Comprehensive Analytics of Large Data Query Processing on Relational Database with SSDs

Keisuke Suzuki[1], Yuto Hayamizu[1], Daisaku Yokoyama[1], Miyuki Nakano[2], and Masaru Kitsuregawa[1,3]

[1] University of Tokyo
{keisuke,haya,yokoyama,kitsure}@tkl.iis.u-tokyo.ac.jp
[2] Shibaura Institute of technology
miyuki@sic.shibaura-it.ac.jp
[3] National Institute of Informatics

Abstract. Solid-state drives (SSDs) are widely used in large data processing applications due to their higher random access throughput than HDDs and capability of parallel I/O processing. The I/O bottlenecks that HDDs on database systems face can be resolved by using SSDs because of these advantages. However, access latency on cache hierarchy may become a new bottleneck in SSD-based databases. In this study, we quantitatively analyzed the behavior of SSD-based databases by taking hashjoin operation. We found that cache misses in SSD-based databases can be decreased by reducing the hashtable size to fit into the cache. This is because the I/O cost is not increased by the high throughput of the SSDs, even though the hashjoin partition files are fragmented. We also observed that cache misses are not increased by taking a multi-hashjoin query. This is because the total size of multiple hashtables can fit into the cache size in SSD-based databases, which is in contrast to HDD-based databases, where hashtables require almost all of the available memory. Overall, our analytics clarify that the performance of multiple queries in SSD-based databases can be improved by considering data access locality of the hashjoin operation and determining the appropriate hashtable size to fit into the cache.

Keywords: RDBMS, SSD, Hashjoin, OLAP.

1 Introduction

Flash solid-state disks (SSDs) are likely to improve the I/O bottleneck of data intensive applications due to their lower latency and higher throughput than conventional hard disk drives (HDDs). They are widely used in heavy I/O workload environments as their capacity is constantly growing and the price is dropping.

SSDs offer the same block interface as HDDs, so it is easy to integrate SSDs into a storage system that enables users to access both kinds of devices transparently. We expect that the throughput of SSDs is enough to resolve the I/O bottleneck of HDDs and even maintain their bandwidth. However, another performance bottleneck may occur by fully exploiting their I/O performance in

H. Wang and M.A. Sharaf (Eds.): ADC 2014, LNCS 8506, pp. 135–146, 2014.

SSD–integrated systems. Thus, in this paper, we investigate the performance of an SSD–integrated system and show that it is insufficient to simply treat SSDs as a faster disk-they are also a key element offering a new paradigm for data intensive application performance models.

The I/O costs of conventional HDD-based database systems are often larger by an order of magnitude than memory access costs and CPU calculation costs. Therefore, I/O bandwidth limits the total performance of queries. In contrast, SSDs fill the gap between I/O costs and memory access and calculation costs, especially in the case of random I/Os. If the utilization of SSDs results in removing the I/O bottleneck, memory access and calculation costs may become a new bottleneck. That means we have to consider better utilization of computing resources such as cache and memory. We comprehensively analyzed the performance of SSD-based databases by taking a hashjoin operation that is often used in large data query processing. We then obtained the following information.

- The overall performance of a hashjoin is seriously affected by cache miss penalties. These misses can be reduced by setting a small hashtable size to fit the hashtables into the cache, but this cannot be done on HDD-based databases because using a small memory space causes fragmentation of hashtable partitions and decreases the I/O throughput. In contrast, SSDs improve the I/O throughput even though there are many fragmented hashtable partitions since they have no mechanical seek time and achieve a better performance than HDDs.
- By processing a query of multiple hashjoins, such as queries of decision support systems, cache misses are likely to increase more than with a single join query since multiple hashtables may exist at the same time and share a cache. SSD-based databases can avoid such increases by reducing the individual hashtable size enough to fit some hashtables into the cache. Thus, we have to consider data access locality of hashjoins.

The primary contributions of this paper are as follows:

- We confirm that we can shrink the size of hashtable size to obtain a good total performance of query execution in SSD-based databases.
- We confirm that the potential of improving the performance of multiple query execution by setting the appropriate memory size.

The remainder of this paper is organized as follows. Related work is presented in Section 2. Section 3 explains the behavior and expected processing cost of the hashjoin operation that is used in our analysis. Section 4 shows the basic access performance of HDDs and SSDs. In Section 5, we discuss our experimental analysis of the utilization of SSDs on large data query processing. We conclude with some final insights in Section 6.

2 Related Work

Recently, there has been much research in the area of SSD-integrated database systems. These can be roughly divided into three categories of SSD usage: buffer pool extension, indexing, and HDD-SSD mixed hybrid storage management.

Concerning buffer pool extension, Bhattacharjee et al. proposed a temperature-aware caching (TAC) schema [1,2] that monitors and obtains the statistics of the access patterns of data and then decides which data to keep in the cache on the basis of their access frequency. The FaCE system [3], proposed by Kang et al., uses the multiversion FIFO cache replacement algorithm to reduce the random write. The buffer pool extension is one of promising fields of SSD usage. However, as mentioned in [4], the buffer pool extensions are not beneficial for ad hoc large data processing queries which we focus on in this paper.

Hybrid storage management resembles the idea of caching in that it basically places frequently accessed data on SSDs and less accessed data on HDDs. Koltsidas et al. [5] detect workloads for the pages and distribute read-intensive pages on SSDs and write-intensive pages on HDDs, which overcomes the random write weakness of SSDs. The hStorage-DB [6] semantically analyzes the I/O workload of queries from execution plans. This approach enables data placing prior to query execution, and for that reason, cache filling up time and monitoring overheads are not needed. Hybrid HDD-SDD usage schemes are important for SSDs with less capacity and higher price than HDDs. Our intention is first to elaborate upon the query processing for SSD-only databases.

As for indexing, the FD-tree [7] optimizes writing performance by aggregating write requests, while the PIO B-tree [3] exploits the internal parallelism of SSDs. Indices are typically used on a scan whose data selectivity is low, while a sequential scan and hashtable are likely to be used on a high-selectivity scan. Tsirogiannis et al. [8] utilize the column-based table store to exploit the random access performance of SSDs and propose a column store database oriented hashjoin algorithm.

The studies above focus only on the I/O characteristics of SSDs. In this work, we analyze not only I/O behaviors but also the entire performance improvement of database systems and other component bottlenecks.

3 Join Operation with Hashtable

With large data processing tasks such as DSS queries, a hashtable is typically used on several database operations such as aggregation, projection, and join. We use a hashjoin operation to evaluate performance improvement by SSDs since hashjoin is one of the heaviest workload operations within databases. We clarify that the high I/O throughput of SSDs affects the entire performance of hashjoin operation from the aspect of memory access latency.

3.1 Grace Hashjoin [9] and Hybrid Hashjoin [10]

When a hashtable cannot fit into the main memory due to its size, Grace join divides the target data into partitions to fit each partition into memory and stores them on a disk.

The process of Grace hashjoin is divided in two phases: build and probe. For the sake of explanation, assume the join operation of relations R, S, and

hashtables are created on S. First, in the build phase, both target relations are partitioned by the same hash function and partitions are written to disk. Next, in the probe phase, two partitions R_i and S_i ($1 \leq i \leq n$, $n = S/\text{memorysize}$), which have the same hash value, are selected to join. S_i is loaded and its hashtable created on memory and then the tuples of R_i are matched with the tuples of S_i by referring to the hashtable. This operation is repeated for each partition.

Grace hashjoin consumes memory space for only the write buffer of each partition at the build phase. Hybrid hashjoin utilizes the rest of the memory space to hold the hashtable of the first partition S_1. The memory residing partition (S_1) and the counterpart partition R_1, which have the same hash value as S_1, are processed without being stored to disk. This is how hybrid hashjoin can reduce the I/O cost of an operation with S_1 and R_1.

3.2 Processing Cost of Hashjoin

An HDD-based DBMS often uses Grace hashjoin or hybrid hashjoin on large data query processing because a vast amount of I/Os seriously decreases processing throughput. The rest of this paper deals with hybrid hashjoin algorithms.

The I/O pattern of hashjoin depends on the size of working memory (working memory means available memory space for each hashjoin operation.). This is because the partition size and the number of partitions are decided to fit a respective hashtable for partitions of S into working memory. Therefore, when the working memory space is small, many small partitions are created, which results in the generation of many fragmented partition files. Many random I/Os are invoked to access these fragmented partition files. The fragmentation causes serious I/O throughput degradation on HDDs since random I/Os are 100 - 1000 times slower than sequential I/Os. For this reason, much memory space is typically assigned for hashjoin on HDDs to avoid fragmentation. Concerning SSDs, however, the random I/Os are not slower by an order of magnitude, and therefore the fragmentation has less impact on the I/O throughput. This condition enables less memory space to be used.

A hashtable generated at the probe phase is repeatedly accessed by matching a tuple of relation R. This means that data access locality is expected and has to be considered. The size of working memory is also related to the number of cache misses of the probe phase. When a hashtable for partition S_i fits into a cache, cache misses do not occur after loading partitions. The hashtable size is limited by working memory size, so a hashtable can be fit into a cache and utilization becomes high when the working memory size is smaller than the cache size. SSDs help keep the memory size small without much I/O throughput degradation. Thus, SSDs are expected not only to improve the I/O bottleneck but also to help reduce the number of cache misses on a hashjoin.

4 Basic Performance of HDDs and SSDs

The architecture of SSDs is fundamentally different from that of HDDs. Rotating disks and moving heads to address data are the bottleneck of random accesses

Table 1. Experimental platform setup

CPU	Xeon X7560 (L3 Cache: 24 MB) @ 2.27 GHz x 4
DRAM	64 GB
Storage (SSD)	ioDrive Duo x4 (8 Logical units, Software RAID0)
Storage (HDD)	SEAGATE ST3146807FC x12 (Software RAID0)
Kernel	linux-2.6.32-220
File system	ext4

on HDDs, while SSDs are pure electronic devices so they have no seek time. Another important characteristic is that current SSDs are composed of multiple flash chips, which means they are able to process some I/Os simultaneously. We took some I/O micro-benchmark programs and measured the basic I/O behaviors of SSDs and HDDs in actual use. We then analyzed the basic performances and clarified the differences between SSDs and HDDs.

4.1 Experimental Setup

Table 1 shows the platform setup of our experiment. The interface of the SSDs is PCI Express and that of the HDDs is Fibre Channel. I/O scheduler is set to noop for both storages. The SSDs are tied up by software RAID0 with chunk size = 64 kB and use ext4 file system. The HDDs are set up in the same way.

4.2 Throughput of Sequential and Random Access

To confirm that the random access of the SSDs was superior to the HDDs, we ran micro-benchmark programs of sequential and random read I/Os.

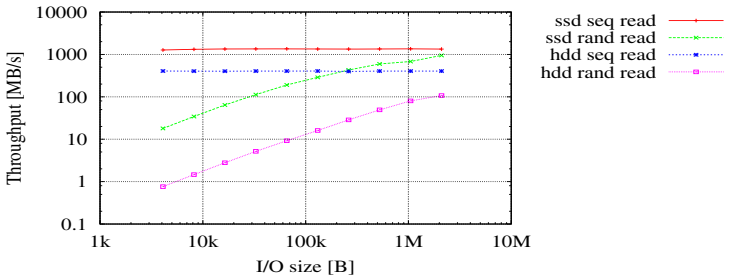


Fig. 1. I/O throughput for sequential and random accesses on SSD and HDD

Figure 1 shows the I/O throughput of SSD and HDD by varying the size of individual I/Os. The throughput of the sequential read (seq read) of SSD is 3.1 times faster than that of HDD and the throughput of the random read (rand read) is 8.6 - 23.7 times faster. The difference of I/O throughput on random

read becomes larger at smaller I/O size settings. The throughput of sequential read does not depend on I/O sizes because of the read ahead function, while in contrast, the throughput of random read is proportional to I/O sizes.

4.3 Throughput of Mixture Workload

The I/O workload of actual applications consists of both read and write functions. The internal parallelism and high random access throughput of SSDs take advantage of such workloads. To demonstrate this, we used a benchmark program of a mixture of I/Os. The workload consisted of read 75% and write 25%. The ratio of read/write was similar to our experimental hashjoin operation mentioned in Section 5. The benchmark executes operations as follows. (1) Open two files: one for read operations and the other for write operations. (2) Issue I/O operations in a specified I/O size. First three read I/Os are issued, then one write I/O is issued, and then the process is repeated. The read operations sequentially scan a file and write operations add data to the other empty file.

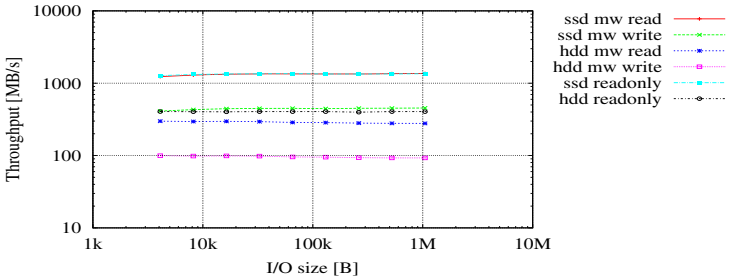


Fig. 2. I/O throughput for mixture workload of read and write accesses on SSD and HDD

Figure 2 shows the throughputs of the mixture workload (mw) for SSD and HDD. The results of readonly workloads (mentioned in Section 4.2) are plotted for comparison. Read/write throughputs of the mixture workload on SSD are both 4.2 times higher than on HDD. The difference of the read throughput of the mixture workload between SSDs and HDDs is larger than that of readonly workload. The read throughput of the mixture workload on SSD is the same as that of the readonly workload. In contrast, on HDD, the read throughput on the mixture workload is 0.74 times smaller than that of the readonly workload. This result indicates that SSDs are capable of parallel I/O processing and suitable for mixture I/O workloads.

5 Experimental Analysis of Hashjoin Operation

We performed experiments with hashjoin queries and analyzed the I/O throughput improvement and access costs on the cache hierarchy of large data query processing on an SSD-based database.

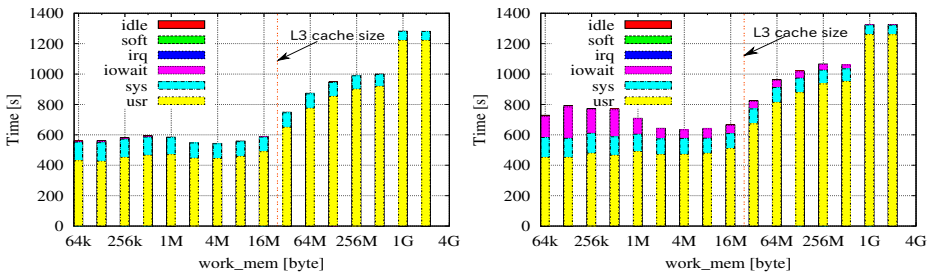
5.1 Database Setup and Workload

We used PostgreSQL [11] for RDBMS and set shared buffer size to 8 GB. We created the same databases on SSD and HDD by using data of TPC-H [12] benchmark at a scale factor of 100. Hashjoin processing performance depends on the number of partition files and hashtable size. The more the number of partition files is increased, the smaller the hashtable size becomes. In the case of HDD-based databases, it is preferable to decrease the number of partition files. This is because the fragmentation of files causes serious I/O performance degradation. There is a trade-off related to working memory size, so we handle it as a parameter to control the workload of hashjoin and observe the processing performance for each value. `Work_mem` is a PostgreSQL parameter that describes the in-memory buffer size per database operation, that is, the working memory size in a hashjoin. Since PostgreSQL uses hybrid hashjoin, when `work_mem` is larger than the entire hashtable size, no partitions are written to disk. We experimented on two queries in SSD and HDD environments: (1) a single join query, with the join part and lineitem tables on `partkey`, and (2) a realistic workload query, with TPC-H query 8, which contains the join of 8 tables.

We measured the query execution time by changing the `work_mem` size between 64 kB and 2 GB. To observe the breakdown of CPU utilization, mpstat(1) is used, and L3 cache references and cache misses are measured by a Linux profiler perf[13].

5.2 Single Join Query

To demonstrate that SSDs improve the throughput of query executions, we experimented with join operation on part and lineitem tables. Each tuple of a lineitem table is joined with one tuple of a part table which has the same `partkey` in this query. The table sizes of the part and the lineitem are 20 GB and 86 GB, respectively. The hashtable is created on the part table, and its total size is about 800 MB.



(a) SSD **(b)** HDD

Fig. 3. Single join query execution time

Figure 3 shows the hashjoin execution time and its breakdown (usr, system, iowait, irq, soft irq, and idle) for the respective `work_mem` values in the SSD or HDD environment. In the figure, usr indicates CPU operational cost and the total of sys and iowait indicates I/O operational cost.

When `work_mem` is smaller than L3 cache size (64 kB - 16 MB), SSD and HDD show different trends. The smaller `work_mem` is, the more I/O cost is stacked up on HDD, because the I/O throughput is saturated owing to the fragmentation. In contrast, the SSD results show that I/O costs are lower than HDDs and approximately not changed in every point, which indicates remaining I/O bandwidth.

When `work_mem` is larger than an L3 cache size (larger than 32 MB), the CPU cost is growing in both environments and the I/O cost is no longer a bottleneck. This is due to the increased number of cache misses because the hashtable size is too big for the L3 cache size.

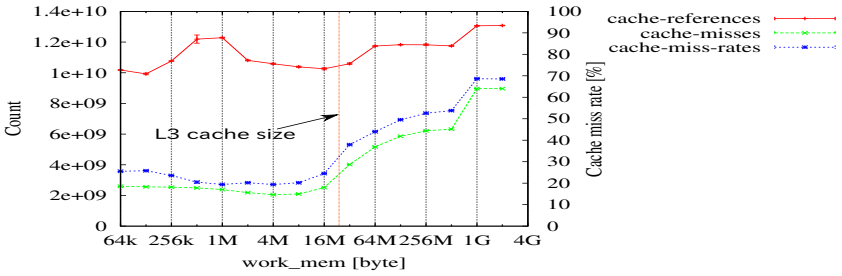


Fig. 4. Number of cache references/misses and cache miss rates for each `work_mem` sizes on SSD

Figure 4 shows the number of L3 cache references, misses, and miss rates for each `work_mem` on the SSD measurement. For example, execution with `work_mem` = 1 GB has 7×10^9 larger cache misses than 4 MB, and the DRAM access latency is about 100 nanoseconds in our experimental environment. Consequently, execution with `work_mem` = 1 GB gets a $7 \times 10^9 \times 100(\text{ns}) = 700(\text{s})$ larger cache miss penalty, which is consistent with the difference of CPU cost between 4 MB and 1 GB in Figure 3a.

When `work_mem` is larger than the entire hashtable size (larger than about 800 MB), only one partition is created and then execution time becomes the same. The reason for the steeply increasing cache misses from `work_mem` = 512 MB to 1 GB is that the average bucket length of a hashtable is larger on `work_mem` = 1 GB. This is the implementation dependent problem for the hashjoin of PostgreSQL. The average bucket length for each `work_mem` is 2.2 on 64 kB, 3.8 on 128 kB, 5.7 on $256kB - 512MB$, and 10.5 on larger than 1 GB, and the number of lineitem tuples is 6×10^8. Then, the difference of the total number of bucket scans between 512 MB and 1 GB is $(10.5 - 5.7) \times 6 \times 10^8 \approx 3 \times 10^9$, which fits the difference of the number of cache misses in Figure 4.

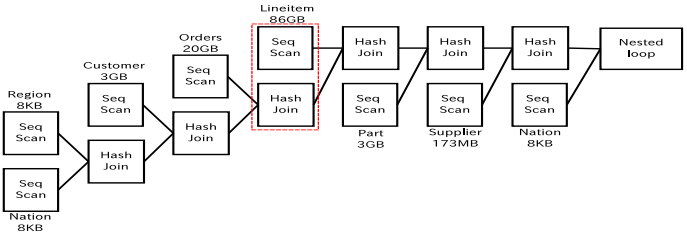
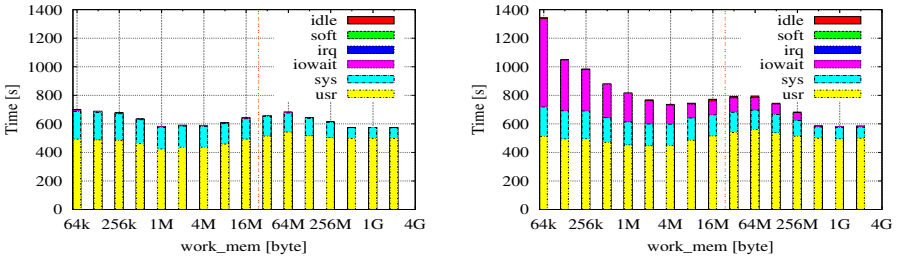


Fig. 5. The query execution plan of TPC-H query 8

5.3 TPC-H Query

We measured the TPC-H query to demonstrate that the bottleneck of query processing changes the same way as a single join in actual DSS queries. We used query 8, which contains the join of 8 tables. Some calculation parts of the query are removed, since we are interested in only the I/O performance behavior of the query. The execution plan is as shown in Figure 5. For each hashjoin operation, a hashtable is created on the bottom side node in Figure 5. At the point where I/O is the heaviest (enclosed by a red circle in Figure 5), the total hashtable size is about 400 MB.

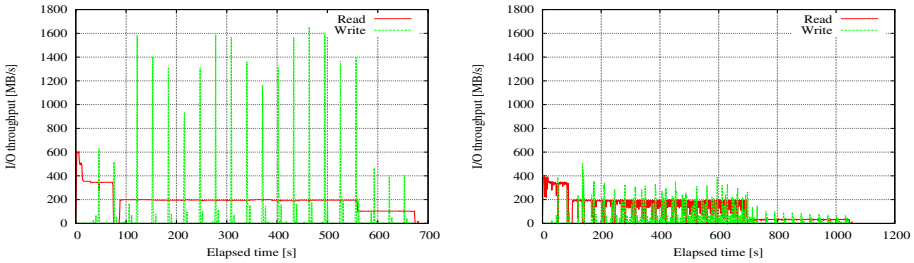


(a) SSD **(b)** HDD

Fig. 6. TPC-H query 8 execution time

Figure 6 shows the results of query measurement on SSD and HDD. The difference of I/O cost between SSD and HDD is more conspicuous than single join query.

Figure 7 shows the I/O throughput timeline during the execution of query 8 on SSD and HDD, which are observed under `work_mem` = 128 kB. In the phase of the lineitem table scan (about 90 - 550 seconds in Figure 7a, 100 - 700 seconds in Figure 7b), the read I/O throughput is sometimes decreased by the write I/O, which writes hashtable partitions to storage during HDD execution. This is not observed during SSD execution, since SSDs can process multiple I/Os in parallel, as mentioned in Section 4.3. In the phase of hashjoin probing after the lineitem



(a) SSD **(b)** HDD

Fig. 7. The timeline of I/O throughput during query 8 execution (`work_mem` = 128 kB)

scan (about 550 - 680 seconds in Figure 7a, 700 - 1050 seconds in Figure 7b), the processing time on SSD is about 2.7 times faster than HDD. Since partition files are fragmented when `work_mem` is small, the I/O throughput of HDD is low, which inhibits the processing performance.

The I/O cost starts to increase from `work_mem` = 8 MB in Figure 6, even though it is smaller than the L3 cache size. In the build phase of hybrid hashjoin, in order to process the tuple matching of the memory residing hashtable without temporarily writing them to storage, the join result for those tuples is directly passed to the next operator in a pipeline style. For this reason, some hashtables may simultaneously reside in memory when multiple hashjoins are included. Three hashtables share a cache in this query, so setting `work_mem` as 8 MB fills up the L3 cache.

6 Discussion

We discussed the performance bottleneck of hashjoin in Section 3.2, that is, I/O and cache misses. The result of a single hashjoin execution on a HDD (Figure 3b) shows that our assumption was correct: the less `work_mem` used, the higher the I/O cost. This is a result of the increased number of fragmented files. On the other hand, when `work_mem` is larger than the L3 cache size, the CPU cost becomes large because of the increasing number of cache misses. However, the result of multiple hashjoins on an HDD (Figure 6b) suggests that the number of cache misses has no relation to the `work_mem` setting at a multiple hashjoin query. The CPU cost does not grow as in a single join query even if `work_mem` is much larger than the L3 cache size. For this reason, query execution time becomes shorter when we set `work_mem` to 512 MB or larger. This is a peculiar case for the high data distribution locality of lineitem and orders tables, as follows. The predicate of the third join operation, which is indicated by a red circle in Figure 5, is `orders.orderkey` = `lineitem.orderkey`. There are four tuples on average that have the same `orderkey` on the lineitem table. Tuples are stored in ascendant order of `orderkey` in the lineitem table, so several tuples with the same `orderkey` are concentrated in the table, that is, the data access

locality occurs on a hashtable. Therefore, the same hash bucket is likely to be accessed successively at the probe phase, and consequently the number of cache misses becomes small. On the other hand, tuples with the same `partkey` are not concentrated, so the number of cache misses becomes large in a single query. If data locality is low in TPC-H query 8, the CPU cost increases when `work_mem` is larger than 8 MB, as in the results of a single query execution, and then near `work_mem` = 4 MB points would be optimal. In such a case, to avoid an increase of cache misses, it is inevitable to get some overheads of I/O fragmentation on the HDD-based database. These I/O cost overheads are decreased on the SSD-based database. The measurements on SSD in Figure 3a indicate that the query execution time is not affected by I/O fragmentation and rather is likely to be affected by the cache miss penalties.

Considering these results, when SSDs are used, it is better to keep the working memory size small enough to fit the hashtable into the cache. However, in the current HDD-based hashjoin implementation, very large memory is required to decrease the number of fragmented files. The working memory size can be reduced as long as the fragmentation of partitions does not cause I/O bottleneck in SSD-based databases. In our experiments, there was no I/O bottleneck even if `work_mem` was 64 kB.

As a result of using less memory space for a hashjoin operation, the portion of cache and memory space remains free. This remaining cache will help improve the performance of complex queries and parallel multiple queries. A complex query such as multiple hashjoin operations are executed in a pipeline manner. For example, TPC-H query 8 deploys three hashtables at the same time. If all hashtables can reside together in a cache, the number of cache misses becomes small. Another case of utilization is the parallel execution of multiple queries. The cache and other computing resources may not be fully consumed by sequential query processing. (Here, by other computing resources we mean CPU cores (most current processors have several cores internally) and I/O bandwidth (I/O bandwidth of SSDs becomes wider by parallel I/O processing such as mixture I/O workload for its internal parallelism)). Parallel query execution enables us to utilize remaining resources and improve the entire query processing performance.

7 Conclusion

In this paper, we experimentally analyzed the performance improvement and newly observed bottlenecks of large data query processing in SSD-based databases. Our experiments on hashjoin queries showed that cache miss penalties seriously affected the query processing performance. We found that it is preferable to set a small hashtable size to fit into the cache on SSD-based databases, as this reduces the number of cache misses at the probe phase. Hashtable size should be relatively large on HDD-based databases because I/O cost becomes large in a small hashtable size on HDDs. This is due to the poor I/O throughput of HDDs under fragmentation caused by generating many hashtable partition files when the hashtable is small. In contrast, the I/O cost of SSDs is not increased by the fragmentation. Thus, considering data access locality of hashjoin

is more important at the query execution in SSD-based databases. Experiments on a modern SSD-based system showed that hashtable size can be reduced to 64 kB without any increase to I/O cost by the fragmentation. As a result of reducing hashtable size, the portion of cache and memory space that are not used by hashtable remains free. Those remaining resources can be utilized to improve the performance of a multiple hashjoin query such as TPC-H query 8. Another promising way to utilize the remaining resources is parallel execution of multiple queries. Exploring data access locality of multiple queries will be the focus of our future work.

Acknowledgment. This work is partially supported by JSPS KAKENHI Grant Number 24300034 and 26280130.

References

1. Bhattacharjee, B., Ross, K.A., Lang, C., Mihaila, G.A., Banikazemi, M.: Enhancing recovery using an SSD buffer pool extension. In: DaMoN 2011, pp. 10–16. ACM (2011)
2. Canim, M., Mihaila, G.A., Bhattacharjee, B., Ross, K.A., Lang, C.A.: SSD bufferpool extensions for database systems. Proc. VLDB Endow. 1435–1446 (2010)
3. Kang, W.H., Lee, S.W., Moon, B.: Flash-based extended cache for higher throughput and faster recovery. Proc. VLDB Endow. 5(11), 1615–1626 (2012)
4. Do, J., Zhang, D., Patel, J.M., De Witt, D.J., Naughton, J.F., Halverson, A.: Turbocharging DBMS buffer pool using SSDs. In: SIGMOD 2011, pp. 1113–1124. ACM (2011)
5. Koltsidas, I., Viglas, S.D.: Flashing up the storage layer. Proc. VLDB Endow. 1(1), 514–525 (2008)
6. Luo, T., Lee, R., Mesnier, M., Chen, F., Zhang, X.: hStorage-DB: Heterogeneity-aware data management to exploit the full capability of hybrid storage systems. Proc. VLDB Endow. 5(10), 1076–1087 (2012)
7. Li, Y., He, B., Yang, R.J., Luo, Q., Yi, K.: Tree indexing on solid state drives. Proc. VLDB Endow. 3(1-2), 1195–1206 (2010)
8. Tsirogiannis, D., Harizopoulos, S., Shah, M.A., Wiener, J.L., Graefe, G.: Query Processing Techniques for Solid State Drives. In: SIGMOD 2009, pp. 59–72. ACM (2009)
9. Kitsuregawa, M., Tanaka, H., Moto-Oka, T.: Relational Algebra Machine GRACE. In: Goto, E., Furukawa, K., Nakajima, R., Nakata, I., Yonezawa, A. (eds.) RIMS 1982. LNCS, vol. 147, pp. 191–214. Springer, Heidelberg (1983)
10. Schneider, D.A., De Witt, D.J.: A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In: SIGMOD 1989, pp. 110–121. ACM (1989)
11. PostgreSQL, `http://www.postgresql.org/`
12. Transaction Processing Performance Council, An ad-hoc, decision support benchmark, `http://www.tpc.org/tpch/`
13. Perf, `https://perf.wiki.kernel.org/`