Optimizing Hash Partitioning for Solid State Drives

Mincheol Shin
Dept. of Computer Science
Yonsei University
50 Yonsei-ro, Seodaemun-gu,
Seoul, Korea
smanioso@yonsei.ac.kr

Hongchan Roh
Dept. of Computer Science
Yonsei University
50 Yonsei-ro, Seodaemun-gu,
Seoul, Korea
fallsmal@cs.yonsei.ac.kr

Wonmook Jung
LG Electronics
19, Yangjae-daero 11-gil,
Seocho-gu, Seoul, Repuiblic
of Korea
wonmook.jung@lge.com

Sanghyun Park*
Dept. of Computer Science
Yonsei University
50 Yonsei-ro, Seodaemun-gu,
Seoul, Korea
sanghyun@cs.yonsei.ac.kr

ABSTRACT

The use of flashSSDs has increased rapidly in a wide range of areas due to their superior energy efficiency, shorter access time, and higher bandwidth when compared to HDDs. The internal parallelism created by multiple flash memory packages embedded in a flashSSDs, is one of the unique features of flashSSDs. Many new DBMS technologies have been developed for flashSSDs, but query processing for flashSSDs have drawn less attention than other DBMS technologies. Hash partitioning is popularly used in query processing algorithms to materialize their intermediate results in an efficient manner. In this paper, we propose a novel hash partitioning algorithm that exploits the internal parallelism of flashSSDs. The devised hash partitioning method outperforms the traditional hash partitioning technique regardless of the amount of available main memory independently from the buffer management strategies (blocked I/O vs page sized I/O). We implemented our method based on the source code of the PostgreSQL storage manager. PostgreSQL relation files created by the TPC-H workload were employed in the experiments. Our method was found to be up to 3.55 times faster than the traditional method with blocked I/O, and 2.36 times faster than the traditional method with page-sized I/O.

CCS Concepts

•**Information systems** → **Query operators;**

*Corresponding author. Tel.: +82 2 2123 5714; fax: +82 2 365 2579; E-mail address: sanghyun@cs.yonsei.ac.kr.

SAC'16, April 4-8, 2016, Pisa, Italy

DOI: http://dx.doi.org/10.1145/2851613.2851663

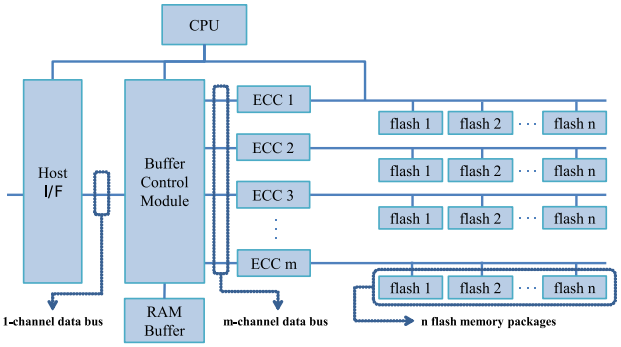


Figure 1: FlashSSD architecture [18]

Keywords

Query Execution; Hash Join; Hash Partitioning; Flash Storage Devices; Internal Parallelism of flashSSDs

1. INTRODUCTION

In the last decade, flashSSDs have been rapidly adopted in a wide range of areas because of their superior features when compared to HDDs. Personal users have employed flashSSDs in their laptop and desktop computers in order to boost operating system and speed up access to frequently used applications. Data centers have also adopted flashSSDs to improve energy efficiency and performance. Many DBMS technologies have been developed for flashSSDs. While majority of previous studies have focused on the buffer and index management of DBMSs on flashSSDs, query processing algorithms for flashSSDs have drawn less attention from researchers.

1.1 The Internal Parallelism of FlashSSDs

A typical flashSSD architecture is shown in Figure 1. A flashSSD includes a CPU, a host interface (host I/F), a RAM buffer, Error-Correcting Code(ECC) modules, multiple data transfer channels and multiple flash memory pack-

ages (chips). Multiple flash memory packages in a flashSSD are assembled into groups each of which is connected to a channel.

The multiple flash memory packages embedded in a flashSSD create internal parallelism. [18] emphasized the importance of exploiting internal parallelism. Using the internal parallelism of flashSSDs, DBMS applications can utilize higher bandwidths of flashSSDs. As mentioned in [3, 9, 18], requesting a number of I/Os (outstanding I/Os) at the same time regardless of the access pattern (random or sequential) is an efficient way to exploit the internal parallelism of flashSSDs. Our benchmark results in Section 2.4 support this notion. Large granularity and a high number of outstanding I/Os guarantee that DBMSs utilize nearly the maximum bandwidth of flashSSDs. The bandwidth gap between the case when internal parallelism was utilized at maximum and the case when it was not is significant. In our benchmark results, exploiting internal parallelism increased the flashSSD bandwidth by a factor of up to 10.

Psync I/O (Parallel Synchronous I/O): A new I/O request interface, Psync I/O, was proposed for exploiting the internal parallelism of flashSSDs [18]. Psync I/O allows applications to request outstanding random I/Os all at once. Psync I/O was implemented using libaio API in Linux.

1.2 Hash Partitioning for FlashSSDs

In this paper, we focus on optimizing the hash partitioning algorithm for flashSSDs. Hash partitioning is a popular operation in query processing. Many query processing algorithms, including hash join algorithms and aggregation operations, use hash partitioning to efficiently materialize intermediate results. Moreover, distributed processing systems like Hadoop also uses a hash partitioning algorithm for the same reason. In MapReduce [5], a mapper partitions their intermediate results and a reducer brings these partitions for reduce function. Tajo [4] which is a distributed data warehouse also uses a hash partitioning algorithm.

The process of the conventional hash partitioning algorithm [2] may be described as follows. First, the allocated main memory space is divided into at least one input buffer and s output buffers where s is the number of partitions. Next, the input relation is read so as to be as large as the input buffer. For each tuple in the input buffer, the hash value of the tuples key is calculated. The tuple is moved to the output buffer for the partition corresponding to the hash value. Before moving the tuple, the method verifies whether or not the output buffer has enough space to store the tuple. If sufficient space is not available, the output buffer is written to the secondary storage and the tuple is subsequently moved to the output buffer. These processes are iterated until all tuples in the input relation are deployed to the corresponding partitions.

[12] proposed *ParaHash* which is a hash partitioning algorithm exploiting the internal parallelism of FlashSSDs. However, *ParaHash* uses many threads for each hash join operation (20 threads in the paper). If several queries are executed simultaneously, the high costs of context switching is inevitable. For this reason, *ParaHash* cannot be an option for commercial database systems.

1.3 Problems and Proposed Method

Previous studies regarding query processing have often used the convenient assumptions that a single query processing operation can utilize a large main memory space and that the buffer manager can fully support the blocked I/O method [6]. In practice, this seems somewhat idealistic. It is hard for DBMSs to allocate a large amount of main memory for every single query processing operation in a real environment. The memory space for query processing is allocated per session, but most DBMSs use the buffer pool as a global resource. In the case of running complex queries, each query processing operation in the complex query competes with other operations for more query processing memory. In addition, DBMSs can be connected by many users, which reduces memory space for each query processing operation or shrinks global buffer pool. A large amount of input data for hash partitioning is also an issue that must be addressed. We will cover this topic in more detail using examples of popular DBMSs in Section 2.2. Furthermore, some DBMSs do not fully support blocked I/O. For a detailed explanation of blocked I/O and page-sized I/O, the reader is referred to Section 2.1.

In order to develop a more practical hash partitioning algorithm for flashSSDs, we considered the performance of a traditional hash partitioning algorithm when both a large and small amount of memory situation is available. When a very large amount of main memory is available and blocked I/O method can be employed, the internal parallelism of flashSSDs can be utilized even with the original hash partitioning method. The output buffer of the hash partitioning method can be large enough to request I/Os with large I/O sizes which can guarantee the maximum bandwidth of flashSSDs. Based on our benchmark results in Section 2.4, if the I/O size is greater than 1MB, the outstanding I/O level (i.e. the number of I/Os requested at once) does not matter. In other words, even with a single I/O request and a 1MB I/O size, DBMSs can utilize the maximum bandwidth of flashSSDs.

However, we cannot secure a large main memory space for query processing for a DBMS session and thus, the original hash partitioning algorithm is unable to exploit internal parallelism of flashSSDs. Even if we have a large amount of main memory available for query processing, the original algorithm cannot exploit internal parallelism unless the DBMS fully supports the blocked I/O method.

In this paper, we propose an efficient hash partitioning algorithm, called *n-way hash partitioning (N-Hash)*. Regardless of the main memory size or the support for blocked I/O, N-Hash can exploit internal parallelism by generating a high number of outstanding I/Os. Consequently, it outperforms the traditional hash partitioning method. N-Hash replaces a traditional output buffer structure with a circular-queue-like output buffer, denoted simply as a circular buffer. Each circular buffer structure is divided into 8KB sub-structures called sub-buffers. Based on the circular buffer, N-Hash generates a number of I/Os by flushing as many sub-buffers as possible. The created outstanding I/Os are delivered to flashSSDs via the Psync I/O interface [18].

We implemented N-Hash based on the source code of the

PostgreSQL storage manager. The PostgreSQL relation files were created by the TPC-H benchmark [20], which is a well known online analytical processing(OLAP) workload, were used in the experiments. From the results, N-Hash was found to be up to 3 times faster than the traditional hash partitioning algorithm with blocked I/O. Based on the page sized I/O, N-Hash was up to 2 times faster than the traditional hash partitioning algorithm.

The contributions of this paper are summarized as follows:

- N-Hash is the most efficient hash partitioning algorithm designed for flashSSDs. Original hash partitioning algorithm cannot exploit the internal paralleism of flashSSDs. *ParaHash* [12] can utilize the high I/O bandwidth of flash SSDs, but *ParaHash* consumes a lot of CPU resources because of many threads for the hash partitioning. *ParaHash* has trade-off between performance of hash partitioning algorithm and overall system. N-Hash overcomes these problems of traditional hash partitioning algorithms.
- N-Hash outperforms the traditional hash partitioning algorithm, regardless of the memory size or the support for blocked I/O. Since N-Hash shows outstanding performance even when the available memory size is very small, it can help the DBMS memory-tuner to manage main memory in a flexible manner. Consequently, DBMSs can allocate a larger amount of memory to more important modules (e.g. the buffer pool). In the same vein, it guarantees the performance of DBMSs in more stressful situations where the amount of input data is large and there are many user connections or complex queries.

1.4 Paper Organization

Background information and related work are presented in Section 2, and our proposed hash partitioning algorithm, *N-Hash* is introduced in Section 3. In Section 4, empirical results obtained with *N-Hash* are discussed. Conclusions are presented in Section 5.

2. BACKGROUND AND RELATED WORK

2.1 Blocked I/O vs Page-sized I/O

DBMSs basically manage the database in pages whose size is typically 8kb. Such pages are read and written between the secondary storage and main memory through the buffer manager. The read and write mechanisms for the pages are collectively called the page-sized IO. Storing data into pages makes it easier for DBMSs to manage data. However, use of the page-sized I/O mechanism lead to many I/Os with very small granularity. Therefore, in query processing, the blocked I/O technique has been preferred. In the blocked I/O method, read and write operations are performed in blocks whose the size is greater than the page size.

2.2 Memory Management in DBMSs

In this section, we introduce memory management for hash partitioning in three representative DBMSs: PostgreSQL, DB2 and Oracle. To investigate the memory management mechanism for hash partitioning in PostgreSQL, we surveyed the PostgreSQL 9.1.4 documentation [19] and analyzed the associated source code. For DB2 and Oracle, we referred to their respective manuals written for system administrators [10, 11, 17].

In PostgreSQL, the total memory for output buffers is only 8MB and output buffer is performed in the mannar of page-sized I/O. PostgreSQL allocates 1MB for an in-memory hash table and 8MB to write data exceeding the in-memory hash table. The hash join algorithm in PostgreSQL uses an in-memory hash table in a special memory region called working memory and page-sized output buffers (for partitions) are placed in the other memory region. PostgreSQL flushes the output buffers using the page-sized I/O method.

Both DB2 and Oracle can dynamically redistribute available main memory space between several main memory regions including the buffer pool, regions for hashing and sorting and so on. The operation to redistribute the memory space, itself, is expensive. Since depriving memory space cannot be performed instantly and should be performed very carefully, a very elaborate process is needed.

2.3 Related Work

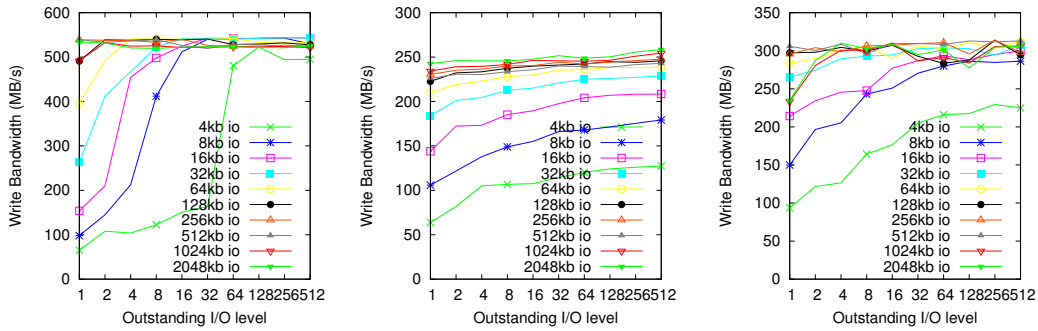
Many flash memory oriented indexes have been proposed. Early studies [15, 22] focused on reducing the number of page writes caused by index operations. FD-tree [13] was the first flashSSD-oriented index that exploited the outstanding sequential bandwidths of flashSSDs, while PIO B-tree [18] was the flashSSD-oriented index that exploited the internal parallelism of flashSSDs.

Query processing algorithms for flashSSDs have drawn less attention than the other DBMS technologies. The performance of major ad-hoc join algorithms for flashSSDs was assessed using the buffer allocation method proposed by [8] in the study [6]. Based on the PAX layout [1] and late materialization, FlashJoin [21] demonstrated outstanding performance when the projectivity and selectivity of queries were low. FlashJoin [21] is a general pipelined join algorithm that reduces the amount of I/O transfers based on the PAX layout [1] and late materialization strategy.

[12] is a research for a hash join algorithm to exploit internal parallelism of flashSSDs. *ParaHashJoin* divides a relation into k regions. k threads read each corresponding region (*ParaScan*). Each thread also perform hash partitioning for read region(*ParaHash*). All threads share the in-memory hash table for hash partitioning. However, to achieve maximum performance of *ParaHashJoin*, too many of threads are needed. In [12], *ParaHashJoin* uses 20 threads.

2.4 Psync I/O Write Benchmark Results

In this section, benchmarks results are presented in order to investigate the effects of internal parallelism on flashSSD bandwidths. Because exsisting benchmark tools does not support Psync I/O, we developed our own benchmark tool called AIOMicrobench that generates outstanding I/Os. Following work presented in [18] Libaio API was used to implement Psync I/O . To make the I/Os bypass the file system cache, unbuffered I/O (direct I/O in Linux) was adopted. In a hexacore 3.2GHz Linux machine with 16GB of RAM, we



(a) Fusion-io IODrive (b) Micron p300 (c) OCZ vertex3 max IOPS

Figure 2: Benchmark results of the write operation with Psync I/O on three flashSSDs

used three flashSSD devices for benchmarks, Fusion-io IODrive [7], micron p300 [14] and OCZ vertex 3 max IOPS [16]. We carefully chose the flashSSDs for the benchmarks to in order to evaluate various flashSSD architectures. IODrive is a high-priced enterprise-level flashSSD with PCI-e host I/F (interface) and 50nm SLC NAND flash memory, while p300 is an enterprise-level flashSSD with SATA3 host I/F and 35nm SLC NAND flash memory. vertex3 max IOPS is a consumer-level flashSSD with SATA3 host I/F and 32nm MLC NAND flash memory. The benchmarks were used to measure the write bandwidth as the I/O size and outstanding I/O level were varied. In each benchmark, we created an 8GB file. The I/O size was increased by multiples of 2, from 4 KB to 2048KB. For each I/O size, we varied the outstanding I/O level from 1 to 512. The total amount of data written by the I/O operations was 2GB.

The write bandwidth results obtained with IODrive, p300 and vertex3 max IOPS are shown in Figure 2. In general, the bandwidths of the flashSSDs increased when either the I/O size or the outstanding I/O level was increased. As shown in Figure 2(a), regardless of I/O sizes, the bandwidth of IODrive reached the maximum bandwidth of flashSSDs. In contrast, the bandwidth of p300 and vertex3 Max IOPS did not reach the maximum bandwidth when the I/O sizes were small. In the case of p300, the bandwidth reached the max bandwidth when the I/O size was greater than or equal to 64 KB. The bandwidth of vertex3 max IOPS did not reach the maximum bandwidth when the I/O size was 4KB.

3. N-HASH

In this section, we introduce our novel hash partitioning algorithm, *n-way hash partitioning (N-Hash)*. In the traditional hash partitioning algorithm, a DBMS writes just one output buffer filled with tuples when the output buffer becomes full as described in Section 1.2. However, in *N-Hash*, the DBMS writes both the output buffer filled with tuples and portions of other output buffers filled with tuples. Consequently, *N-Hash* can create outstanding I/Os, and can better utilize the internal parallelism of flashSSDs. In *N-Hash*, each output buffer is divided into several small blocks each of which is the basic I/O unit for writes. The overall process to fill and flush our own output buffer is analogous to that of a circular queue.

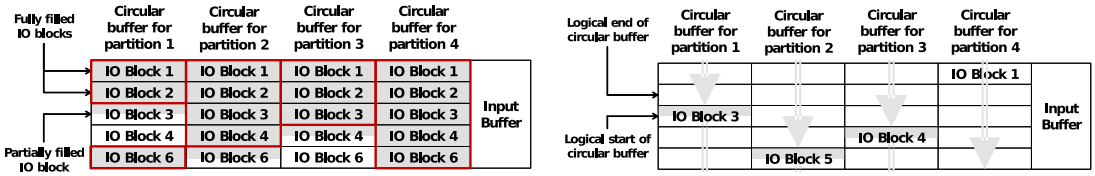
Algorithm 1: NHash(R,n,sizeIB,sizeOB,keyList)

Input:
R : relation which will be partitioned
n : relation into which the t will be inserted
sizeIB : size of input buffer
sizeOB : size of circular output buffer
keyList : list of partitioning key
Output: none

```
1  inBuf ← alloc(sizeIB);
2  outBufs ← alloc(n * sizeOB);
3  relOffset ← 0;
4  while (sizeRead ← readData(inBuf, R, sizeIB, relOffset))
   != 0 do
5      relOffset += sizeRead;
6      foreach tuple t In inBuf do
7          hv ← hashFunc(t, keyList);
8          if emptySpace(outBufs[hv]) < sizeof(t) then
9              batches ← mkBatches(outBufs, hv, n);
10             flushBatches(batches);
11             batches ← NULL
12         end
13         insert(outBufs[hv], t);
14     end
15     writeAllFilledSubBufferes(outBufs,n);
16 end
```

As shown in Figure 3, *N-Hash* uses the input buffer for the input relation and an alternative data structure instead of the original output buffer, namely circular output buffer (circular buffer in short). Each *circular buffer* is subdivided into uniform pieces, called *sub-buffer*. Each sub-buffer can have the following three states: fully-filled, partially-filled and empty. Fully-filled means that the sub-buffer is full and there is no more space in which to insert tuples. Partially-filled indicates that there is available space where tuples can be inserted. In an empty sub-buffer, there is no inserted tuple. The gray portions in each sub-buffer of Figure 3(a) represent regions filled with tuples; fully-filled sub-buffers are entirely marked in gray color, and partially-filled sub-buffers have some gray portions. Empty sub-buffers have no gray shade.

The process of *N-Hash* is described as follows: First, *N-Hash*



(a) Before the tuples in circular buffers are written into storage.

(b) After fully-filled sub-buffers are written to the disk.

Figure 3: Illustration of the internal structure of memory for N-Hash in two situations.

reads the input relation in the input buffer as large as the input buffer. Each tuple in the input buffer is moved to a corresponding circular buffer on the basis of the hash value if the circular buffer has an empty space to accommodate the tuple (Lines 4 to 14 of Algorithm 1). When one of the circular buffers becomes full, all the fully-filled sub-buffers (Lines 8 to 12) regardless of the circular buffer which to the sub-buffers belong. The write operation is performed as batches of contiguous fully-filled sub-buffers in all the circular buffers. In Figure 3(a), a rectangle surrounded with red bold lines represents a batch to be written as a chunk, and there are five batches to be flushed. After the write operation, only one partially filled sub-buffer remains in each circular buffer while other sub-buffers become empty as illustrated Figure 3(b). The partially filled sub-buffer in each circular buffer becomes the beginning sub-buffer of the circular buffer. The next tuple to be inserted into the circular buffer is appended next to the last tuple in the partially filled sub-buffer. This process is iterated until the input relation is completely read. At the end of the iteration, non-empty sub-buffers are flushed to the flashSSDs (Line 15).

To perform Psync I/O, *N-Hash* maintains a list of batches to write. After all the batches are appended to the list, I/Os for the batches are requested all at once through Psync I/O, where the outstanding I/O level (the number of I/Os requested at once) is equal to the number of batches in the list. However, in a circular buffer, the first tuple may not be placed in the first sub-buffer and thus, one or two batches to be written can exist in the buffer. For example, the first tuple of the circular buffer for partition 1 in Figure 3(b) is placed on sub-buffer 3. When the circular buffer for partition 1 becomes full, the order for writing sub-buffers will be the 3rd, 4th, 5th, 1st and 2nd sub-buffer. Sub-buffer 3, 4 and 5 compose one batch while the other batch is comprised of sub-buffer 1 and 2.

There is an issue with how to determine the size of a sub-buffer. As described in Section 2.4, if the sub-buffer is too small, the bandwidth of the flashSSDs cannot reach a maximum value. We can set the default size of a sub-buffer to the minimum I/O size which can reach the maximum bandwidth. However, If the available size of memory is too low, the size of each circular buffer should be less than or equal to the minimum I/O size which can reach the maximum bandwidth. If we use one sub-buffer for each circular buffer, outstanding I/O level is always one. When the outstanding I/O level is low, it is hard to exploiting the internal parallelism of the flashSSDs (see Section 2.4). Therefore, in this case, we divide the circular buffer into multiple sub-buffers regardless of I/O size.

To incorporate our idea with page-sized I/O scheme, the algorithm must be adjusted slightly. Instead of making a batch of contiguous sub-buffers, each sub-buffer comprises a batch. Consequently, Psync I/O is performed with the outstanding I/O level equal to the number of fully-filled sub-buffers.

4. EXPERIMENT

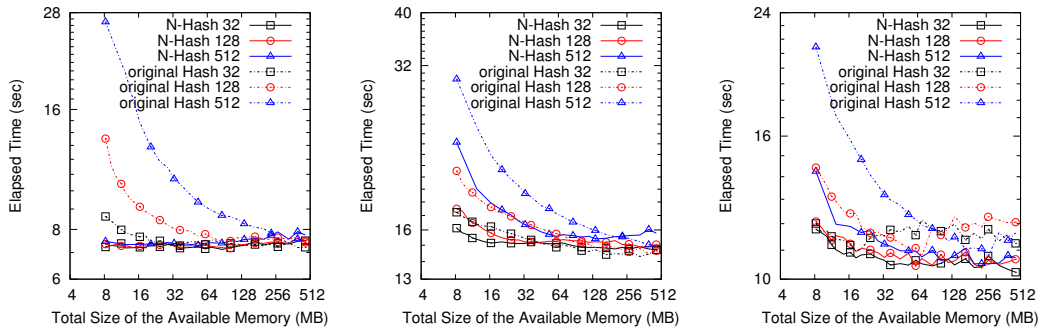
In this section, we compare the performance of *N-Hash* with that of the traditional hash partitioning method. However, there are differences between the configurations that the query processing algorithms use for their own hash partitioning. For example, the buffer allocation method for the hash partitioning of Grace Hash Join is different from that for the hash partitioning of Hybrid Hash Join. To reflect these differences, we compared the performance of *N-Hash* with that of the traditional hash partitioning algorithm in two different environments. In the first experimental set, we measured the performance of each algorithm as the output buffer size and number of partitions was varied in order to evaluate the algorithms with various buffer allocation settings. In the second set of experiments, we investigate the performance of hash partitioning associated with join processing. we used the optimized buffer allocation method for Grace hash join suggested by [8]. Lastly, we measured the performance of hash partitioning based on the page-sized I/O (the previous two experimental set used blocked I/O). In the three experimental set, the performance of the two algorithms was measured as the amount of available main memory was varied.

4.1 Experimental Setting

Based on the source code of the PostgreSQL storage manager, we implemented a single-threaded program to read and partition a PostgreSQL relation file. Non-buffed I/O was used (direct I/O), where I/Os pass through the file system cache. Psync I/O was implemented using libaio API. We used the same Linux machine and flashSSDs described in Section 2.4. We also used the ORDERS table in the TPC-H workload [20], which was created with scale factor of 10 and contained 15 million tuples with the 2.04 GB file. The ORDERS table was partitioned using its foreign key, O_CUSTKEY.

4.2 Experiments for Fixed Number Partitions

Here, the amount of the available memory was varied from



(a) Fusion-io IODrive (b) Micron p300 (c) OCZ vertex max IOPS

Figure 4: Elapsed time in log scale, varying the available memory size

8MB to 452MB. We also varied the number of partitions, as 2, 8, 32, 128 and 512 in the experiments. To evaluate the write performance, the input buffer size was fixed at 4MB. The 4MB-sized input buffer size is sufficient to exploit the internal parallelism of the flashSSDs. The remainder of the main memory was allocated to the output buffers. Figure 4 shows the elapsed time of *N-Hash* and the traditional hash partitioning in log scale. In the figure, the results obtained with 2 and 8 partitions are omitted since similar performance was observed with 32 partitions. Regardless of the algorithm, the elapsed time decreased as the amount of available memory increased or the number of partitions decreased. This is because either increasing the available memory size or reducing the number of partitions made each output buffer larger. Enlarged output buffer led to better utilization of the internal parallelism of the flashSSDs. However, since our method created outstanding I/Os and thus, further exploited the internal parallelism, *N-Hash* outperformed traditional hash partitioning. *N-Hash* was up to 3.55, 1.31 and 1.50 times faster than the traditional hash partitioning algorithm on the IODrive, p300 and vertex3 max IOPS, respectively. It is worth noting that on the IODrive, the performance of *N-Hash* was nearly independent of the available main memory size, whereas the performance of the traditional hash partitioning algorithm was severely degraded when the available main memory size was small.

4.3 Experiments for Hash Partitioning in Grace Hash Join

In this experimental set, experiments were conducted using the buffer allocation method for Grace Hash Join presented in [8]. The size of each memory component is defined as follows:

$$B = \left\lceil \frac{|R|F + \sqrt{|R|^2F^2 + 4M|R|F}}{2M} \right\rceil \quad (1)$$

$$O = \left\lfloor \frac{M}{B+1} \right\rfloor \quad (2)$$

$$I = M - B \times O \quad (3)$$

where B is the number of partitions for Grace hash join, while O and I are the size of the output and input buffers, respectively, in partitioning phase.

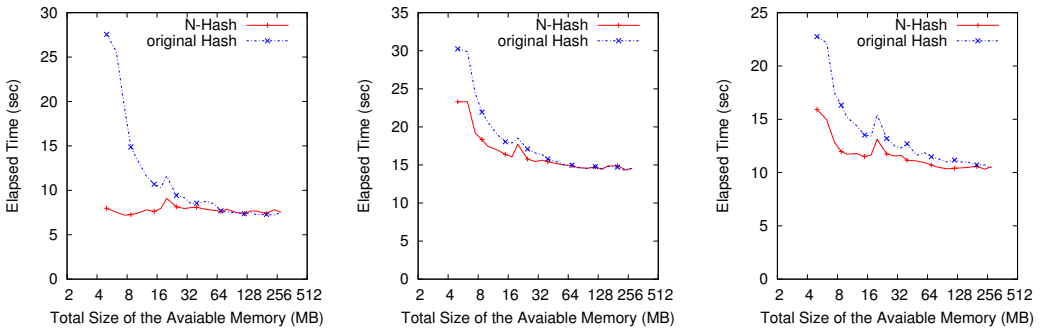
Figure 5 shows the elapsed time of *N-Hash* and optimized hash partitioning algorithm proposed by [8]; samples of the results and parameters are given in Table 1. Here, the available memory size was varied from 4MB to 256MB. *N-Hash* outperformed the traditional hash partitioning algorithm on all the flashSSDs. The speed was increased by amaximun of 3.45, 1.30 and 1.43 times on the IODrive, p300 and vertex3 max IOPS, respectively. In Figure 5, the performance of IODrive is independent of the size of allocated memory for the hash partitioning operation, while the performance of p300 and vertex3 max IOPS is not. When the DBMS allocates less available memory for the hash partitioning operation, the size of each output buffer is also small. For example, the size of each output buffer is 8, 32, 152 KBs on 5, 10, 20 MBs. On p300 and vertex3 Max IOPS, these output buffer sizes are not sufficient to perform write operations with the maximum bandwidth as observed in Figure 2.

It may seem odd that the elapsed time was increased around 20MB on all flashSSDs. As we can see in table 1, the input buffer size was configured at 208KBs following the buffer allocation method proposed in [8]. Because 208KB-sized read operation cannot utilize the maximum bandwidth of flashSSDs, the read operation took a longer amount of time when compared with other memory conditions.

In terms of the memory requirements to yield the same performance with both algorithms, *N-Hash* requires a lower amount of memory for the same level of performance when compared to the traditional hash partitioning algorithm. On the IODrive, the performance of *N-Hash* is always maximized regardless of the amount of available memory. On vertex3 max IOPS, *N-Hash* is performed with maximum performance if the size of available memory is larger than 10MB. On the other hands, the performance of traditional hash partitioning algorithm is the same as of that of *N-Hash* with 200MB of available memory. This means that *N-Hash* makes the DBMS perform 20 times more concurrent users or about 20 times bigger input data with maximum performance when compared with traditional hash partitioning algorithm.

4.4 Experiment for Page-sized I/O

We compared *N-Hash* with the traditional hash partitioning algorithm, based on page-sized I/O as the amount of

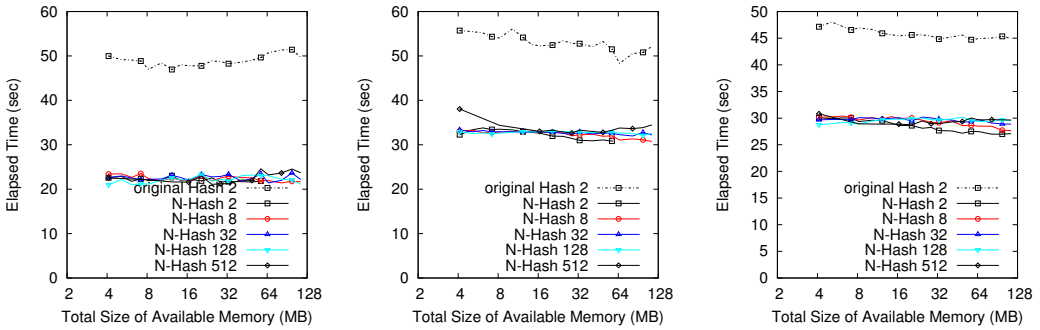


(a) Fusion-io IODrive (b) Micron p300 (c) OCZ vertex max IOPS

Figure 5: Elapsed time, using the buffer allocation method for Grace hash join

Table 1: Detailed results from Figure 5

Device	Parameters				N-Hash			Original Hash			speedup
	Mem (MB)	B	O (KB)	I (KB)	total (s)	read (s)	write (s)	total (s)	read (s)	write (s)	
iodrive	5	517	8	856	7.98	3.05	3.51	27.56	3.16	21.85	3.45X
	10	259	32	1696	7.4	2.95	3.21	13.58	3.05	8.38	1.84X
	20	130	152	208	9.09	3.72	3.21	11.61	4.94	4.56	1.28X
	30	87	336	720	7.96	3.3	3.24	9.17	3.65	3.91	1.15X
	50	53	920	1160	7.85	3.3	3.26	8.75	3.55	3.89	1.12X
	100	27	3560	3720	7.51	3.16	3.41	7.48	3.04	3.59	1X
	200	14	13312	13312	7.38	2.8	3.54	7.29	2.82	3.48	0.99X
p300	5	517	8	856	23.3	5.99	15.43	30.24	6.09	21.75	1.3X
	10	259	32	1696	17.46	5.87	9.8	20.65	5.84	12.32	1.18X
	20	130	152	208	17.7	6.8	8.43	18.54	7.22	8.97	1.05X
	30	87	336	720	15.44	5.89	7.97	16.57	6.24	8.46	1.07X
	50	53	920	1160	15.16	5.82	8.03	15.39	5.96	7.97	1.01X
	100	27	3560	3720	14.57	5.8	7.66	14.56	5.83	7.62	1X
	200	14	13312	13312	14.89	5.72	8.04	14.71	5.68	7.95	0.99X
vertex3 max IOPS	5	517	8	856	15.93	5.59	8.25	22.76	5.52	15.05	1.43X
	10	259	32	1696	11.72	5.02	4.98	15.18	5.29	7.56	1.29X
	20	130	152	208	13.11	6.18	4.62	15.43	8.3	4.89	1.18X
	30	87	336	720	11.56	5.39	4.67	12.49	6.15	4.57	1.08X
	50	53	920	1160	11.08	4.98	4.87	11.62	5.21	5.05	1.05X
	100	27	3560	3720	10.35	4.53	4.81	10.99	4.59	5.38	1.06X
	200	14	13312	13312	10.59	4.35	5.06	10.71	4.25	5.42	1.01X



(a) Fusion-io IODrive (b) Micron p300 (c) OCZ vertex max IOPS

Figure 6: Elapsed time, using page-sized I/O

the available memory was varied from 4MB to 112MB. The number of partitions was also varied, as 2, 8, 32, 128 and 512. The results obtained in this experiment are shown in Figure 6. Allocating more memory made no significant differences for the traditional hash partitioning algorithm. This is because the output buffer size does not change in page-sized I/O. However, since our method created outstanding I/Os, *N-Hash* outperformed traditional hash partitioning. Specifically *N-Hash* was up to 2.36, 1.68 and 1.59 times faster than the traditional hash partitioning on the IODrive, p300 and vertex3 max IOPS, respectively.

5. CONCLUSION AND DISCUSSION

In this paper, we proposed a novel hash partitioning algorithm for flashSSDs, called *n-way hash partitioning (N-Hash)* that exploits the internal parallelism of flashSSDs. Regardless of the main memory size or support for blocked

I/O, *N-Hash* can exploit the internal parallelism of flashSSDs by outstanding I/Os. Consequently, *N-Hash* outperforms the traditional hash partitioning method. From the experimental results, *N-Hash* was found to be up to 3.55 times faster than the traditional hash partitioning algorithm, with blocked I/O. With page sized I/O, our algorithm was up to 2.36 times faster than the traditional hash partitioning algorithm. In addtition, *N-Hash* can reduce the cost of dynamic memory tuning for DBMSs and make DBMSs more endurable in stressful situations where the amount of input data is large and there are many user connections or complex queries, because *N-Hash* exhibits excellent performance with a small amount of memory and occupies CPU resources for just one thread. However, several types of flashSSDs like NVMe are developed. Because NVMe supports more I/O queue and channel than traditional flashSSDs, Evaluating N-Hash on NVMe is necessary.

6. ACKNOWLEGMENTS

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

7. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, 2002.

[2] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 323–333. Morgan Kaufmann Publishers Inc., 1984.

[3] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 266–277. IEEE, 2011.

[4] H. Choi, J. Son, H. Yang, H. Ryu, B. Lim, S. Kim, and Y. D. Chung. Tajo: A distributed data warehouse system on large clusters. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1320–1323. IEEE, 2013.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] J. Do and J. M. Patel. Join processing for flash ssds: remembering past lessons. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 1–8. ACM, 2009.

[7] Fusion-io. iodrive data sheet. http://www.fusionio.com/load/-media-/1ufytn/docsLibrary/FIO_DS_ioDrive.pdf, 2010.

[8] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the truth about ad hoc join costs. *The VLDB Journal*, 6(3):241–256, 1997.

[9] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the international conference on Supercomputing*, pages 96–107. ACM, 2011.

[10] IBM. Ibm db2 10.1 for linux, unix, and windows database administration concepts and configuration reference. http://public.dhe.ibm.com/ps/products/db2/info/vr101/pdf/en_US/DB2AdminConfig-db2dae1010.pdf, 2012.

[11] IBM. Ibm db2 10.1 for linux, unix, and windows troubleshooting and tuning database performance. http://public.dhe.ibm.com/ps/products/db2/info/vr101/pdf/en_US/DB2PerfTuneTroubleshoot-db2d3e1010.pdf", 2012.

[12] W. Lai, Y. Fan, and X. Meng. Scan and join optimization by exploiting internal parallelism of flash-based solid state drives. In *Web-Age Information Management*, pages 381–392. Springer, 2013.

[13] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment*, 3(1-2):1195–1206, 2010.

[14] Micron. p300 data sheet. http://www.micron.com/~/media/Documents/Products/Product\%20Flyer/p300_product_brief.pdf, 2010.

[15] G.-J. Na, S.-W. Lee, and B. Moon. Dynamic in-page logging for flash-aware b-tree index. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1485–1488. ACM, 2009.

[16] OCZ. Vertex3 max iops product sheet. http://www.ocztechnology.com/res/manuals/OCZ_Vertex3_MAX_IOPS_Product_sheet.pdf, 2011.

[17] Oracle. Oracle database concepts 11g release 2 (11.2). http://www.oracle.com/pls/db112/to_pdf?pathname=server.112/e25789.pdf, 2011.

[18] 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. *Proceedings of the VLDB Endowment*, 5(4):286–297, 2011.

[19] The PostgreSQL Global Development Group. Postgresql 9.1.6 documentation. http://www.postgresql.org/files/documentation/pdf/9.1/postgresql-9.1-A4.pdf, 2013.

[20] Transaction Processing Performance Council. Tpc benchmark h: Standard specification, revision 2.14.4. http://www.tpc.org/tpch/spec/tpch2.14.4.pdf, 2012.

[21] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 59–72. ACM, 2009.

[22] C.-H. Wu, T.-W. Kuo, and L. P. Chang. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):19, 2007.