEY attribute as the primary key.

Fig. 6 shows the execution time of the hybrid hash-join operation. While the elapsed hash time is similar to the disk I/O time in the first phase, the probing time is the relatively dominant component of total execution time in the second phase compared to the building and disk I/O times. This probing operation first selects a specific record in the S table, and then continues to scan the hash table of the R table by comparing the primary key in the R table and the foreign key in the S table until a corresponding record is found in the hash table. This means that the probing operation must scan the hash table of the R table many times. Thus, it is particularly important to minimize the probing time to improve the performance of the hybrid hash-join operation.

3.2 The Proposed Scheme

In this study, we propose a new hash-join scheme, called bucket-sorted hash-join, to sort records only within a bucket in order to reduce the probing time, while the hy-

Algorithm 1: Insert Operation

```
1  Function Insert(Record R, The Bucket B)
2  while (B is full) do
3  |   B : = B->next;
4  |   if (B is equal to the last bucket) then
5  |   |   Allocate a new bucket;
6  |   |   B : = the new bucket;
7  |   |   break;
8  if (B ! = empty) then
9  |   A := The last record in B;
10 |   R : = The new record;
11 |   while (A.Key > R.Key) do
12 |   |   Move right A in the bucket;
13 |   |   A := A's preceding record;
14 |   Add R into A's right side;
15 else
16 |   Add R into B;
```

brid hash-join scheme sequentially scans records in a hash table until the corresponding records are matched.

Algorithm 1 presents the detailed process of an insert operation in the first and second phases in our proposed scheme. To insert a new record into a certain bucket in the proposed scheme, it first checks whether the bucket is full or not. If the bucket is full, it moves to the next bucket. If the bucket is not full, it starts comparing the record to insert with the last record in the bucket. If the key of the inserting record is smaller than the key of the last record in the bucket, it moves the last record to the next, and then compares the preceding record to the last record again. Otherwise, it simply inserts the new record next to the current record. The scheme repeats this process until a new record is properly added to a certain bucket and indicates that the new record will be inserted in the proper slot in the bucket by maintaining the ascending order among records within the bucket.

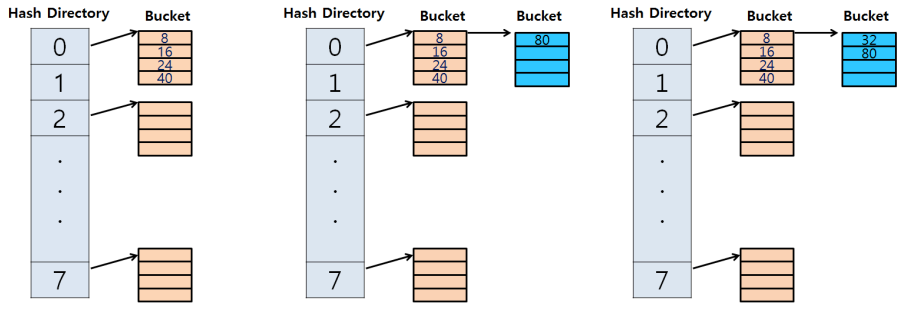
Algorithm 2 presents the detailed process of a probing operation in the second phase in our proposed scheme. In our proposed scheme, we use the binary search algorithm to find the record within a certain bucket instead of scanning records. To probe the record, the scheme visits the first bucket to find the corresponding record using the binary search algorithm. If the corresponding record is found, it combines these two records. Then, it moves to the next bucket to search, and keeps doing so until the last bucket is visited because multiple records with the same key value can exist in some tables.

Next, we explain the proposed scheme with a simple example, and Fig. 7 depicts how the bucket-sorted hash join works. While the proposed scheme is the same as that of the hybrid hash-join in the first phase, in the second phase the 0 directory entry (first partition) initially has one bucket to hold four records, 8, 16, 24 and 40, as shown in Fig. 7 (a). Within the bucket, the four records are stored in ascending order, and now a new record, 80, is inserted in this directory entry. Because the current bucket has no space to store the new record, the new bucket is first allocated and linked, and then the record is inserted in the first slot in the second bucket. Fig. 7 (b) shows the status of the hash table after the new record (80) is added. Now, we add the new record 32 to the first directory entry in the hash table. Because the first bucket is full, the algorithm checks whether the next bucket has available space to store the new record. The record in the first slot is moved to the second slot, and the new record will be added to the first slot in the second bucket to maintain the ascending order among records within the second bucket. Fig. 7 (c) shows the status of the hash table after the new record 32 is added. Thus, a certain record can be easily searched for with our proposed scheme, using the binary search algorithm.

Algorithm 2: Probe Operation

```
1  Function Probe(Record R, Bucket B)
2  while (B ! = The Last Bucket) do
3  |   BinarySearch(R,B);
4  |   B : = B->next;
5  Function BinarySearch(Record R, Bucket B)
6  First : = the first record in Bucket B;
7  Middle : = the middle record in Bucket B;
8  Last : = the last record in Bucket B;
9  while (First <= Last) do
10 |   if (Middle.Key < R.Key) then
11 |   |   First := The next record of Middle;
12 |   else if (Middle.Key == R.Key) then
13 |   |   Combine A and B;
14 |   else
15 |   |   Last := The preceding record of Middle;
16 |   Middle = The middle record Between First and Last;
17 return NULL;
```

This implies that our proposed scheme can significantly reduce the probing time in the second phase, compared to the hybrid hash-join scheme.



(a) Before adding a record 80. (b) After adding a record 80. (c) After adding a record 32.

Fig. 7. Bucket-sorted hash join.

3.3 Time and Memory Space Complexity

Assume that we use a separate chaining to resolve a collision in a hash table, and additional memory spaces other than the key/value pairs in the hash table directory are not needed. The cost of a hash join includes i) partitioning the relations into blocks, ii) creating a hash table entry, iii) probing with a hash table, iv) reading and writing the disk blocks, and v) moving the data to the write memory buffer.

During a partitioning phase, the relations R and S are divided into similar sized partitions, R_i and S_i, using a hash function. Assume that each partition R_i and S_i can be fit into one memory block with a proper hash function. Then, the partitioning cost, $T_{Partition}$, is defined as

$$\begin{aligned} T_{Partition} &= (N_r + N_s) * (T_h * CL * O(B_n) + T_m) \\ &+ (\lceil NB_r/(NB_m - 1) \rceil + \lceil NB_s/(NB_m - 1) \rceil) \\ &* (T_r + T_w) \end{aligned}$$

where N_r and N_s are the number of records in the relations, and T_h and T_m are the hashing and memory access times for a record, respectively. One page block in memory is used for

Table 1. Notations.

Notation	Description
R, S, O	Two relations to join R and S, output relation O.
N_r, N_s	Number of records in relation R and S. Assume $N_s \ll N_r$.
NB_r, NB_s, NB_o	Number of page blocks in relation R and S.
B_n	Number of entries in a bucket.
CL	Average chain length, $\lceil N_s/B_n \rceil$.
q	Utilization factor for a hash table, $q = N_{s_0}/N_s$, i,e, the ratio of the first page block.
NBq_s, NBq_r	Number of pages for the first partition block.
R_i,S_i, O_i	ith partition of relation R, S and O.
NB_m	Number of page blocks in main memory.
T_m	Time to move to a write buffer.
T_h	Time to create a hash table entry for a record, $T_h * CL * O(B_n)$ to generate unsorted bucket and $T_h * CL * O(B_n^2)$ to generate partially sorted bucket.
T_p	Probing time for a record, $T_p * CL * O(B_n)$ for sequential search bucket and $T_p * CL * O(logB_n)$ for binary search bucket.
T_r	Read one page block from disk to memory time.
T_w	Write one page block from memory to disk time.

streaming the relations and the other $NB_m - 1$ page blocks are used to store the partition blocks. Then, the hashing time for a record is $T_h * CL * O(B_n)$ and $(\lceil NB_r/(NB_m - 1) \rceil + \lceil NB_s/(NB_m - 1) \rceil)$ blocks are read from disk into memory during the partitioning phase.

Once the relations are properly partitioned, each partition, S_i, is read into main memory and a corresponding hash table entry is generated by another hash function. Assume S is a smaller size relation, a hash table is generated based on the relation S, one page block is used for streaming the relation, $(NB_m - 2)$ page blocks are used for hash table, and the last one block is used as an output buffer. Then, the hash table generation cost T_{Hash} is defined as

$$\begin{aligned} T_{Hash} &= N_s * (T_h * CL * O(B_n) + T_m) \\ &+ (\lceil NB_s/(NB_m - 2) \rceil) * T_r \end{aligned}$$

where $T_h * CL * O(B_n)$ is a hash entry generating time and $\lceil NB_s/(NB_m - 2) \rceil$ is the number of blocks to read during this phase.

Then, the hashing table is preserved in the memory and the other relation block, R_i, is read into the memory to probe. The probing cost is defined as

$$\begin{aligned} T_{Probe} &= N_r * (T_p * CL * O(B_n) + T_m) \\ &+ (\lceil NB_s/(NB_m - 2) \rceil - 1) * NB_r * T_r \end{aligned}$$

where $T_p * CL * O(B_n)$ is the probing time and $(\lceil NB_s/(NB_m - 2) \rceil - 1) * NB_r$ is the number of blocks to read during the probing.

Finally, the joined result relation block, O_i, is written back to the disk. The cost is defined as

$$T_{Write} = NB_o * T_w.$$

When the two relations, R and S, are pre-sorted, then a bucket searching term, $O(B_n)$ can be replaced with $O(logB_n)$ using a binary search.

- **Hybrid Hash Join.** In a hybrid hash join, the record belonging to the first partition block stays in memory to reduce the disk I/O during a partitioning phase. Assume that q is the ratio for the first partition block of either the relation S or R. Then, each page block from S is read into the memory and a hash table is generated with the records. If the record belongs to the first partition block, then the record is stored in the hash table. Otherwise, the record is moved into a write buffer area. Therefore, the first page block (*i.e.*, $q * N_s$ records) stays in the main memory and the other page blocks (*i.e.*, $(1-q) * N_s$ records) are moved back to a disk through a write buffer. After the partitioning, each page block, except the first block from R, is moved into memory to probe using the hash table. If the record belongs to the first partition block, then record is probed with the generated hash table. Otherwise, the record will be moved to a write buffer area.

  The partitioning phase leaves the records belonging to the first page block in memory. Therefore, the total cost of partitioning is defined as

$$\begin{aligned} T_{Partition} &= (N_r + N_s) * (T_h * CL * O(B_n)) \\ &+ (N_r + N_s) * (1-q) * T_m \\ &+ (\lceil NB_r/(NB_m - NBq_r - 1) \rceil \\ &+ \lceil NB_s/(NB_m - NBq_s - 1) \rceil) \\ &* (T_r + (1-q) * T_w) \end{aligned}$$

  where q is the ratio for the first page block, and NBq_r and NBq_s are the numbers of the first page blocks for relations R and S, respectively.

  Because only $(1-q)$ page blocks are needed to read into a main memory, the cost of hashing table creation is defined as

$$\begin{aligned} T_{Hash} &= N_s * (T_h * CL * O(B_n) + (1-q) * T_m) \\ &+ (\lceil (NB_s - NBq_s)/(NB_m - 2) \rceil) * T_r \end{aligned}$$

  Now, the probing cost is defined as

$$\begin{aligned} T_{Probe} &= (1-q) * N_r * (T_p * CL * O(B_n) + T_m) \\ &+ (\lceil (NB_s - NBq_s)/(NB_m - 2) \rceil - 1) \\ &* (NB_r - NBq_r) * T_r \end{aligned}$$

  Finally, the cost associated with writing the results to disk stays the same as

$$T_{Write} = NB_o * T_w$$

  Similarly, when the relations are already sorted, we can reduce the probing time by using a binary search rather than a linear search. Therefore, the hash table searching term $O(B_n)$ can be replaced by $O(logB_n)$.

- **Bucket-Sorted Hash Join.** During hash table creation, the keys in the hash table are partially sorted in the bucket-sorted hash join. Therefore, instead of a linear search inside a page block, the bucket-sorted hash join uses a binary search on the partially sorted page block. Consequently, it requires more time to generate a partially sorted hash table entries within a bucket, but probing time is reduced by using a binary search within a bucket.

Because partition phase is the same as that of the hybrid hash join, the total cost is defined as

$$\begin{aligned} T_{Partition} &= (N_r + N_s) * (T_h * CL * O(B_n)) \\ &+ (N_r + N_s) * (1-q) * T_m \\ &+ (\lceil NB_r/(NB_m - NBq_r - 1)\rceil \\ &+ \lceil NB_s/(NB_m - NBq_s - 1)\rceil) \\ &* (T_r + (1-q) * T_w) \end{aligned}$$

During a hash table creation phase, blocks should be partially sorted, which requires $O(B_n^2)$ instead of $O(B_n)$ as in hybrid hash join. The cost to create a hash table is defined as

$$\begin{aligned} T_{Hash} &= N_s * (T_h * CL * O(B_n^2) + (1-q) * T_m) \\ &+ (\lceil (NB_s - NBq_s)/(NB_m - 2)\rceil) * T_r \end{aligned}$$

Since the hash table is partially sorted, we can use a binary search within a bucket during the probing phase which requires $O(logB_n)$ rather than $O(B_n)$. Then, the probing cost is defined as

$$\begin{aligned} T_{Probe} &= (1-q) * N_r * (T_p * CL * O(log(B_n)) \\ &+ (\lceil (NB_s - NBq_s)/(NB_m - 2)\rceil - 1) \\ &* (NB_r - NBq_r) * T_r \end{aligned}$$

The cost associated with writing the result blocks to the disk remains the same as

$$T_{Write} = NB_o * T_w$$

In this method, if relations R and S are already sorted, then we can replace the sorting terms $O(B_n^2)$ by $O(1)$ during the hash table creation. The probing phase would also then require $O(logB_n)$ with a binary search .

Lemma 1 *When relation R has more than* $\frac{(B_n^2 - B_n)}{B_n - logB_n} * N_s$ *records, then the hybrid hash join* T_J^H *requires more operations than the bucket-sorted hash join* T_J^S.

Proof: Assume T_J^H is the cost for the hybrid hash join and T_J^S is the cost for bucket-sorted hash join. Then, based on the notation in Table 1 and the above equations, we can define the following relation.

$$\begin{aligned} T_J^H - T_J^S &= N_s * T_h * CL * O(B_n) \\ &+ N_r * (1-q) * T_p * CL * O(B_n) \\ &- N_s * T_h * CL * O(B_n^2) \\ &- N_r * (1-q) * T_p * CL * O(logB_n) \end{aligned}$$

After replacing the fixed costs T_h and T_p with a constant, the hybrid hash join requires $(N_s + N_r) * O(B_n)$ operations, while the bucket-sorted hash join needs $N_s * O(B_n^2) + N_r *$

$O(logB_n)$ operations. Then, the above equation will be changed approximately to

$$\begin{aligned} T_J^H - T_J^S &\approx (N_s + N_r) * O(B_n) \\ &- N_s * O(B_n^2) \\ &- N_r * O(logB_n) \end{aligned}$$

Therefore, the hybrid hash join T_J^H requires a higher number of operations compared to bucket-sorted hash join T_J^S when the number records of relation R is greater than $\frac{(B_n^2 - B_n)}{B_n - logB_n} * N_s$. In conclusion, the cost of sorting during the hash table creation can be recovered within the probing phase by using a binary search scheme in the bucket-sorted hash join when the relation has enough number of records.

4. EXPERIMENT VALIDATION

In this section, we explain the experimental set-up and show the performance evaluation [1].

4.1 Experimental Set-up

In our experiments, we used only two tables, R and S, which stand for CUSTOMER and ORDERS tables, respectively, in TPC-H [26], as shown in Fig. 5. In addition, HHJ_R_S, HHJ_S_R, BHJ_R_S, and BHJ_S_R represent hybrid hash join with building R and probing S, hybrid hash join with building S, and probing R, bucket-sorted hash join with building R and probing S, and bucket-sorted hash join with building S and probing R, respectively. Table 2 lists the configurations of our test-bed, and we determine the hash table size using the buffer size and the sizes of tables R and S in [27].

Table 2. Experimental test-bed configuration.

CPU	Intel(R) Core(TM) i7-4790 Quad-Core, 3.6 GHz
Front-side Bus	1.6 GHz
Main Memory	16 Gbytes
Storage Interface	Serial ATA3
HDD	Seagate Barracuda ST1000DM003 7200 RPM, 1 Tbytes
OS	Cent OS 6.5 (Linux 2.6.34)
File System	ext4

4.2 Performance Comparison

The first experiment executes the join operation between the R and S tables, which have only 450K tuples and a varied number of records from 450K to 2250K, respectively. In the first phase, the join operation reads and hashes tuples of table R into buckets using a certain hash function, and then the hashed tuples are written back to the disk. The same treatment is conducted on the tuples in table S. In the second phase, tables R and S will be used to build and probe a hash table, respectively.

[1] In this paper, the performance is determined by the completion time, which is the total elapsed time to process all requested operations.

Fig. 8 shows the completion times of the grace hash join scheme, hybrid hash join scheme and the proposed scheme, breaking down the total elapsed time for the hashing and disk I/O times in the first phase, and the building, probing, and disk I/O times in the second phase. The figure shows that the first phase hashing time for hybrid hash join scheme is longer than that of grace hash join scheme, but the probing time in the second phase is shorter. This is because by hashing the first partition of table S, the algorithm directly probes the hash table of the first partition of table R in the memory. Furthermore, the grace hash join scheme has a higher total execution time than that of the hybrid hash join. From now, our experiment only compares the performance of our proposed scheme (Bucket-sorted Hash Join) to that of hybrid hash join scheme in this paper. In this experiment, the execution times of the first phases of these two schemes are exactly the same because the operations of the first phases are exactly the same. In addition, the disk I/O times in the first and second phases are negligible.

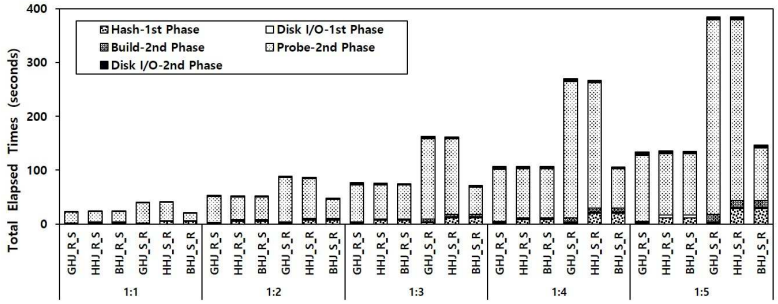


Fig. 8. Analysis of completion times between grace hash join, hybrid hash join and bucket-sorted hash join by varying the number of records in table S.

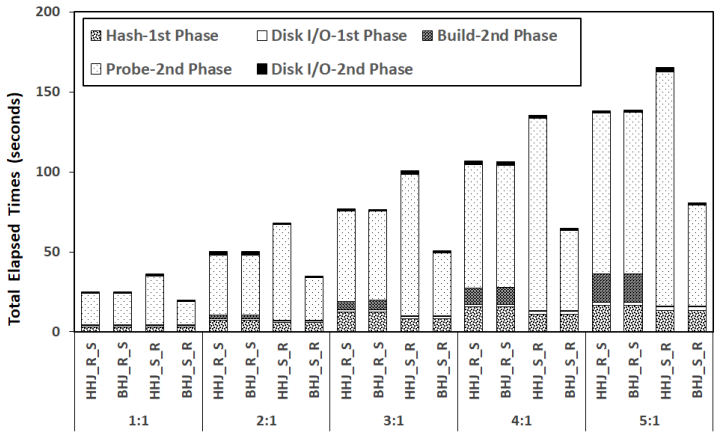
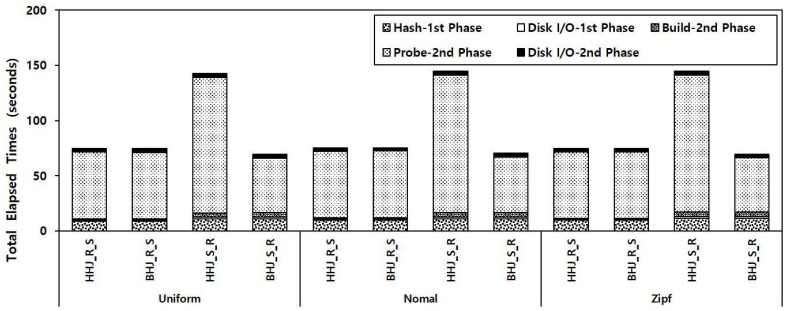


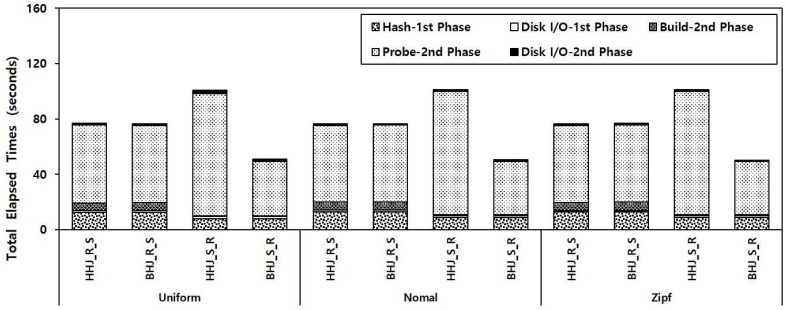
Fig. 9. Analysis of completion times between hybrid hash join and bucket-sorted hash join by varying the number of records in table R.

Interestingly, BHJ_S_R (*i.e.*, our proposed scheme with building S and probing R) shows the best performance because it results in the shortest probing time in this experiment, if the ratio of the tuples in tables R and S is less than or equal to 1-4. Compared to the hybrid hash join scheme, the proposed scheme shows a significantly reduced probing

time in the second phase, whereas the building time is slightly deteriorated. The proposed scheme spends more building time sorting records sorted within each bucket when building S in the second phase, but much less time to probe within a bucket during probing R by the binary search scheme. However, HHJ_R_S (*i.e.*, the hybrid hash join scheme with building R and probing S) shows better performance compared to our proposed scheme if the ratio of the tuples in tables R and S is 1-5. This is because our proposed scheme incurs more overhead to maintain all the records sorted in a bucket. As we expected, the total execution times of our proposed and hybrid hash schemes are similar at building R and probing S because the records in table R are already sorted. In addition, HHJ_S_R (*i.e.*, the hybrid hash join with building S and probing R) shows the worst performance owing to the long probing time in this experiment.



(a) If the numbers of tuples of tables R and S are 450K and 1350K, respectively.



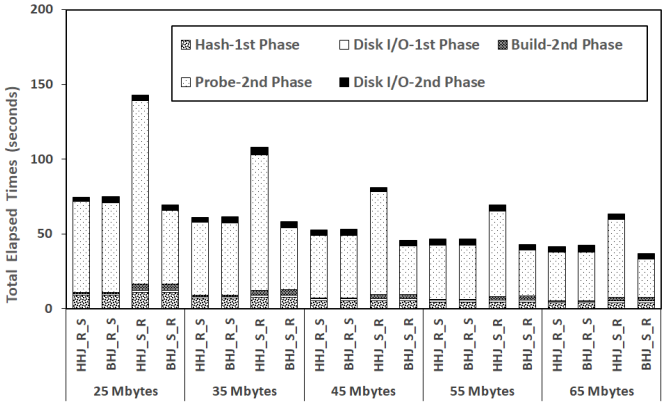
(b) If the numbers of tuples of tables R and S are 1350K and 450K, respectively.

Fig. 10. The analysis of completion times between hybrid hash join and bucket-sorted hash join by varying the distributions.

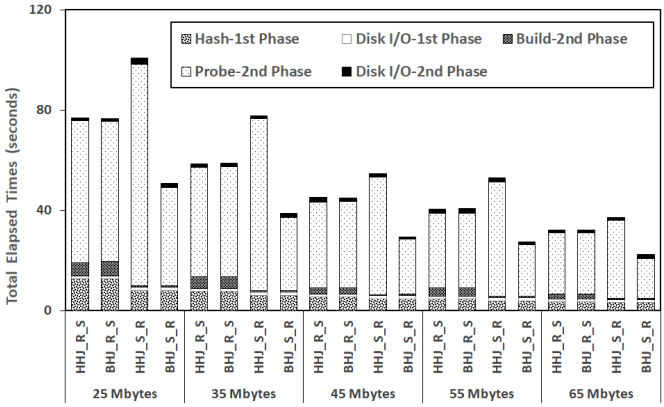
Next, we set table S to only 450K records and table R to a varied number of records from 450K to 2250K in this experiment, and Fig. 9 shows the completion times of the hybrid hash join scheme and the proposed scheme. In this experiment, BHJ_R_S (*i.e.*, the proposed scheme at building S and probing R) outperforms the hybrid hash join scheme by up to 300% because it significantly reduces the probing time in the second phase, whereas HHJ_S_R (*i.e.*, the hybrid hash join scheme at building S and probing R) shows the worst performance. When probing R after building S, the algorithm needs to scan all the buckets linked to the same directory entry to perform a join operation correctly because values in the CUSTKEY attribute can be duplicated.

The proposed scheme can quickly search a corresponding record in a bucket using a

binary search, whereas the hybrid hash join should scan all records across buckets because records in a bucket are not sorted. The proposed scheme and hybrid hash join scheme show remarkably similar performance at building R and probing S. All records in a bucket are naturally sorted at hashing R because all the tuples in table R are maintained in a sorted order, and therefore, the algorithm can search a corresponding record using the binary search algorithm in a bucket when probing R. Now, we additionally use normal and zipf distributions to generate tuples in tables R and S randomly, whereas we used the uniform distribution in previous experiments. In this experiment, the numbers of records in tables R and S are set to 450K and 1350K, respectively, and vice versa. As shown in Figs. 10 (a) and (b), the total elapsed times are similar across different distributions, while our proposed scheme shows the best performance at hashing S and probing R. This result implies that the used hashing function could randomly distribute the tuples in tables R and S in these two schemes because the distributions do not affect the performances of both the bucket-sorted and hybrid hash join schemes in our experiments.



(a) If the numbers of tuples of tables R and S are 450K and 1350K, respectively.



(b) If the numbers of tuples of tables R and S are 1350K and 450K, respectively.

Fig. 11. Analysis of completion times between hybrid hash join and bucket-sorted hash join by varying the buffer size.

We varied the buffer sizes from 25 MB to 65 MB in this experiment. In Fig. 11, the

completion times in two schemes are the fastest at the largest buffer size, but the slowest at the smallest buffer size. This is because the bigger buffer size can hold more records in main memory to incur fewer disk accesses, as the buffer size increases. In this experiment, our proposed scheme also shows the shortest completion times at hashing S and probing R, whereas the hybrid hash join scheme shows the worst completion times owing to a long probing time in the second phase.

Here, we compare the performance impact of bucket size from 4K bytes to 256K bytes when the R and S tables have 450K and 1350K tuples, respectively, in hybrid hash join and in our proposed schemes in this experiment. As shown in Fig. 12, the hybrid hash join scheme shows similar performance as the bucket size increases, because it has no overhead to maintain the records in a sorted order. By contrast, our proposed scheme shows shorter completion times as the bucket size increases up to 64 Kbytes, because the records are maintained in a sorted order in a certain bucket, reducing the probing time during join operation. If the bucket size increases beyond 64 Kbytes, our proposed scheme shows worse completion times. This is because the sorting time within a bucket increases as the bucket size increases. However, our proposed scheme still shows much better performance compared to the hybrid hash join scheme across versatile bucket sizes.

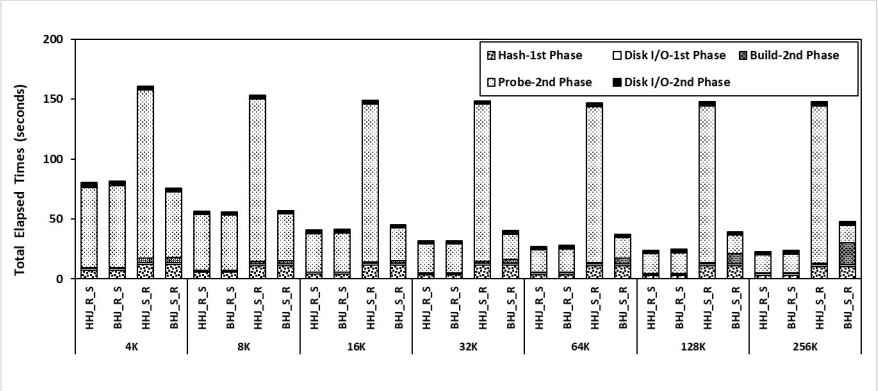


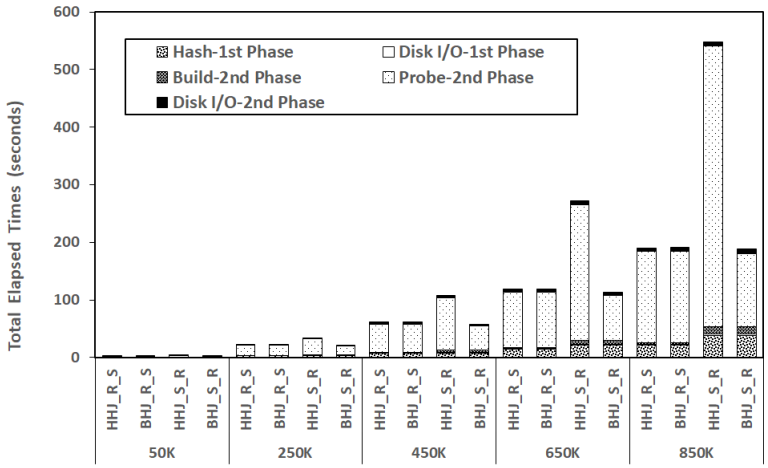
Fig. 12. Analysis of completion times between hybrid hash hoin and bucket-sorted hash join by varying the bucket size.

Fig. 13 (a) shows the completion times as the number of records increases from 50K to 850K in table R, and the ratio of the number of tuples in R and S is 1 to 3. Meanwhile, Figure 13 (b) shows the completion times as the number of records increases from 50K to 850K in table S and the ratio of number of the tuples in R and S is 3 to 1. In this experiment, our proposed scheme improves the completion times by up to 300% compared to the hybrid hash join scheme. As the number of records increases, more buckets are linked to each hash directory entry and additional probing time is required to visit more buckets. The hybrid hash join scheme requires negligible building time but much longer probing time because it must scan all records across multiple buckets. By contrast, the proposed scheme shows much shorter probing time because the binary search algorithm is used to search a bucket.

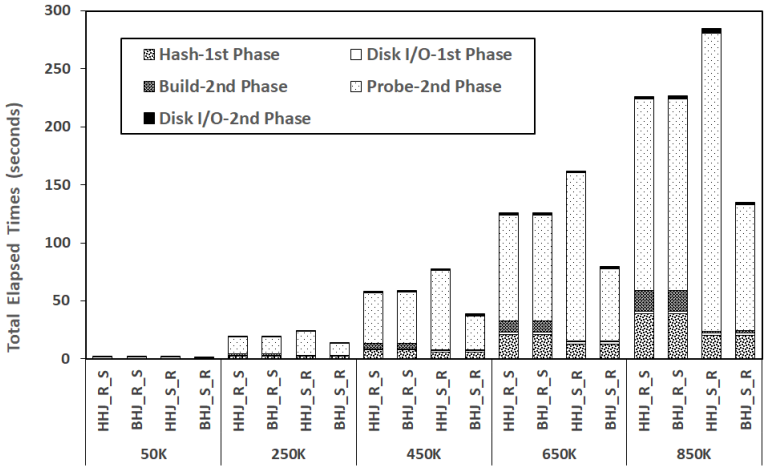
In a nutshell, our proposed scheme shows significant performance improvement compared to hybrid hash join scheme in versatile experimental environments when hashing the table S and probing the table R, and shows better or similar performance compared to hybrid hash join scheme when hashing the table R and probing the table S.

5. CONCLUSION

In this study, we proposed the bucket-sorted hash join scheme, a new hash join scheme designed to improve performance. Our scheme can easily search the corresponding record during probing operations because the records are maintained in a sorted order only within each bucket, not across the buckets. Thus, it can significantly reduce the execution time of a join operation. The experimental results demonstrated that our proposed scheme can improve performance by up to 300 % compared the hybrid hash join scheme.



(a) The ratio of the numbers of tuples of tables R and S is 1 to 3.



(b) The ratio of the numbers of tuples of tables R and S is 3 to 1.

Fig. 13. Analysis of completion times between hybrid hash join and hybrid hash sort join by varying the number of records.

In the future, we plan to extend our work for application to a range-join operation. Furthermore, we plan to propose a completely new join algorithm to significantly improve its performance over that of prior schemes.

ACKNOWLEDGMENTS

This research was supported by the Ministry of Trade, Industry & Energy (MOTIE, Korea) under Industrial Technology Innovation Program. No.10063130, Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2019R1A2C1006159), and the 2018 Yeungnam University Research Grant for Gyu Sang Choi, and the National Research of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2019R1F1A1060752) for Byung-Won On.

REFERENCES

1. D. Comer, "Ubiquitous B-tree," *ACM Computing Surveys*, Vol. 11, 1979, pp. 121-137.
2. G. S. Choi, B. W. On, and I. Lee, "Pb+-tree: Pcm-aware b+-tree," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 27, 2015, pp. 2466-2479.
3. S. Lee, B. Moon, C. Park, J. Kim, and S. Kim, "A case for flash memory SSD in enterprise database applications," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2008, pp. 1075–1086.
4. S. Lee and B. Moon, "Design of flash based DBMS: An in-page logging approach," in *n Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2007, pp. 55-66.
5. F. Bedeschi et al., "A multi-level-cell bipolar-selected phase-change memory," in *Proceedings of IEEE International Solid-State Circuits Conference*, 2008, pp. 428-625.
6. H. Chung et al., "A 58nm 1.8v 1gb pram with 6.4mb/s program bw," in *Proceedings of IEEE International Solid-State Circuits Conference*, 2011, pp. 500–502.
7. Intel, "Intel to sample phase change memory this year," 2010, http://www.dailytech.com/Intel+to+Sample+Phase+Change+Memory+This+Year/article6371.htm.
8. Numonyx, "Numonyx Omneo P8P PCM 128-Mbit parallel phase change memory," 2005.
9. C. Villa, D. Mills, G. Barkley, H. Giduturi, S. Schippers, and D. Vimercati, "A 45nm 1Gb 1.8V phase-change memory," in *Proceedings of IEEE International Solid-State Circuits Conference*, 2010, pp. 270-271.
10. D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi, "Spin-transfer torque magnetic random access memory (stt-mram)," *Journal on Emerging Technologies in Computing Systems*, Vol. 9, 2013, pp. 13:1-13:35.
11. E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, 2013, pp. 256-267.
12. H. Shin, Y. Choi, B.-W. On, and G. S. Choi, "Hybrid hash sort join scheme in database management system," in *Proceedings of Korea Computer Congress*, 2005, pp. 203-205.
13. Y. Choi, B.-W. On, I. Lee, and G. S. Choi, "A comparative study of pram-based join algorithms," *Journal of KIISE*, Vol. 42, 2015, pp. 379-389.

14. D. A. Schneider and D. J. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1989, pp. 110-121.
15. S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2010, pp. 975-986.
16. L. J. Haas, M. J. Carey, M. Livny, and A. Shukla, "Seeking the truth about ad hoc join costs," *The VLDB Journal*, Vol. 6, 1997, pp. 241–256.
17. P. Mishra and M. H. Eich, "Join processing in relational databases," *ACM Computing Surveys*, Vol. 24, 1992, pp. 63-113.
18. K. Bratbergsengen, "Hashing methods and relational algebra operations," in *Proceedings of the 10th International Conference on Very Large Data Bases*, 1984, pp. 323-333.
19. M. W. Blasgen and K. P. Eswaran, "Storage and access in relational data bases," *IBM Systems Journal*, Vol. 16, 1977, pp. 363-377.
20. L. D. Shapiro, "Join processing in database systems with large main memories," *ACM Transactions on Database Systems*, Vol. 11, 1986, pp. 239-264.
21. R. Lawrence, "Early hash join: a configurable algorithm for the efficient and early production of join results," in *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005, pp. 841-852.
22. M. Kitsuregawa, M. Nakayama, and M. Takagi, "The effect of bucket size tuning in the dynamic hybrid grace hash join method." in *Proceedings of International Conference on Very Large Data Bases*, 1989, pp. 257-266.
23. M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of hash to data base machine and its architecture," *New Generation Computing*, Vol. 1, 1983, pp. 63-74.
24. J. M. Patel, M. J. Carey, and M. K. Vernon, "Accurate modeling of the hybrid hash join algorithm," in *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1994, pp. 56-66.
25. D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, *Implementation Techniques for Main Memory Database systems*, 1984, Vol. 14.
26. Transaction Processing Performance Council, http://www.tpc.org/.
27. J. Do and J. M. Patel, "Join processing for flash ssds: Remembering past lessons," in *Proceedings of the 5th ACM International Workshop on Data Management on New Hardware*, 2009, pp. 1-8.

HyunKwang Shin is presently a Ph.D. candidate in Information and Communication Engineering, Yeungnam University. His research interests are text mining, sentiment analysis, social network analysis, machine learning and databases systems.

Byung-Won On is an Associate Professor in Department of Software Convergence Engineering at Kunsan National University, Gunsan, Jeollabuk-do, Korea. In 2007, he earned his Ph.D. degree in Department of Computer Science and Engineering at the Pennsylvania State University at University Park, PA, USA. His research interests are probabilistic models, entity resolution and search, social network analysis and mining, PCM-based in-memory databases, and cutting-edge big data technologies.

Ingyu Lee is an Associate Professor at Sorrell College of Business in Troy University. He received his Ph.D. at the Department of Computer Science and Engineering in Pennsylvania State University on scientific computing algorithm and software. Main research concerns developing efficient scientific computing algorithms based on mathematical modeling using high performance computing architectures and their applications to real world problems including information retrieval, data mining, and social networking

Gyu Sang Choi received the Ph.D. degree in Computer Science and Engineering from Pennsylvania State University. He was a research staff member at the Samsung Advanced Institute of Technology (SAIT) in Samsung Electronics from 2006 to 2009. Since 2009, he has been with Yeungnam University, where he is currently an Associate Professor. His research interests include non-volatile memory, storage systems, big data systems and supercomputing.