Advanced Block Nested Loop Join for Extending SSD Lifetime

Hongchan Roh, Mincheol Shin, Wonmook Jung, and Sanghyun Park

Abstract—Flash technology trends have shown that greater densities between flash memory cells increase read/write error rates and shorten solid-state drive (SSD) device lifetimes. This is critical for enterprise systems, causing such problems as service instability and increased total cost of ownership (TCO) because of SSD replacement. Therefore, numerous studies have focused on decreasing the amount of the DBMS writes. However, there has been no research that focused on decreasing the amount of temporary writes, which are primarily created by join processing. In DBMSs, there are two major join-processing algorithms, i.e., hybrid hash join (HHJ) and sort merge join (SMJ), proven to be the best according to DBMS workload; however, the two algorithms produce temporary writes of intermediate results. Therefore, we instead look to the block-nested loop join (BNLJ); it is well-known that the two algorithms are better than BNLJ, but BNLJ creates no intermediate result writes. It is reasonable to use BNLJ for a major join algorithm if its performance can be enhanced similar to those of HHJ and SMJ, considering BNLJ's advantage of extending SSD lifetimes. Therefore, in this paper, we propose an advanced BNLJ (ANLJ) algorithm that can match the performance of the two main join algorithms.

Index Terms—Query processing, join algorithm, block nested loop join, ANLJ, SSD

1 Introduction

Flash memory technologies have recently advanced substantially in terms of densities between cells and read/write bandwidth. The exponential growth in NAND flash memory capacity seemed like a readvent of Moore's law [1] which are observations that the capacity of semiconductor devices tends to double every 18 months. Storage technology evolution is leading an evolution of computational technologies, creating renovation pressures on upper computer system architectures including host interfaces such as NVMe [2], block layer software stacks, OS file systems, and DBMSs. This pressure exists because software stacks were designed on the basis of the high latencies of hard disk drives (HDDs).

The rapidly growing capacity of flash memory, however, paid a costly price in that write endurance (i.e., the limited number of program/erase cycles of a flash memory block) has worsened and error rates of reading correct data have increased as shown in Table 1. Nonetheless, this trend is not expected to change. Whenever every new flash media generation comes out, flash endurance continues to decrease [3]. As summarized in Table 2, the lower the node size, the lower the write endurance.

Solid-state drive (SSD) manufacturers have been trying to address this growing problem by adding more silicon error-correction code (ECC) chips and firmware logic with more sophisticated garbage collection and wear-leveling algorithms. Despite their best efforts, new-generation SSDs, including new flash chips, are suffering from write-endurance problems. Device lifetimes of SSDs are defined by five-year usage guarantees on the basis of device full writes per day (DWPD) [4], which range from 0.1 to 30 [5]. For consumer SSDs with a SATA I/F, 1.0-3.0 DWPD is the majority, whereas 10 DWPD is the majority for enterprise SSDs with SAS, PCIe, and NVMe I/Fs [5]. One DWPD indicates that the five-year SSD-device lifetime is guaranteed if the SSD is fully written no more than once per day. In other words, the device will be worn-out within one year if the device is fully written five times per day. The SSD base price is increased in proportion to the DWPD value.

Low write endurance of SSDs might not be a problem for personal computer users, but can be a critical problem for enterprise users, causing service instability and increasing total cost of ownership TCO) because of the need to replace SSDs that are worn out completely. In general, the more SSD writes, the greater the TCO.

Given the above, previous studies have focused on decreasing the number of SSD writes. Write operations caused by database management systems (DBMSs) can be classified into the following four categories: DBMS page writes; index page writes; log page writes; and temporary page writes. Previous studies primarily focused on decreasing the number of writes for the first three categories. In-page logging [6] was an attempt to decrease DBMS page writes by writing physiological logs rather than writing entire DBMS pages. Several studies including BFTL [7], IPL B+-tree [8], μ-tree [9], and PIO B-tree [10], focus on decreasing index page writes by buffering index page updates and applying them in batch or logging updated index entries

- *H. Roh is with SK Telecom, Seoul 04539, Korea. E-mail: hongchan.roh@sk.com.*
- *M. Shin, W. Jung, and S. Park are with the Department of Computer Science, Yonsei University, 134 Shinchon-dong Seodaemun-gu Seoul Korea, Seoul 04539, Korea. E-mail: {smanioso, jngwnmk, sanghyun}@yonsei.ac.kr.*

Manuscript received 2 Feb. 2016; revised 2 Dec. 2016; accepted 7 Jan. 2017. Date of publication 11 Jan. 2017; date of current version 3 Mar. 2017. Recommended for acceptance by F. Li
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2017.2651803

TABLE 1
Flash Write Endurance wrt Cell Density [15]

Type	Density	Program/Erase Cycles
SLC	1 bit/cell	100,000
MLC	2 bits/cell	1,000-30,000
TLC	3 bits/cell	800-1,000
3D Flash	2-3 bits/cell	MLC & TLC levels

TABLE 2
Flash Write Endurance wrt Process Node Size [20]

Process Node Size	Capacity per Die	Program/Erase Cycles
30 nm	1,2,4 GB	6,000
25 nm	2,4,8 GB	3,000
20 nm	4,8,16 GB	1,000
15 nm	8,16,32 GB	Less than 1,000

rather than updating entire index pages. More recently, RocksDB [11], which is a variant of level DB [12] and a LSM tree-based [13] key/value store, addressed the level DB's write-amplification problem caused by the compaction operation, which merges index pages in two consecutive levels by adding a feature called universal compaction [14].

Most temporary file writes (i.e., the third category) are created when processing join queries. Because of the limit on DBMS buffer cache sizes, joining large relations requires writing intermediate results to SSDs. The first two categories of writes (i.e., DBMS and index page writes) are essential but for the third category of writes, we propose eliminating these types of writes if possible, even if the additional processing time results; we propose this approach because the effect of needless writes causes increased TCO and service instability.

There are three traditional join-processing algorithms, i.e., grace hash join (GHJ) [16], hybrid hash join (HHJ) [17], and sort merge join (SMJ) [18]. It has been empirically proved that using either of HHJ or SMJ provides the best results, depending on DBMS workloads [19]; however, these algorithms cause intermediate writes, thus causing SSDs to wear out more quickly

As an alternative method, we propose using the block-nested loop join (BNLJ) algorithm [18]. Though it has been proven that SMJ and HHJ are better than BNLJ in terms of I/O and CPU processing time, BNLJ creates no intermediate result writes. It is therefore reasonable to use BNLJ as a major join algorithm if its performance can be improved to be similar to that of HHJ and SMJ, considering BNLJ's advantage of eliminating intermediate writes.

In this paper, we propose an advanced BNLJ algorithm (ANLJ) that can match the performance of the two aforementioned join algorithms with a reasonable range of DBMS DRAM shared buffer sizes. ANLJ decreases the number of loops required by BNLJ to half that of BNLJ thus decreasing the number of relation scans given a 1:N cardinality join condition.

In our experiments, ANLJ not only outperformed BNLJ regardless of various experimental settings, but also showed similar performance to that of GHJ, HHJ and SMJ when the DRAM buffer size was 26 percent of the smaller relation, while with larger buffer sizes ANLJ showed better performance. In addition, when relations were sorted in order of the join key values, ANLJ outperformed the other join algorithms regardless of DRAM buffer size conditions.

These results indicate that ANLJ can be used when the given DRAM buffer size is greater than 26 percent of the smaller relation. Considering current trends in which DRAM prices are dropping and large DRAM capacity is installed on enterprise servers, ANLJ is applicable mostly. Given ANLJ's advantage of eliminating temporary file writes, we claim that even when the DRAM condition is not met, in a certain range from 10 to 25 percent, ANL option should be carefully examined, on the basis of the tradeoffs between the loss in performance and the gain in reducing TCO, thus extending the SSD lifetime.

The remainder of this paper is organized as follows. In Section 2 we introduce the algorithm and data structures of ANLJ. In Section 3, we provide a cost analysis. Next, we empirically evaluate ANLJ in Section 4. In Section 5, we discuss related work. We then conclude our paper and provide avenues for future work in Section 6.

2 ANLJ

In this section, we introduce the differences between BNLJ and our proposed ANLJ, providing the ANLJ algorithm and illustrative examples.

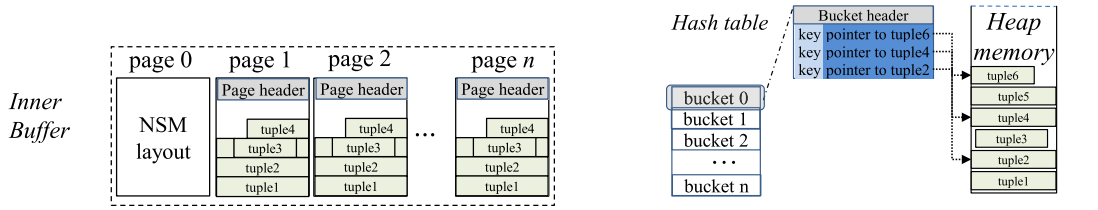
2.1 Differences between BNLJ and ANLJ

ANLJ differs from BNLJ in terms of the following major points.

Contradicting Common Wisdom. ANLJ selects the child relation as the outer relation regardless of whether it is larger than the parent relation. In contrast, BNLJ selects the smaller relation as the outer relation, an approach that has been considered as common wisdom for BNLJ.

Tuple-Recharging Process. ANLJ has a *tuple-recharging* process, i.e., ANLJ continuously reads the outer relation in the middle of each inner loop (more precisely at every step of each inner loop) to recharge the new child tuples into the heap memory as much as the free memory, which were created by deallocating once-joined child tuples on the basis of the observation that once-joined child tuples have no chance of being joined with other parent tuples. Conversely, BNLJ reads the outer relation only at the beginning of each inner loop. Fig. 2 illustrates this tuple-recharging process. Shown in Fig. 2a, from the hash table of outer relation, ANLJ eliminates the once-joined child tuples, which are found by looking up the corresponding child tuples with a foreign key (FK) value identical to the primary key (PK) value of each parent tuple in the inner buffer. New child tuples are allocated into the heap memory as much as the memory size saved by deallocating the once-joined tuples, as shown in Fig. 2b.

Incremental Read. ANLJ does not read child tuples into the outer buffer at once, but instead recharges them by replacing the once joined child tuples with new ones while incrementally reading the child relation. At the end of an inner loop, the outer tuples (i.e., child tuples) in the memory remain in ANLJ whereas BNLJ empties the outer buffer to load new outer tuples.



(a) The inner buffer containing NSM layout pages of the parent relation (b) The Hash table of the child relation

Fig. 1. The inner and outer buffer structure of ANL.

Temporary Buffer. Unless the input tuples are streamed from a pipelined operator, ANLJ needs an input buffer to read outer tuples and temporarily hold them before inserting the tuples into the table and copy to the heap memory.

2.2 ANLJ Algorithm

In this section, we describe the ANLJ algorithm by dividing the process into two parts, i.e., the initiating phase and the iterative phase. Algorithm 1 of Fig. 3 presents our pseudo code for our algorithm.

2.2.1 Initiating Phase

In the initiating phase, the child relation is configured as the outer relation and the parent relation is configured as the inner relation. Next, in the outer buffer, an hash table is configured and the entire memory allocated for the outer buffer is used for the memory space for tuple copy from the read buffer. Next, until the the memory space is full, child tuples are inserted by reading them into a temporary buffer or streaming them from a lower pipelined operator (lines 3-4).

2.2.2 Iterative Phase (Loops)

In the iterative phase, each step of the outer loop starts by scanning of the parent relation from the beginning of its file. Each inner loop ends when the scan for the parent relation reaches the end of the file (lines 5-7).

The outer loop continues until no child tuple exists in the outer hash table after the child relation is completely read (line 5). The outer hash table becomes empty when all child tuples of the child relation are joined with the corresponding parent tuples and are thus all eliminated from the outer hash table.

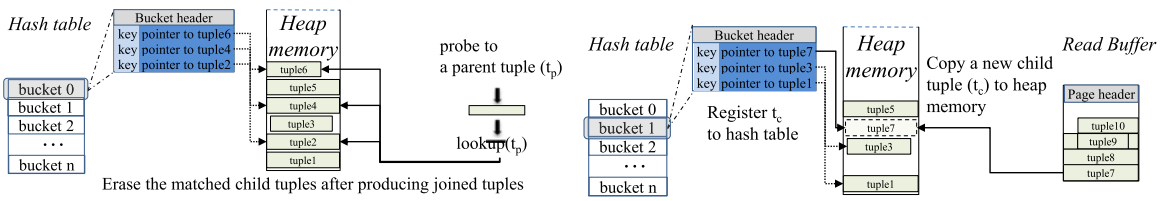
Each inner loop consists of the three processes below.

Inner Load Process (Lines 8-11). The entire memory region of the inner buffer is emptied (line 8). Parent tuples are read from the current offset of the parent relation file to the inner buffer by reading the parent relation as large as the inner buffer. If parent tuples are pipelined from a lower pipelined operator, ANLJ requests streamed parent tuples to a lower pipelined operator and stores the retrieved parent tuples in the inner buffer until it becomes full (lines 9-10).

Lookup Process (Lines 12-17). ANLJ inspects each parent tuple in the inner buffer and finds the corresponding child tuples that have a FK value identical to the PK value of the parent tuple using the outer hash table (lines 11-12). Unless at least one child tuple is found, ANLJ skips the rest of this process and inspect the next parent tuple. Otherwise, for each child tuple found, ANLJ produces a joined tuple by concatenating the child tuple with the parent tuple (lines 14-16). ANLJ then erases the child tuple from the outer hash table and deallocate the tuple pointer (line 16). This process is repeated until all parent tuples in the inner buffer are inspected (line 9). Fig. 2a illustrates this process.

Tuple-Recharging Process (Lines 17-23). To fill the empty slots of the outer hash table, created by the erasure in the lookup process, new child tuples are recharged into the outer hash table by further reading the child relation file from the current offset or by requesting streamed child tuples to a lower pipelined operator (lines 18). Regarding the inner lookup *(lines 20-23)*, for each new child tuple are inserted into the outer hash table (line 21). This process continues until the outer hash table becomes full (line 18). Fig. 2b illustrates this process.

Obsolete-Tuple Deletion Process. Each tuple of child relation has a chance to be joined with tuples of parent relation. Without the FK constraint, however, some of child tuples cannot be joined with tuples of parent relation. These tuples will consume memory and more new child tuples cannot be read. In order to clear the memory space for later tuple-recharging process, we delete these child tuples from the outer hash table and from heap memory after each child tuple is checked with every parent tuples (line 21-23). For this purpose, ANLJ adopted a tuple identifier queue (TIQ) which contains the identifiers of child tuples in arriving order. In the tuple recharging process, each tuple's identifier is enqueued to TIQ. At the end of each inner loop step, we check if the oldest tuple is obsolete by dequeing them from TIQ, and if so, the oldest tuple is deleted from the outer hash table and deallocated from the memory (line 23). In order to check if a tuple is obsolete or not, we adopted a



(a) Lookup on the outer hash table and join (b) Insertion of a new child tuple

Fig. 2. Tuple-recharging process.

Input: parent relation file P, child relation file C, outer buffer B_o, temp buffer B_t, inner buffer B_i, outer hash table H_o, parent tuple t_p, child tuple t_c, child tuple identifier $t_c.id$, TIQ Q
Output: joined tuple t_j

```
1:P.offset := 0, C.offset := 0
2:H_o := configure an hash table in B_o
3:while H_o is not full
4:    t_c := GetTuple(C, B_t), and insert t_c to H_o
5:while H_o is not empty or C.offset < |C|        /* Outer loop */
6:    P.offset := 0
7:    while P.offset < |P|                        /* Inner loop */
8:       empty B_i and H_i                        /* Inner load process */
9:       while B_i is not full
10:         t_p := GetTuple(P, B_i), and insert t_p to B_i if B_i is not full
11:      for each parent tuple t_p in B_i do      /* Lookup process */
12:         look up H_o with the PK value of t_p
13:         if T_c := {t_c| t_c in H_o and t_c.FK = t_p.PK} is not Ø then
14:            for each t_c in T_c do
15:               t_j := Concatenate(t_p, t_c)
16:               produce t_j, erase t_c from H_o, and deallocate t_c
17:      while H_o is not full                    /* Tuple-recharging process */
18:         t_c := GetTuple(C, B_t)
19:         register t_c to the H_o
20:         enqueue t_c.id to Q
21:      while there is obsolete tuples
22:         t_c.id := dequeue from Q
23:         deallocate t_c and erase t_c from H_0
GetTuple(relation, buffer)
24:if tuples are directly read from relation files then
25:   if every tuple in buffer has been probed then
26:      read tuples from relation.offset as large as |buffer|
27:      increment relation.offset by the number of the read pages
28:      return the first tuple of the read tuples
29:   else   probe to the next tuple and return it
30:else   //tuples are pipelined
31:   get a tuple from the lower pipeline operator
32:   increment relation.offset if necessary, and return the tuple
```

Fig. 3. Algorithm 1: ANLJ algorithm.

global variable called *total_step_count* which is increased for every inner loop step (not initialized at the beginning of the next inner loop), and each tuple identifier has its *birth_step_count* which is the total_step_count when the child tuple is inserted to the hash table. Obsolete tuples are filtered by checking if the tuple's *birth_step_count* is less than or equal to *(total_step_count – total #of steps in an inner loop)*.

2.3 Tuple-Recharging Examples

In this section, we use two examples to explain the mechanism by which ANLJ decrease the number of inner loops.

For simplicity, we use the following conditions. First, the number of child tuples matched with each parent tuple is the same in the examples (e.g., for the 1:n cardinality between a parent and child relation, n child tuples are matched to each parent tuple). Second, parent tuples are stored in sorted order of PK values, while child tuples are stored in random order of FK values. Third, the *inner lookup* process of ANLJ (lines 20-22) is not considered. New child tuples fill the free slots created by the previous *lookup process* without checking them with the inner hash table (more child tuples can be recharged into the outer buffer with the *inner lookup* process).

ANLJ decreases the number of inner loops required for a join operation by increasing the number of joined tuples in each inner loop. Because the total number of joined tuples in a join operation is fixed, if the number of joined tuples produced in each inner loop increases, the number of inner loops required for the join decreases.

The number of the joined tuples produced in an inner loop differs according to how often outer tuples are recharged into the outer buffer. The more frequently tuple-recharging occurs, the greater number of joined tuples that are produced in an inner loop. This is why ANLJ performs tuple-recharging at every step of an inner loop. In contrast, BNLJ recharges the outer tuple only once at the beginning of each inner loop. Therefore, at the end of each inner loop, BNLJ produces $|B_o|F_{out}$ joined tuples, which is the number of outer tuples loaded at the beginning (for comparison, we consider BNLJ with the setting choosing the child relation for the outer relation).

We provide two examples to illustrate how the number of joined tuples in an inner loop increases in proportion with the number of tuple-recharging operations performed. In the first example, tuple-recharging is performed twice in an inner loop at the beginning and the midpoint of the inner loop, as illustrated in Fig. 4. In the second example, tuple-recharging is performed three times at the beginning, once at the one-third point of the inner loop, and once at the two-thirds point of the inner loop, as shown in Fig. 5. On the

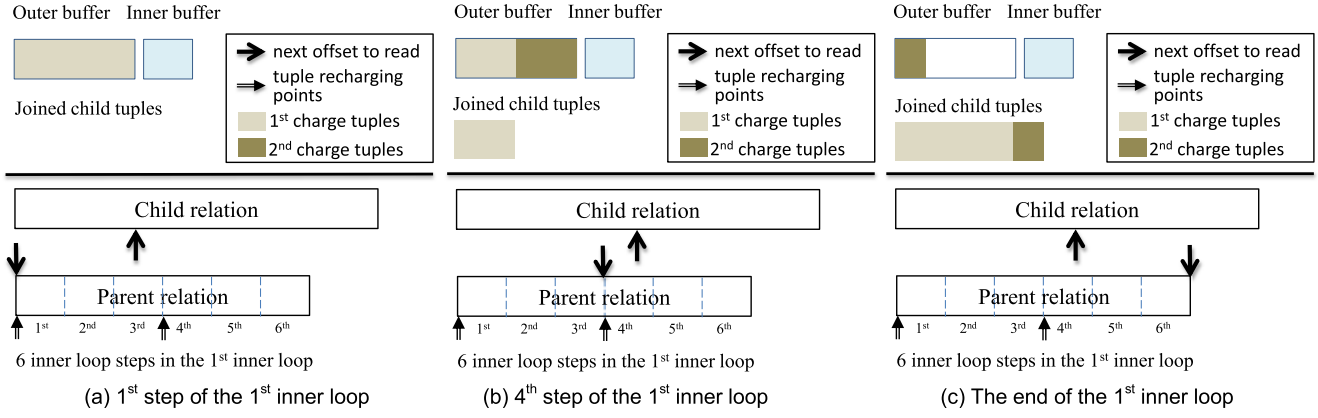


Fig. 4. A tuple-recharging example when child tuples are recharged twice.

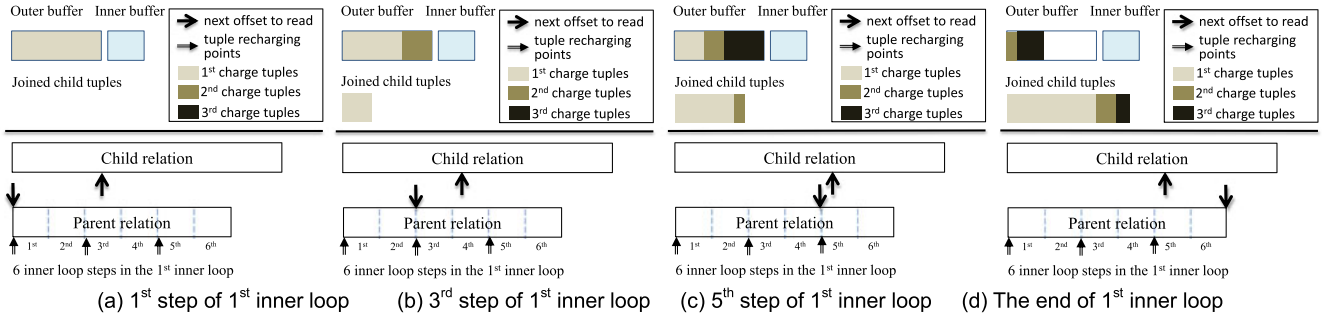


Fig. 5. A tuple-recharging example when child tuples are recharged three times.

basis of the inner buffer size, note that each inner loop has six steps in the examples.

In the first example, $|B_o|F_{out}$ child tuples are loaded into the empty outer buffer at the beginning of the 1st inner loop (the 1st step) as shown in Fig. 4a. We refer to these tuples loaded for the first time as the 1st charge. Until the mid-point of the inner loop (the 4th step), half of the 1st-charge tuples, $1/2|B_o|F_{out}$ child tuples, are matched with the parent tuples from the 1st to 3rd blocks of the parent relation, as shown in Fig. 4b, while the same number of joined tuples $1/2|B_o|F_{out}$ tuples are produced. This occurs because each parent tuple has the same number of matching child tuples (from the first condition) and half of the parent tuples in the parent relation are inspected. Because free tuple slots equal in number to the joined child tuples are created, $1/2|B_o|F_{out}$ child tuples are recharged into the outer buffer. We refer to these tuples as the 2nd charge. Until the end of the inner loop, the rest of the 1st- charge tuples are matched with the parent tuples from the 4th to 6th blocks, producing the same number of joined tuples $1/2|B_o|F_{out}$. Furthermore, half of the 2nd- charge tuples, $1/2^2|B_o|F_{out}$ child tuples, are matched with the parent tuples, producing the same number of joined tuples. Therefore, the number of joined tuples produced from the mid-point to the end becomes $(1/2 + 1/2^2)|B_o|F_{out}$. This process is illustrated in Fig. 4c. In the outer buffer, only the other half of the 2nd charge tuples, $1/2^2|B_o|F_{out}$ child tuples, remain. The total number of joined tuples produced in the inner loop can be calculated by adding the number of joined tuples for each point, $1/2|B_o|F_{out}$ for the mid-point and $(1/2 + 1/2^2)|B_o|F_{out}$ for the end of the inner loop. In total, the number of joined tuples is $5/4|B_o|F_{out}$.

Fig. 5 illustrates the second example in which the tuple-recharging is performed three times. The number of joined tuples can be calculated in the same manner used in the first example. Until the one-third point, shown in Fig. 5b), one third of the 1st-charge tuples, $1/3|B_o|F_{out}$ child tuples, are matched with the corresponding parent tuples, producing the same number of joined tuples. Because the number of free slots is identical to the number of joined tuples, $1/3|B_o|F_{out}$ child tuples are recharged to the free slots and become the 2nd charge. Until the two-thirds point, shown in Fig. 5c, another one third of the 1st-charge tuples plus one third of the 2nd-charge tuples, $(1/3 + 1/3^2)|B_o|F_{out}$ child tuples in total, are matched with the parent tuples, while the same number of tuples (the 3rd charge) are loaded to the free slots. Until the end, shown in Fig. 5d), the last one third of the 1st-charge tuples plus another one third of the 2nd-charge tuples plus one third of the 3rd-charge tuples, $(1/3 + 2/3^2 + 1/3^3)|B_o|F_{out}$ child tuples in total, are matched with the parent tuples. The total number of joined tuples is the sum of the joined tuples for each point, which is summarized as $37/27|B_o|F_{out}$ tuples.

As shown in the two examples, the number of joined tuples in an inner loop increases as tuple-recharging is performed more frequently. In summary, BNLJ, ANLJ with tuple-recharging twice, and ANLJ with tuple-recharging three times produce $|B_o|F_{out}$, $5/4|B_o|F_{out}$, and $37/27|B_o|F_{out}$ joined tuples, respectively.

3 Cost Analysis

We analyze the amount of I/O generated by ANLJ and BNLJ. Our analysis makes two assumptions for simplicity. First, selectivity is not considered. Second, the outer buffer sizes of BNLJ and ANLJ are the same. The buffer sizes can differ but there will be only a slight difference because it is beneficial to BNLJ and ANLJ both to allocate as much main memory as possible to the outer buffer.

3.1 The Cost of BNLJ

BNLJ selects the smaller relation as the outer relation. The outer relation is then scanned once ($|P|$ when $|P| < |C|$), while the inner relation ($|C|$ when $|P| < |C|$) is scanned as many times as the number of inner loops needed. Here the number of inner loops is equal to the number of times a block of the outer relation is loaded into the outer buffer. Therefore, the number of inner loops can be calculated by dividing the smaller relation size ($|P|$ when $|P| < |C|$) by the outer buffer size ($|B_o|$). We can then calculate the amount of I/O needed for BNLJ, i.e., C_b as follows:

$$Cb = |P| + |C| \left\lceil \frac{|P|}{|B_o|} \right\rceil, \ when\ |P| < |C| \quad (1)$$

$$Cb = |C| + |P| \left\lceil \frac{|C|}{|B_o|} \right\rceil, \ when\ |P| \geq |C| \quad (2)$$

3.2 The Cost of ANL

In ANLJ, the child relation is selected as the outer relation, regardless of its size. The outer relation is scanned once ($|C|$), while the inner relation ($|P|$) is scanned as many times as the number of inner loops needed. We introduce a new multiplicative factor called the *Loop reduction factor* (L) to model the extent to which the number of inner loops are decreased in ANLJ compared to BNLJ (L is used as a

TABLE 3
Terminology

Notation	Description
Parent relation	A relation with a primary key corresponding to a foreign key of its child relation
Child relation	A relation that includes the primary key or a candidate key of its parent relation as its foreign key
Child tuple	A tuple from a child relation
Parent tuple	A tuple from a parent relation
Inner relation	A relation which is scanned repetitively inside a loop of BNLJ and ANL
Outer relation	A relation which is scanned once in BNLJ and ANL
Inner buffer	An input buffer for the inner relation
Outer buffer	An input buffer for the outer relation
PK	The primary key of the parent relation
FK	The foreign key of the child relation
$\|P\|$	The parent relation size in number of pages (a typical DBMS page size is 8KB)
$\|C\|$	The child relation size in number of pages
$\|B_o\|$	The outer buffer size in number of pages
$\|B_i\|$	The inner buffer size in number of pages
L	The loop reduction factor
F_{out}	Fanout: the average number of tuples within a page of the child relation

denominator in our formula). ANLJ incrementally reads the outer relation in the middle of an inner loop by replacing the once joined tuples with new ones. Therefore, in an inner loop, ANLJ reads more outer relation pages than BNLJ does. Consequently, the number of inner loops required for ANLJ is less than that of BNLJ. The amount of I/O required for ANLJ, i.e., C_a is calculated as follows:

$$Ca = |C| + |P|\frac{|C|}{|B_o|}\Big/L \tag{3}$$

Unlike (1) and (2), note that (3) has no ceiling function wrapping the second term, because ANLJ can terminate in the middle of the last inner loop when there are no child tuples in the outer hash table, whereas BNLJ terminates only at the end of the last inner loop when the inner relation has been completely scanned.

3.3 Cost Comparison

In this section, we prove that the amount of I/O required for ANLJ is less than that of BNLJ when L (the loop reduction factor) is greater than or equal to 2. In the next section, we also prove that L is close to 2 even in the worst case.

Theorem 1. *Given the notation ($|P|$, $|C|$, $|B_o|$) described in Table 3, the amount of I/O required for ANLJ C_a is less than that of BNLJ C_b when L is greater than or equal to 2.*

Proof. We will show that C_a is less than C_b when $L = 2$. If this holds, C_a is less than C_b when $L > 2$ because C_b does not change, but C_a decreases as L grows. Let $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ denote the ratio of the child relation size to the parent relation size $|C|/|P|$ and the ratio of the parent relation size to the outer buffer size $|P|/|B_o|$, respectively. When $|P| \geq |C|$, C_a is less than C_b of (2) because of the denominator L in C_a. When $|P| < |C|$, the proof for $C_a < C_b$ is as follows.

Using $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, C_b of (1) is transformed into $Cb = |P| + \alpha|P|\lceil\beta\rceil$. Likewise, along with the L value 2, C_a is transformed into $Ca = \alpha|P| + (|P|\alpha\beta)/2$.

We need to prove that $C_a < C_b$ via

$$\alpha|P| + (|P|\alpha\beta)/2 \quad < \quad |P| + \alpha|P|\lceil\beta\rceil$$

After canceling $|P|$ and rearranging terms, we have

$$2\lceil\beta\rceil - \beta > 2 - 2/\alpha \tag{4}$$

For the $\boldsymbol{\alpha}$ value, $\boldsymbol{\alpha}$ is greater than 1 because of given condition $|P| < |C|$. Accordingly, $2/\alpha > 0$ and thus the right side of (4), $2 - 2/\alpha$, is less than 2. Given this, to prove that (4) holds, we need to show

$$2\lceil\beta\rceil - \beta \geq 2 \tag{5}$$

However, $\boldsymbol{\beta}$ is greater than 1 because ad-hoc join algorithms are designed for cases in which the outer buffer cannot hold the entire file of the smaller relation ($|B_o| < |P|$). Therefore, we will show that (5) holds for $\beta > 1$, dividing it into two cases $1 < \boldsymbol{\beta} < 2$ and $\boldsymbol{\beta} \geq 2$.

Because $\lceil\beta\rceil = 2$ for $1 < \boldsymbol{\beta} < 2$, we have

$$\begin{aligned} 2\lceil\beta\rceil - \beta &= 4 - \beta \\ &= 4 - \beta > 2, \ when \ 1 < \beta < 2 \end{aligned}$$

Thus, (5) holds for $1 < \boldsymbol{\beta} < 2$.

Because $\lceil\beta\rceil \geq \beta$, $2\lceil\beta\rceil \geq 2\beta$, and hence we have

$$\begin{aligned} 2\lceil\beta\rceil - \beta &\geq \beta \\ 2\lceil\beta\rceil - \beta &\geq 2, \ when \ \beta \geq 2 \end{aligned}$$

Thus, (5) holds for $\boldsymbol{\beta} \geq 2$. This completes the proof. □

3.4 Loop Reduction Factor

We analyze the loop reduction factor in three cases according to how relation files are organized. The loop reduction factor can differ based on the key order in which tuples are stored and the skew in tuple cardinality (i.e., the skew in the number of child tuples matched to each parent tuple).

We first analyze the best case. The cost of ANLJ is minimized when two relations are stored in the sorted order of key values (the PK value for the parent relation and the FK value for the child relation), regardless of the skew in tuple cardinality. This also includes the case in which one of the relations is reverse sorted because the reverse-sorted relation can be considered as sorted if the relation is read backward from the end to the beginning of the file. In general, this case includes when the child relation has the identical order of key values (or complete reverse order) in which the tuples of the parent relation are stored even if the tuples are stored in a random order of their key values.

Next, we analyze the worst case. The cost of ANLJ is the highest when the tuples of the parent relation are stored in random order of key values as compared to the order of key values in which tuples of the child relation are stored, regardless of the skew in tuple cardinality.

TABLE 4
Symbols Used in the Analysis

Notation	Description
S	The total number of inner loop steps of an inner loop $\lvert P \rvert / \lvert B_i \rvert$
$w_{i,j}$	The jth *wave* of the ith inner loop
$w_{i,j}(x)$	The number of jth wave tuples recharged at the xth step of the ith inner loop
J_i	The total number of joined tuples in the ith inner loop
$J_i(x)$	The total number of joined tuples from 1st step to xth step in the ith inner loop

Third, we analyze the other cases, including the case in which relations are stored in semi-sorted order of key values regardless of the skew in tuple cardinality. In this case, the cost falls between the cost of the best and worst cases.

In the sections that follow, we provide our analyses of the best and the worst cases.

3.4.1 The Best Case

When the two relations are sorted or have identical key order, L is equal to $|C|/B_o$ because the number of inner loops required for the join is one. In this case, ANLJ scans each relation only once regardless of how much main memory has been allocated. Because it operates nearly identical to SMJ, ANLJ reads two relations into their input buffers from the beginning, while producing joined tuples. The input streams are continuously read from the two relations. If all tuples in the inner buffer are inspected, then ANLJ reads the next tuples from the next offset of the inner relation. The outer relation is continuously read through the tuple-recharging process at every step of the inner loop. When the inner relation is scanned once, the outer relation is completely scanned, and no outer tuple is left in the outer buffer. ANLJ terminates with only one inner loop. ANLJ differs from SMJ in that ANLJ uses hash tables to find matching tuples and can process non-sorted relations only if the tuples of the relations are stored in the identical order of their key values.

3.4.2 The Worst Case

When the tuples of the parent relation are stored in a random order of key values compared to the order of key values in which the tuples of the child relation are stored, the loop reduction factor becomes close to two (this will be proved in Theorem 5).

We first analyze the case when there is no skew in the tuple cardinality (i.e., each parent tuple is matched to the same number of child tuples: n child tuples for 1:n cardinality). Then, the skewed case will be analyzed with simulation results.

We will show that the loop reduction factor is close to two by analyzing the number of joined tuples produced in an inner loop for ANL and BNL (the BNL mentioned in this sub-section refer the BNL with the setting choosing the child relation for the outer relation).

We found an interesting *mathematical coincidence* that the core formula (15) in Theorem 5 for the analysis of the loop reduction factor is identical to the formula for the expected length of the sequence generated by the *replacement selection* method [21].

The loop reduction factor is the ratio of the number of the inner loops required for BNL to the number of the inner loops required for ANL. The number of inner loops can be calculated by dividing the total number of joined tuples with the average number of joined tuples produced in an inner loop J_i. For both BNL and ANL, the total number of joined tuples is the same as the total number of the child tuples $|C| \cdot Fout$. For BNL, the average number of joined tuples produced in an inner loop is equal to the number of outer tuples that the outer buffer can hold $|B_o| \cdot Fout$. Therefore, the loop reduction factor of ANL can be defined as the ratio of the average number of the joined tuples in an inner loop of ANL to that of BNL, as in (6).

$$\begin{aligned} L &= \frac{(|C| \cdot F_{out})/(|B_o| \cdot F_{out})}{(|C| \cdot F_{out})/\overline{J_i}} \\ &= \frac{\overline{J_i}}{|B_o| F_{out}} \end{aligned} \tag{6}$$

Hereinafter, we provide an analysis of the number of joined tuples in an inner loop of ANL. The notations used for this analysis is described in Table 2. For the analysis, we employ a concept that we call a *wave*. We denote the tuples loaded into the outer buffer at the beginning of an inner loop as the *initial tuples*. At every tuple recharging point, we denote the newly recharged tuples due to the free slots created by joined initial tuples as *1*st *wave* tuples. We also denote the recharged tuples due to the free slots created by joined *1*st *wave* tuples as 2nd *wave* tuples. In general, *(j + 1)* th *wave* tuples are the recharged tuples due to the free slots created by the joined *j*th *wave tuples*.

Theorem 2. *Given the notation $w_{i,j}$ in Table 2, the total number of joined tuples in the ith inner loop J_i is the sum of the number of the tuples in each wave $w_{i,j}$.*

$$J_i = \sum_j w_{i,j} \tag{7}$$

Proof. The total number of joined tuples is the number of the joined tuples among the initial tuples plus the number of joined tuples in the newly recharged child tuples. Since all the recharged tuples belong to one of the waves, the number of joined tuples in the recharged tuples is the sum of the number of the joined tuples among the tuples of each wave. The number of joined tuples among the initial tuples is identical to the number of 1st wave tuples because ANL recharges the same number of new tuples into the outer buffer as the number of joined tuples among the initial tuples. Likewise, the number of joined tuples among the *j*th *wave* tuples $w_{i,j}$ is identical to the total number of $(j+1)$th *wave* tuples $w_{i,j+1}$. Consequently, the total number of joined tuples is the sum of the number of the tuples in each *wave*. □

Theorem 3. *Given the notations (J_1, $w_{1,1}$) described in Table 2, the total number of joined tuples in the 1st inner loop is:*

$$\begin{aligned} J_1 \approx \int_0^S w_{1,1}(k)dk + \frac{\int_0^S \int_0^l w_{1,1}(k)dkdl}{S} + \frac{\int_0^S \int_0^m \int_0^l w_{1,1}(k)dkdldm}{S^2} \\ + \cdots, where\ w_{1,1}(x) = (|B_o|F_{out})/S \end{aligned} \tag{8}$$

Proof. From (7), the total number of joined tuples in the 1st inner loop is calculated as follows:

$$J_1 = \sum_j w_{1,j} \tag{9}$$

In the 1st inner loop, $|B_o|F_{out}$ initial tuples are loaded at the beginning. At every step of the 1st inner loop, ANL produces $(|B_o|F_{out})/S$ joined tuples due to the uniform tuple cardinality (no skew in the tuple cardinality), thus recharging the same number of 1st wave tuples. Therefore, we have $w_{1,1}(x) = (|B_o|F_{out})/S$.

The number of $(j+1)$th *wave* tuples recharged at the xth step is, the sum of the jth *wave* tuples recharged until the xth step, divided by the total number of the inner loop steps S, due to the uniform tuple cardinality. Therefore we have:

$$w_{1,j+1}(x) = \frac{\sum_{k=1}^{x} w_{1,j}(k)}{S} \tag{10}$$

For a large enough S, we can estimate the series using integrals. We then have the following:

$$w_{1,2}(x) = \frac{\int_0^x w_{1,1}(k)}{S}dk, \;\; w_{1,3}(x) = \frac{\int_0^x \int_0^l w_{1,1}(k)dkdl}{S^2}, \;\; \cdots \tag{11}$$

The total number of jth wave tuples is the sum of the wave tuples recharged at each step.

$$w_{i,j} = \sum_{x=1}^{S} w_{i,j}(x) \approx \int_0^S w_{i,j}(x) \tag{12}$$

This gives the following equations:.

$$w_{1,1} = \int_0^S w_{1,1}(k)dk, \;\; w_{1,2} = \frac{\int_0^S \int_0^l w_{1,1}(k)dkdl}{S}, \; w_{1,3} = \frac{\int_0^S \int_0^m \int_0^l w_{1,1}(k)dkdldm}{S^2}, \cdots \tag{13}$$

By substituting (13) into (9), we have (8). □

Theorem 4. *Given the notation J_i described in Table 2, the total number of joined tuples in the ith inner loop is:*

$$J_i \approx |B_o|F_{out}\left(e^i - i\sum_{k=1}^{i-1}\frac{(i-k)^{k-1}e^{i-k}(-1)^{k+1}}{k!}\right), \quad where\ i > 1\ and\ J_1 \approx |B_o|F_{out}(e-1) \tag{14}$$

Theorem 4 provides the number of join tuples produced in each inner loop. The proof of the theorem is presented in Appendix, , which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2017.2651803.

Theorem 5. *The total number of joined tuples of ANL in an inner loop rapidly converges to twice of the number of joined tuples of BNL in an inner loop as inner loops continue, thus making the loop reduction factor L close to 2.*

Proof. The number of joined tuples in an inner loop of BNL is $|B_o|F_{out}$. If we let, we have $Y(i) = J_i/(|B_o|F_{out})$

TABLE 5
Workload Specification

Name	Relations	Sizes	Sorted
TPC-H	Orders	439 MB	Yes
	Lineitem	1941 MB	
TPC-C	Stock	1069 MB	No
	Orderline	983 MB	
Large TPC-H′	Orders	3.3 GB	No
	Lineitem	14.8 GB	

$$Y(i) = e^i - i\sum_{k=1}^{i-1}\frac{(i-k)^{k-1}e^{i-k}(-1)^{k+1}}{k!}, \quad where\ i > 1\ and\ Y(1) = e-1 \tag{15}$$

$Y(i)$ presents the ratio of the number of joined tuples for ANL to the number of joined tuples for BNL in the ith inner loop. From (15), $Y(i)$ is polynomials in e as follows.

$$Y(1) = e - 1 = 1.71828182...$$
$$Y(2) = e^2 - 2e = 1.9524924...$$
$$Y(3) = e^3 - 3e^2 + 1.5e = 1.99579136...$$
$$Y(i) \to 2 \;\; as \;\; i \to \infty$$

As shown above, $Y(i)$ rapidly converges to 2 as i grows. The formal proof of the convergence of $Y(\infty)$ is described in the previous studies [21], [22]. Accordingly, J_i rapidly converges to $2|B_o|F_{out}$. If we substitute the numerator in (6) with $2|B_o|F_{out}$ then we obtain a loop reduction factor of 2. □

4 Experimental Results

In this section, we evaluate the performance of ANLJ with various workloads on two SSDs and an HDD.

First, we verify the cost analysis by comparing the number of joined tuples in an inner loop measured in our implementation with the number of join tuples predicted using the cost analysis. Second, we compare the performance of ANLJ and BNLJ with three different workloads on a SSD while varying the available main memory size. Third, we evaluate the effects of selectivity and sortedness of relations on performance. In Section 4.5, we compare ANLJ with other major join algorithms including GHJ, HHJ, and SMJ.

4.1 Micro Benchmarks

4.1.1 Settings

Workloads. Table 5 specifies the details of the workloads used in our experiments. We used the TPC-H [23] and TPC-C [24] benchmarks to create input relation files in a PostgreSQL DBMS. We carefully selected the relation pair from each benchmark to evaluate ANLJ in various ways. First, for the sorted workload test, we chose the *orders* and *lineitem* relations from the initial relation files created with the TPC-H benchmark, where the tuples of *orders* are stored in sorted order of its PK *o_order_key* and tuples of *lineitem* are stored in sorted order of its FK *l_order_key* by default. Second, for

TABLE 6
SSD Specifications

	Intel 730 [28]	Micron P300 [25]
Inteface	*SATA III*	*SATA III*
Bandwidth	*Read: 550 MB/s*	*Read: 365 MB/s*
	Write: 470 MB/s	*Write: 275 MB/s*
IOPS	*Read: 89000*	*Read: 60000*
	Write: 74000	*Write: 45200*
Latencies	*Read: 50 μs*	*Read: 153 μs*
	Write: 60 μs	*Write: 118 μs*

the non-sorted workload test, we chose the *stock* and *order-line* relations from the initial relation files created with the TPC-C benchmark, where the tuples of *stock* are stored in sorted order of its PK (*s_w_id*, *s_i_id*) and the tuples of *order-line* are stored in a random order of its FK (*ol_w_id*, *ol_i_id*). Third, for the semi-sorted workload test, we created a semi-sorted version of the TPC-H workload by swapping tuples randomly selected from the initial relations. After randomly selecting two tuples from a relation, we swapped the locations of the tuples and stored them into the changed locations. In this way, a certain percentage (percent) of the tuples of each relation were swapped, thus breaking the sorted order. TPC-H and TPC-C benchmarks were configured with scale factors of 2 and 30 warehouses, respectively, such that the total size of the two relations would be close to 2 GB. All workloads included skew in tuple cardinality.

Devices. We conducted our experiments on a Linux machine (Centos 7, ext4 filesystem) with a 6-core 3.2 GHz CPU (12MB cache) and 16 GB DDR3 RAM. We used a flash SSD [25] with SATA III I/F (refer to Table 6).

Implementation. To achieve a balance between a more realistic implementation and better control of our experiments, we directly used relation files created by the PostgreSQL DBMS, and extracted core module code from the storage manager of PostgreSQL (v8.4.0), implementing ANLJ and BNLJ on the basis of this code base. We also implemented a pipelined selection operator because most current DBMSs implement query-processing algorithms in a pipelined manner [26]. For buffer allocation, we applied the optimized buffer allocation strategies for join processing suggested by [27] to BNLJ. For ANLJ, we allocated one eleventh of the given main memory to the inner buffer and inner hash table, the rest being allocated to the outer buffer.

For pipelining, a selection operator was configured. The selectivity was set to 50 percent by default. The selection operator read the input relation files into the 2 MB input buffer and streamed the filtered tuples to the join operator, as illustrated in Fig. 6. We used non-buffered I/O such that the I/O bypasses the OS file system cache. The ext4 file system was used.

4.1.2 Cost Analysis Evaluation

We evaluated the correctness of the cost analysis of ANLJ by comparing the number of joined tuples produced in the real experiments with the number of joined tuples predicted using our cost analysis. For our cost analysis, the number of joined tuples was calculated by substituting $|B_o|$ and F_{out} in (14) with their actual values from the given experimental settings, for each inner loop from 1st to 12th. In actual experiments, the number of joined tuples was also measured for each inner loop from 1st to 12th using the TPC-C workload. We repeated the experiments here, varying memory size from 32 MB to 44 MB in 4 MB increments.

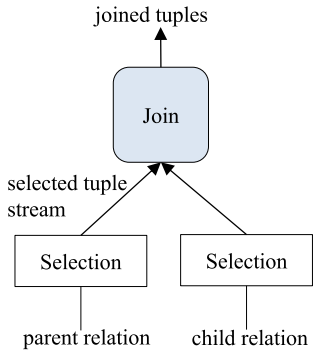


Fig. 6. Pipelining implementation.

Fig. 7 shows our results, where *real* represents the measured values from experiments and *model* represents the calculated values from the analysis. For the four experimental settings (32, 36, 40, 44 MB), the *real* and *model* values showed the same increasing pattern in the number of join tuples with respect to increased number of inner loops. The average gap between the *real* and *model* values was very small, notably 2.56 percent.

4.1.3 Comparing BNLJ with ANLJ while Varying Memory Size

Next, we compared the performance of ANLJ with that of BNLJ using three workloads with varying main memory sizes. For our evaluation with different relation files in terms of sortedness to the key values, we used the *TPC-H*, *Semi-sorted*, and *TPC-C* workloads for sorted, semi-sorted and non-sorted relations files, respectively.

We measured both the total response time and the I/O processing time to provide a more detailed analysis, thereby allowing us to disassemble the total time into CPU and I/O processing times.

For this set of experiments, we increased memory size from 4 MB to 260 MB by 4 MB increments. With more than 260 MB of memory, all tuples of the smaller relation (orders) selected by the selection operation (with default selectivity

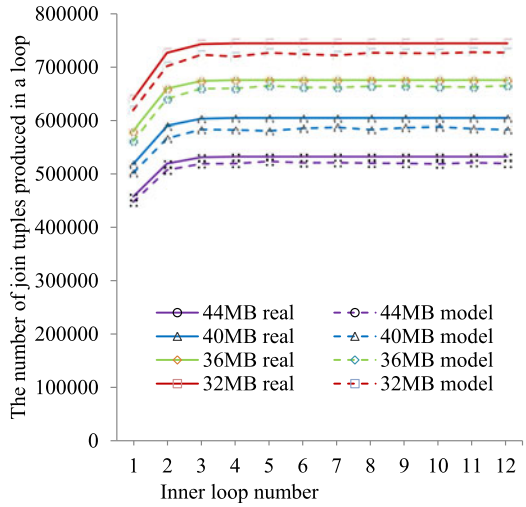


Fig. 7. Cost analysis evaluation.

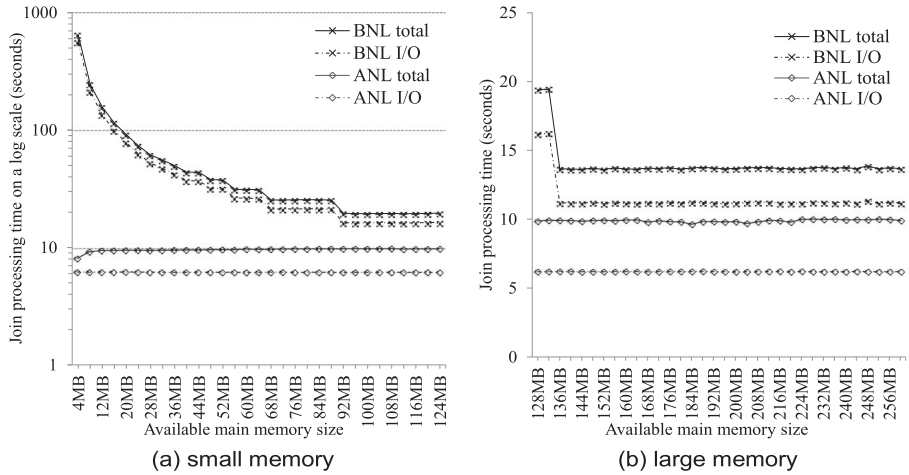


Fig. 8. Comparison with the TPC-H workload.

of 50 percent) were loaded into the outer buffer at once, thus it was meaningless to compare the performance of the two join algorithms.

Fig. 8 shows join processing times of ANLJ and BNLJ on the SSD with different memory sizes with TPC-H workload. Because of the severe performance gap in the small memory region, Fig. 8a shows join processing times on a logarithmic scale. Here, ANLJ showed nearly constant performance. Because the relations were stored in sorted order in the relation files, ANLJ performed like SMJ with a single scan for each relation; however, the performance of BNLJ worsened as memory size decreased because of the increased number of inner loops. Further, ANLJ was up to 77.93 times faster than BNLJ. The performance gap here was relatively steady in the large memory region. ANLJ was 1.36- 1.95 times faster than BNLJ in this large memory region.

The CPU time (i.e., total time minus I/O time) of ANLJ was, on average, 1.4 times less than that of BNLJ in the small memory region; however, in the large memory region, CPU time of ANLJ was 1.45 times greater than that of BNLJ. The increased CPU time of ANLJ occurred primarily because of the tuple-recharging process. For tuple-recharging, ANLJ inserts tuples into and removes tuples from the hash table, whereas BNLJ only inserts the portions of the tuples (i.e., the key and tuple pointer) into its hash table.

The memory size was increased from 8 MB to 560 MB in 8 MB increments. With memory sizes greater than 560 MB, all tuples of the smaller relation (*orderline*) selected by the selection operation were loaded into the outer buffer at once.

Fig. 9 shows the join processing times on the SSD with TPC-C workload. In the small memory region, ANLJ was up to 2.12 times faster than BNLJ, while in the large memory region, ANLJ was 0.98-1.24 times faster than BNLJ. In the entire memory region, ANLJ was faster than BNLJ except when the memory size was 336MB (with a factor of 0.98). Performance gains in the large memory region were small because of the following reasons: first, the I/O cost for reading the outer relation (the first term in (3)) constituted larger portion of the total I/O cost as the memory size increased in ANLJ; second, the CPU time of ANLJ was greater than that of BNLJ (1.28 times on average). Note that only counting I/O time, ANLJ was 1.16-1.46 times faster than BNLJ in the large memory region.

4.1.4 Sortedness and Selectivity

We also investigated the effects of sortedness of relations and the selectivity of the selection operator.

First, to examine the effects of sortedness of relations, we increased the percentage of swapped tuples from 5 to 100 in

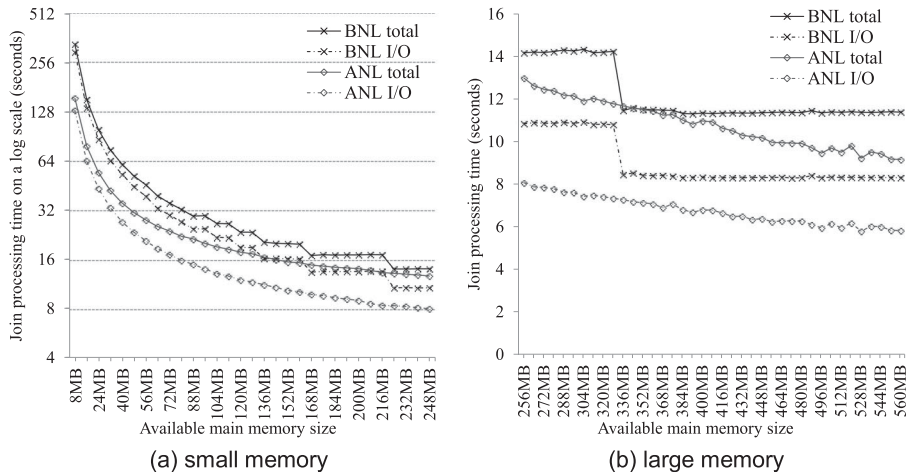


Fig. 9. Comparison with TPC-C workload.

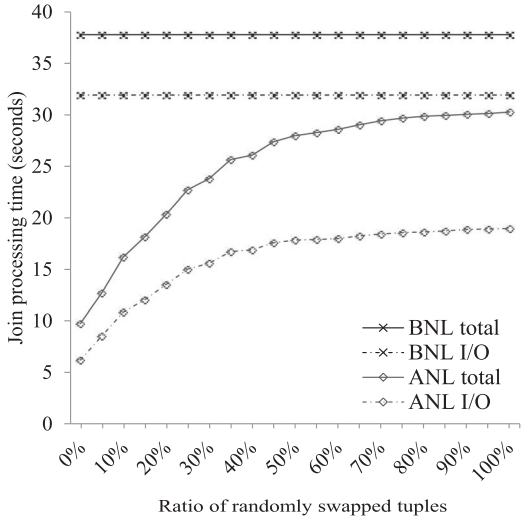


Fig. 10. Comparison varying the sortedness.

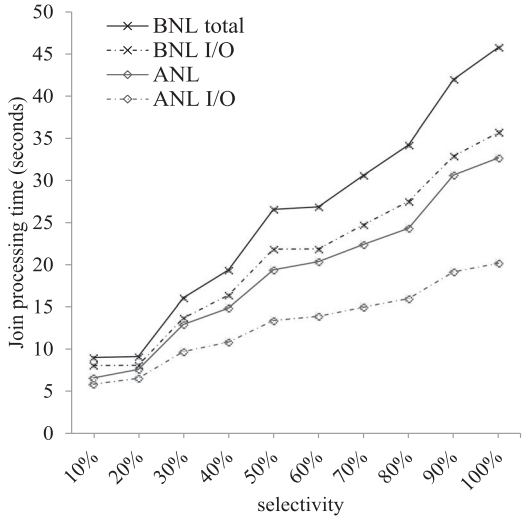


Fig. 11. Comparison varing selectivity.

5 percent increments in the *TPC-H* workload as we created the *semi-sorted* workload (refer to Section 4.1). If all tuples are randomly chosen and swapped, then the tuples of the resulting relation files are stored in a completely random order of the key values. We fixed the memory size at 52 MB.

Fig. 10 shows the join processing times with different degrees of sortedness. The join processing time of ANLJ increased as the percentage of swapped tuples increased but it remained less than that of BNLJ. Once past the 50 percent region, the graphs quickly converged to a certain amount of the join processing time, because the loop reduction factor decreased as the randomness in key values increased, but the loop reduction factor converged close to 2. On the SSD, ANLJ was 1.24-3.83 times faster than BNLJ. Only counting I/O time, ANLJ was at least 1.68 times faster than BNLJ.

Second, to investigate the effects of selectivity, we increased selectivity from 10 to 100 in 10 percent increments in the *TPC-C* workload. We fixed memory size at 104 MB. Fig. 11 presents the join processing times with different selectivity values. The join processing times increased with increased selectivity. ANLJ was 1.20-1.41 times faster than BNLJ on the SSD. The performance gap was nearly steady except at several points. With 10 percent selectivity, ANLJ was 1.36 times faster than BNLJ, while ANLJ was 1.40 times faster than BNLJ with 100 percent selectivity, respectively.

4.2 Comparison with Other Major Join Algorithms

4.2.1 Settings

Workloads. For this experiment, we used the TPC-H [23] with scale factor 16 (refer to Table 5) and created input relation files in a PostgreSQL DBMS (v8.4.0). In order to conduct experiments fairly (since ANLJ performs better in sorted and semi-sorted relations), we altered lineitem and all the tuples of lineitem relations are randomly swapped and thus created non-sorted relations.

Devices. We conducted our experiments on a Linux machine (Centos 7, ext4 filesystem) with a 6-core 3.2 GHz CPU (12 MB cache) and 16 GB DDR3 RAM. We used two flash SSDs. One is Micron P300 SSD [25] with SATAIII host I/F and SLC NAND flash type and the other is Intel 730 SSD [28] with SATAIII host I/Fs and MLC NAND flash type (refer to Table 6). We used one HDD which is 7200 rpm Hitachi Deskstar 7K1000.D [29]

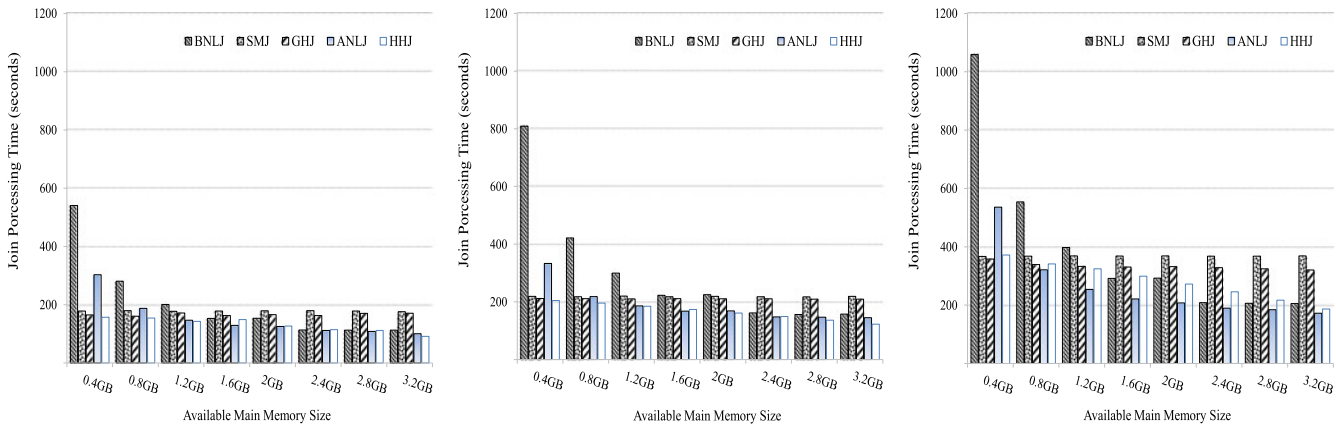
Implementation. we used relation files created by the PostgreSQL DBMS, and extracted core module code from the storage manager of PostgreSQL (v8.4.0), implementing ANLJ, BNLJ, GHJ, SMJ, and HHJ on the basis of it. For buffer allocation, we applied the optimized buffer allocation strategies for join processing suggested by [27]. For ANLJ, we allocated one 11th of the given main memory to the inner buffer and inner hash table, the rest being allocated for the outer relation. The selectivity was set to 100 percent by default. We used non-buffered I/O such that the I/O bypasses the OS file system cache with the ext4 file system. SSDs were preconditioned with fill rates of 70 percent and write-cache of SSD devices was enabled.

4.2.2 Comparing Join Algorithms with Varying Memory Sizes

Finally, we compared the performance of ANLJ with other major join algorithms, i.e., BNLJ, GHJ, HHJ, and SMJ.

Fig. 12 presents total join processing times for the Large TPC-H workload. Regardless of devices, ANLJ showed performance similar to or better than that of GHJ, HHJ, and SMJ when DRAM buffer size was equal to or greater than 24 percent of the smaller relation (i.e., 0.8 GB of DRAM buffer with 3.3 GB of the smaller relation). Especially for I/O times for join processing, ANLJ showed better performance than that of GHJ, HHJ and SMJ, when DRAM buffer size was equal to or greater than 24 percent of the smaller relation.

This result is mainly due to the loop reduction capability of ANLJ. By doing so, ANLJ could reduce the amount of I/O as shown in Table 7, thus leading to the best performance in I/O processing time in Fig. 13. If we sum the page I/O counts of read and writes, excluding 0.4 GB cases, ANLJ has the smallest page I/O count (6041841 for ANLJ, 6337217 for HHJ). Moreover, the write I/O count for ANLJ and BNLJ is zero, which is the key value (prolonging the SSD life time) of these algorithms on SSDs. GHJ and SMJ scan once the relations when making partitions and sorted runs, and writes the temporal relations to the devices and again scan the temporal relation once more, thus resulting two scans and one write for the relation size.

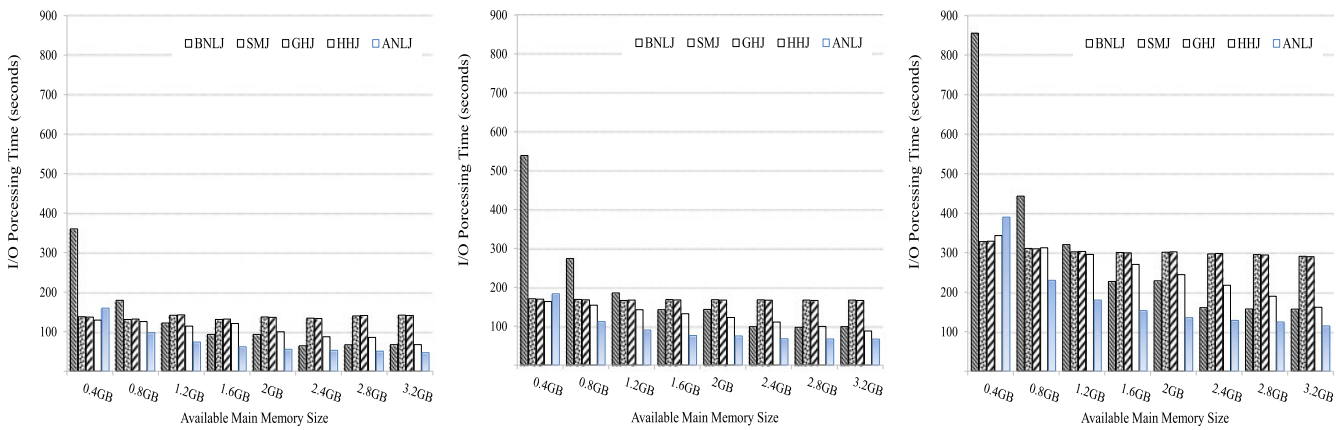


(a) Intel 730 SSD (MLC) (b) Micron P300 (SLC) (c) HDD

Fig. 12. Major Join Algorithm Join Processing Time Comparison with Large TPC-H workload.

TABLE 7
8KB Page I/O Count of Join Algorithms

Memory	BNLJ read	SMJ read	GHJ read	ANLJ read	HHJ read	BNLJ write	SMJ write	GHJ write	ANLJ write	HHJ write
0.4GB	23180849	4687673	4687671	10327451	4518848	0	2363088	2363086	0	2194263
0.8GB	11804705	4687662	4687658	6041841	4330901	0	2363077	2363073	0	2006316
1.2GB	8012657	4687658	4687655	4734654	4137902	0	2363073	2363070	0	1813317
1.6GB	6116633	4687656	4687654	4024988	3932108	0	2363071	2363069	0	1607523
2GB	6116633	4687654	4687653	3610268	3722730	0	2363069	2363068	0	1398145
2.4GB	4220609	4687653	4687652	3439755	3506278	0	2363068	2363067	0	1181693
2.8GB	4220609	4687652	4687652	3302539	3273471	0	2363067	2363067	0	948886
3.2GB	4220609	4687652	4687651	3048058	3036494	0	2363067	2363066	0	711909



(a) Intel 730 SSD (MLC) (b) Micron P300 (SLC) (c) HDD

Fig. 13. Major Join Algorithm I/O Processing Time Comparison with Large TPC-H workload.

If we let the I/O cost for read and write the same, then this can be calculated as 3 relation scans. However, given 1/4 memory of the smaller relation size, ANLJ will have two scans since ANLJ will reduce the number of scans required for BNLJ in half which is 4 scans. This makes ANLJ perform better than GHJ and SMJ. In terms of CPU processing time, CPU processing overhead was higher in ANLJ and BNLJ. This eclipses the I/O processing time gain of ANLJ. On the other hand, this also indicates that if CPU processing time of ANLJ can be enhanced, then ANLJ can outperform other join algorithms. CPU processing time of ANLJ can be enhanced by using multi-threading or double-buffering.

Slight better performance of GHJ than SMJ is due to the sorting CPU overhead of SMJ. Hash partitioning has less CPU processing overhead than sorting.

4.2.3 *Comparing Join Algorithms with Varying Devices*

We compared join algorithms with varying devices.

We used two different SSDs Micron P300 and Intel 730. Micron P300 is made from 25nm SLC NAND flash memory and Intel 730 SSD was made from Intel's MLC NAND flash memory type. Two SSDs share the same host I/F SATAIII. The maximum bandwidth of SATAIII is 6 Gb/s.

All join algorithms better performed in Intel 730 SSD. We suppose this difference is not due to the performance of

NAND Flash type but due to the enhanced FTL firmware of Intel 730 SSD. Two SSDs have 3-year gap in manufacturing (2014 for Intel 730 SSD and 2011 for Miron P300), thus resulting differences in sequential bandwidth and random IOPS as specified in Table 6.

On the HDD, all join algorithms showed similar patterns. Join processing was 1.7 to 2.2 times slower than join processing on the Intel 730 SSD, I/O processing was 1.9 times to 2.6 times slower than I/O processing on the Intel 730 SSD due to its lower bandwidth (150MB/s for read and write) than SSDs. If recent SSDs with NVMe I/F is adopted then this performance gap can be substantial, since NVMe SSDs have much higher bandwidth (1∼3 GB/s) than SATAIII SSDs (∼0.5 GB/s). In addition, the I/O patterns of join processing corresponded to I/O pattern analysis of the previous study [30].

5 Related Work

In the early days of DBMS research, ad-hoc join algorithms such as SMJ, HHJ, GHJ, BNLJ have been proposed. Among these, HMJ and SMJ have been empirically proven to be the best algorithms, depending on workloads [19]; however, these algorithms were designed to be optimized for HDDs. For SSDs, ANLJ eliminates writes to SSDs, thus prolonging SSD-device lifetimes while showing better performance than HHJ and SMJ in moderate DRAM capacity ranges.

Numerous join algorithms exploiting modern hardware features have been proposed. Also, parallel join algorithms emerged according to the advent of multiprocessor systems [31]. Recent parallel join processing algorithms on multicore processors originated from this work. Most recently, in-memory join processing algorithms that exploit modern CPU hardware features such as SIMD, and NUMA, have been proposed. Kim et. al. [32] compared hash join and sort join algorithms using these modern multi-core CPU features. Later more optimized SMJ [33] and hash based join algorithms [34] were proposed. These modern join processing algorithms are orthogonal to our work. ANLJ can be parallelized by adopting parallel join processing techniques, thus significantly decreasing CPU processing overhead if in-memory join processing techniques exploiting SIMD, and NUMA are applied.

Further, a workload-aware join algorithm has been proposed. Diag-join [35] exploits the 1:N cardinality join condition and showed excellent performance (i.e., similar performance to that of ANLJ) when the target join relation is sorted; however, diag-join has a critical limitation in that it cannot be applied to non-sorted relations. As noted above in Section 4.3.3 (TPC-C workload which is the non-sorted relations), ANLJ can be applied to non-sorted relations and, moreover, performs better than SMJ and HHJ as mentioned in Section 4.3.3.

A flash-aware join algorithm has also been proposed. Digest join [36] only extracts join-keys and their corresponding column values and tuple IDs at the first scan of join target relations. Only using these values, it calculates tuple ID pairs for join results. Then, based on the tuple IDs, it reads the other required columns from the relations and produces a completely joined tuples by assembling columns for every join tuple ID pair. Digest join can be regarded as a method that builds a join-key index or a late materialization technique. ANLJ is orthogonal to digest-join in that digest-join can be used as a preprocessing step for other ad-hoc join algorithms. After extracting join keys, and corresponding column values and tuple IDs, diag-join can perform other join algorithms, such as SMJ, HHJ, BNLJ, or ANLJ, to calculate tuple ID pairs for joined tuples.

As related work for flash based DBMSs, other previous studies tried to exploit flash SSDs in different ways. Hybrid approaches using SSDs as a middle layer between DRAM and HDDs were proposed, such as using SSDs as a second buffer cache of DBMSs [37], a row and meta data cache for a key value store [38], and an indirection layer of indexes [39]. The Bw-tree index [40] was designed to be log-structured thus creating flash-friendly sequetnial I/O patterns in order to alleviate the write-amplification problem caused by random writes. Lee et. al also proposed a transactional FTL [41] which can replace journaling operations of SQLite based on the out-of-place update mechanism of FTL.

6 Conclusion

Flash-memory write endurance has been worsening. This trend is unfortunately not expected to change.

We proposed ANL join algorithm (ANLJ) to decrease the join processing time of BNLJ by nearly half with no reduction in flash memory lifetimes. The major join algorithms wear flash memory because of their excessive temporary file writes on SSDs.

We examined the performance of ANLJ by using real DBMS files created from PostgreSQL codebase. ANLJ outperformed BNLJ with various workloads. We also theoretically proved that ANLJ can cut the amount of required I/O of BNLJ in half.

From our results, ANLJ showed performance similar to that of GHJ, HHJ and SMJ when DRAM buffer size was 26 percent of the smaller relation. In addition, when relations were sorted in order of join-key values, ANLJ outperformed the other join algorithms, regardless of DRAM buffer size, indicating that ANLJ can be used when the given DRAM buffer size is greater than 26 percent of the smaller relation. Considering trends of dropping DRAM prices and increasingly large DRAM capacities installed in enterprise servers, ANLJ is applicable in most environments. Furthermore, given ANLJ's advantage of eliminating temporary file writes, even when the noted DRAM condition is not met, in the range from 10 to 25 percent, ANLJ should be examined as a viable option with a tradeoff between losses in performance a reduction of TCO by extending SSD lifetimes.

In our future work, we plan to enhance ANLJ's CPU processing time by using a multi-threaded approach used in parallel join processing algorithms and double buffering. In addition, we plan to exploit modern CPU architecture features (e.g., SIMD and NUMA) just as most recent in-memory join algorithms do.

Acknowledgments

Sanghyun Park is the corresponding author. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (NRF-2015R1A2A1A05001845).

REFERENCES

[1] R. R. Schaller, "Moore's law: past, present and future," *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, Jun. 1997.

[2] A. A. H. Danny Cobb, "NVM express and the PCI express* SSD revolution," in *Intel Developer Forum*. Santa Clara, CA, USA: Intel, 2012.

[3] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of NAND flash memory," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 2–2.

[4] Toshiba, "Flash memory basics for SSD users," (2014). [Online]. Available: https://goo.gl/yzvWVb

[5] J. Gathman, A. McPadden, and G. Tressler, "The Need for Standardization in the Enterprise SSD Product Segment".

[6] S.-W. Lee and B. Moon, "Design of flash-based DBMS: An in-page logging approach," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 55–66.

[7] C.-H. Wu, T.-W. Kuo, and L. P. Chang, "An efficient B-tree layer implementation for flash-memory storage systems," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 3, 2007, Art. no. 19.

[8] G.-J. Na, S.-W. Lee, and B. Moon, "Dynamic in-page logging for flash-aware B-tree index," in *Proc. 18th ACM Conf. Inf. Knowl. Manage.*, 2009, pp. 1485–1488.

[9] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim, "μ-tree: an ordered index structure for NAND flash memory," in *Proc. 7th ACM IEEE Int. Conf. Embedded Softw.*, 2007, pp. 144–153.

[10] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee, "B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives," *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 286–297, 2011.

[11] Facebook, "Under the Hood: Building and open-sourcing RocksDB," (2013). [Online]. Available: http://goo.gl/9xulVB

[12] Google, "Leveldb: A fast and lightweight key/value database library," (2012). [Online]. Available: https://code.google.com/p/leveldb/

[13] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996.

[14] Facebook, "RocksDB feature: Universal compaction," [Online]. Available: https://goo.gl/vxWQFy

[15] D. Kocic, "Case for flash storage - How it can benefit your 1Q0534 enterprise," *Flash Memory Summit*, 2015.

[16] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of hash to data base machine and its architecture," *New Generation Comput.*, vol. 1, no. 1, pp. 63–74, 1983.

[17] 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," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1984, pp. 1–8.

[18] M. W. Blasgen and K. P. Eswaran, "Storage and access in relational data bases," *IBM Syst. J.*, vol. 16, no. 4, pp. 363–377, 1977.

[19] G. Graefe, A. Linville, and L. D. Shapiro, "Sort versus hash revisited," *IEEE Trans. Knowl. Data Eng.*, vol. 6, no. 6, pp. 934–944, Dec. 1994.

[20] S. Huang, "SSD Controller Technologies for TLC NAND," *Flash Memory Summit*, 2013.

[21] W. W. Hooker, "On the expected lengths of sequences generated in sorting by replacement selecting," *ACM Commun.*, vol. 12, no. 7, pp. 411–413, 1969.

[22] B. J. Gassner, "Sorting by replacement selecting," *ACM Commun.*, vol. 10, no. 2, pp. 89–93, 1967.

[23] T. P. P. Council, "TPC benchmark H, standard specification version 2.16."

[24] T. P. P. Council, "TPC benchmark C, standard specification version 5."

[25] Micron, "P300," (2010). [Online]. Available: https://goo.gl/I5VXJa

[26] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 5th Ed. New York, NY, USA: McGraw-Hill Book Company, 2005.

[27] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla, "Seeking the truth about ad hoc join costs," *VLDB J.*, vol. 6, no. 3, pp. 241–256, 1997.

[28] Intel, "Intel 730 SSD," (2014). [Online]. Available: https://goo.gl/wbvoKf

[29] Hitachi, "Hitachi. Deskstart 7K1000.D," (2011). [Online]. Available: https://goo.gl/sqXPqY

[30] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory ssd in enterprise database applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1075–1086.

[31] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," in *Proc. 18th Int. Conf. Very Large Data Bases*, 1992, pp. 27–40.

[32] C. Kim, et al., "Sort versus Hash revisited: fast join implementation on modern multi-core CPUs," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1378–1389, 2009.

[33] M.-C. Albutiu, A. Kemper, and T. Neumann, "Massively parallel sort-merge joins in main memory multi-core database systems," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 1064–1075, 2012.

[34] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core CPUs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 37–48.

[35] S. Helmer, T. Westmann, and G. Moerkotte, "Diag-Join: An opportunistic join algorithm for 1:N relationships," in *Proc. 24rd Int. Conf. Very Large Data Bases*, 1998, pp. 98–109.

[36] L. Yu, O. Sai Tung, X. Jianliang, B. Choi, and H. Haibo, "Optimizing nonindexed join processing in flash storage-based systems," *IEEE Trans. Comput.*, vol. 62, no. 7, pp. 1417–1431, Jul. 2013.

[37] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson, "Turbocharging DBMS buffer pool using SSDs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 1113–1124.

[38] P. Menon, T. Rabl, M. Sadoghi, and H. A. Jacobsen, "CaSSanDra: An SSD boosted key-value store," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 1162–1167.

[39] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee, "Making updates disk-I/O friendly using SSDs," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 997–1008, 2013.

[40] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "*The Bw-Tree: A B-tree for new hardware platforms*," in *Proc. IEEE Int. Conf. Data Eng.*, 2013, pp. 302–313.

[41] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, "X-FTL: Transactional FTL for SQLite databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 97–108.

Hongchan Roh is a PhD student and software engineer at SKT. His research interests include architecture-conscious data management with modern storage devices. He has published peer-reviewed papers in major database conferences and journals.

Mincheol Shin is working toward the PhD degree at Yonsei University. His research interests include architecture-conscious data management with modern storage devices.

Wonmook Jung is working toward the MS degree at Yonsei University. His research interests include information management systems and distributed data processing systems on storage class memory.

Sanghyun Park received the PhD degree from UCLA in 2001. He is a professor in the Computer Science Department, Yonsei University. His research interests include database systems, and bioinformatics. He has published more than 80 peer-reviewed papers in major database conferences and journals.