Join Processing for Flash SSDs: Remembering Past Lessons

Jaeyoung Do
Univ. of Wisconsin-Madison
jae@cs.wisc.edu

Jignesh M. Patel
Univ. of Wisconsin-Madison
jignesh@cs.wisc.edu

ABSTRACT

Flash solid state drives (SSDs) provide an attractive alternative to traditional magnetic hard disk drives (HDDs) for DBMS applications. Naturally there is substantial interest in redesigning critical database internals, such as join algorithms, for flash SSDs. However, we must carefully consider the lessons that we have learnt from over three decades of designing and tuning algorithms for magnetic HDD-based systems, so that we continue to reuse techniques that worked for magnetic HDDs and also work with flash SSDs.

The focus of this paper is on recalling some of these lessons in the context of *ad hoc* join algorithms. Based on an actual implementation of four common ad hoc join algorithms on both a magnetic HDD and a flash SSD, we show that many of the "surprising" results from magnetic HDD-based join methods also hold for flash SSDs. These results include the superiority of block nested loops join over sort-merge join and Grace hash join in many cases, and the benefits of blocked I/Os. In addition, we find that simply looking at the I/O costs when designing new flash SSD join algorithms can be problematic, as the CPU cost is often a bigger component of the total join cost with SSDs. We hope that these results provide insights and better starting points for researchers designing new join algorithms for flash SSDs.

1. INTRODUCTION

Flash solid state drives (SSDs) are actively being considered as storage alternatives to replace or dramatically reduce the central role of magnetic hard disk drives (HDDs) as the main choice for storing persistent data. Jim Gray's prediction of "Flash is disk, disk is tape, and tape is dead" [7] is coming close to reality in many applications. Flash SSDs, which are made by packaging (NAND) flash chips, offer several advantages over magnetic HDDs including faster random reads and lower power consumption. Moreover, as flash densities continue to double as predicted in [9], and prices continue to drop, the appeal of flash SSDs for DBMSs increases. In fact, vendors such as Fusion-IO and HP sell flash-based devices as I/O accelerators for many data-intensive workloads.

The appeal of flash SSDs is also attracting interest in redesigning various aspects of DBMS internals for flash SSDs. One such aspect that is becoming attractive as a research topic is join processing algorithms, as it is well–known that joins can be expensive and can play a critical role in determining the overall performance of the DBMS. While such efforts are well-motivated, we want to approach a redesign of database query processing algorithms by clearly recalling the lessons that the community has learnt from over three decades of research in query processing algorithms. The focus of this paper is on recalling some of the important lessons that we have learnt about efficient join processing in magnetic HDDs, and determining if these lessons also apply to joins using flash SSDs. In addition, if previous techniques for tuning and improving the join performance also work for flash SSDs, then it also changes what are interesting starting point for comparing new SSD-based join algorithms.

In the case of join algorithms, a lot is known about how to optimize joins with magnetic HDDs to use the available buffer memory effectively, and to account for the characteristics of the I/O subsystem. Specifically, the paper by Haas *et al.* [8] derives detailed formulae for buffer allocation for various phases of common join algorithms such as block nested loops join, sort-merge join, Grace hash join, and hybrid hash join. Their results show that the right buffer pool allocation strategy can have a huge impact – upto 400% improvements in some cases. Furthermore, the relative performance of the join algorithms changes once you optimize the buffer allocations – block nested loops join is much more versatile, and Grace hash join is often not very competitive.

The dangers of forgetting these lessons could lead to an incorrect starting point for comparing new flash SSD join algorithms. For example, the comparison of RARE-join [16] with Grace hash join [10] to show the superiority of the RARE-join algorithm on flash SSDs is potentially not the right starting point. (It is possible that the RARE-join is superior to the best magnetic HDD-based join method when run over flash SSDs, but this question has not been answered conclusively.) As we show in this paper, in fact even block nested loops join far outperforms Grace hash join in most cases, on both magnetic HDDs and flash SSDs. Cautiously, we note that we have only tried one specific magnetic HDD and one specific flash SSD, but even this very first test produced interesting results. (As part of future work, we want to look at wider range of hardware for both flash SSDs and magnetic HDDs.)

Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN 2009) June 28, 2009, Providence, Rhode-Island

The focus of this paper in on investigating four popular ad hoc join algorithms, namely block nested loops join, sort-merge join, Grace hash join, and hybrid hash join, on both flash SSDs and magnetic HDDs. We start with the best buffer allocation methods that are proposed in [8] for these join algorithms, and first ask the question: What changes for these algorithms as we replace a magnetic HDD with a flash SSD? Then, we study the effect of changing the buffer pool sizes and the page sizes and examine the impact of these changes on these join algorithms. Our results show that many of the techniques that were invented for joins on magnetic HDDs continue to hold for flash SSDs. As an example, *blocked* I/O is useful on both magnetic HDDs and flash SSDs, though for different reasons. In the case of magnetic HDDs, the use of blocked I/O amortizes the cost of disk seeks and rotational delays. On the other hand, the benefit of blocked I/O with flash SSDs comes from amortizing the latency associated with the software layers of flash SSDs, and generating fewer erase operations when writing data.

The remainder of this paper is organized as follows. In Section 2, we briefly introduce the characteristics of flash SSDs. Then we introduce the four classic join algorithms with appropriate assumptions and buffer allocation strategies in Section 3. In Section 4, we explain and discuss the experimental results. After reviewing related work in Section 5, we conclude in Section 6.

2. CHARACTERISTICS OF FLASH SSD

Flash SSDs are based on NAND flash memory chips and use a controller to provide a persistent block device interface. A flash chip stores information in an array of memory cells. A chip is divided into a number of *flash blocks*, and each flash block contains several *flash pages*. Each memory cell is set to 1 by default. To change the value to 0, the entire block has to be erased by setting it to 1, followed by selectively programming the desired cells to 0. Read and write operations are performed at the granularity of a flash page. On the other hand, the time-consuming erase operations can only be done at the level of a flash block. Considering the typical size of a flash page (4 KB) and a flash block (64 flash pages), the erase-before-write constraint can significantly degrade write performance. In addition, most flash chips only support $10^5 \sim 10^6$ erase operations per flash block. Therefore, erase operations should be distributed across the flash blocks to prolong the service life of flash chips. These kinds of constraints are handled by a software layer known as *flash translation layer* (FTL).

The major role of the FTL is to provide address mappings between the *logical block addresses* (LBAs) and flash pages. The FTL maintains two kinds of data structures: *A direct map* from LBAs to flash pages, and *an inverse map* for rebuilding the direct map during recovery. While the inverse map is stored on flash, the direct map is stored on flash and at least partially in RAM to support fast lookups. If the necessary portion of the direct map is not in RAM, it must be swapped in from flash as required.

While flash SSDs have no mechanically moving parts, data access still incurs some latency, due to overheads associated with the FTL logic. However, latencies of flash SSDs are typically much smaller than those of magnetic HDDs [4].

3. JOINS

In this section, we introduce four classic ad hoc join algorithms that we consider in this paper, namely: block nested loops join, sort-merge join, Grace hash join, and hybrid hash join. The two relations being joined are denoted as R and S. We use $|R|$, $|S|$ and B to denote the sizes of the relations and the buffer pool size in pages, respectively. We also assume that $|R| \leq |S|$. Each join algorithm needs some extra space to build and maintain specific data structures such as a hash table or a tournament tree. In order to model these structures, we use a multiplicative *fudge factor*, denoted as F.

Next we briefly describe each join algorithm. We also outline the buffer allocation strategy for each join algorithm. The I/O costs for writing the final results are omitted in the discussion below, as this cost is identical for all join methods. For the buffer allocation strategy we directly use the recommendations by Haas *et al.* [8] (for magnetic HDDs), which shows that optimizing buffer allocations can dramatically improve the performance of join algorithms (by 400% in some cases).

3.1 Block Nested Loops Join

Block nested loops join first logically splits the smaller relation R into same size chunks. For each chunk of R that is read, a hash table is built to efficiently find matching pairs of tuples. Then, all of S is scanned, and the hash table is probed with the tuples. To model the additional space required to build a hash table for a chunk of R we use the fudge factor F, so a chunk of size $|C|$ pages uses $F|C|$ pages in memory to store a hash table on C.

The buffer pool is simply divided into two spaces; one space, I_{outer}, is for an input buffer with a hash table for R chunks, and another one, I_{inner}, is for an input buffer to scan S. Note that reading R in chunks of size $\frac{I_{outer}}{F}$ ($= |C|$) guarantees sufficient memory to build a hash table in memory for that chunk [5].

3.2 Sort-Merge Join

Sort-merge join starts by producing sorted runs of each R and S. After R and S are sorted into runs on disk, sort-merge join reads the runs of both relations and merges/joins them. We use the tournament sort (*a.k.a.* heap sort) in the first pass, which produces runs that on average are twice the size of the memory used for the initial sorting [11]. We also assume $B > \sqrt{F|S|}$ so that the sort-merge join, which uses a tournament tree, can be executed in two passes [17].

In the first pass, the buffer pool is divided into three parts: an input buffer, an output buffer, and working space (WS) to maintain the tournament tree. During the second pass, the buffer pool is split across all the runs of R and S as evenly as possible.

3.3 Grace Hash Join

Grace hash join has two phases. In the first phase, it reads each relation, applies a hash function to the input tuples, and hashes tuples into buckets that are written to disk. In the second phase, the first bucket of R is loaded into the buffer pool, and a hash table is built on it. Then, the corresponding bucket of S is read and used to probe the hash table. Remaining buckets of R and S are handled in the same way iteratively. We assume $B > \sqrt{F|R|}$ to allow for a two-phase Grace hash join [17].

Join Algorithm	First Phase/Pass		Second Phase/Pass	
Block Nested Loops Join	I_{outer} / I_{inner}	$I_{inner} = \lceil \frac{\sqrt{y\|S\|(y\|S\|+B(y+\|S\|))}-y\|S\|}{y+\|S\|} \rceil$ $y = \frac{Dl}{Dx}$ $I_{outer} = B - I_{inner}$		
Sort-Merge Join	WS / I | O	$I = O = \lceil \frac{(\sqrt{2z-4})\cdot B}{z-8} \rceil$ $z = \frac{(Dl+Ds)\cdot F(\|R\|+\|S\|)}{Dl\cdot B}$ $WS = B - I - O$	I_1 / ⋮ / I_{NR+NS}	$I \simeq \lfloor \frac{B}{NR+NS} \rfloor$
Grace Hash Join	I | O_1 / ⋮ / O_k	$k = \lceil \frac{\|R\|F+\sqrt{\|R\|^2F^2+4B\|R\|F}}{2B} \rceil$ $O = \lfloor \frac{B}{k+1} \rfloor$ $I = B - k \cdot O$	WS' / I	$WS' \simeq \lceil \frac{F\|R\|}{k} \rceil$ $I = B - WS'$
Hybrid Hash Join	WS | O_1 / ⋮ / I | O_k	$I = O = \lceil 1.1\sqrt{B} \rceil$ $k = \lceil \frac{F\|R\|-(B-I)}{B-I-O} \rceil$ $WS = B - I - k \cdot O$	WS' / I	$WS' \simeq \lceil \frac{F\|R\|-WS}{k} \rceil$ $I = B - WS'$

Table 1: Buffer allocations for join algorithms from Haas *et al.* [8]: Ds, Dl, and Dx denote average seek time, latency, and page transfer time for magnetic HDDs, respectively

There are two sections in the buffer pool during the first partitioning phase: one input buffer and an output buffer for each of the k buckets. We subdivide the output buffer as evenly as possible based on the number of buckets, and then give the remaining pages, if any, to the input buffer. In the second phase, a portion of the buffer pool (WS') is used for the i^{th} bucket of R and its hash table, and the remaining pages are chosen as input buffer pages to read S.

3.4 Hybrid Hash Join

As in the Grace hash join, there are two phases in this algorithm, assuming $M > \sqrt{F|R|}$. In the first phase, the relation R is read and hashed into buckets. Since a portion of the buffer pool is reserved for an in-memory hash bucket for R, this bucket of R is not written to a storage device while the other buckets are. Furthermore, as S is read and hashed, tuples of S matching with the in-memory R bucket can be joined immediately, and need not be written to disk. The second phase is the same as Grace hash join.

During the first phase, the buffer pool is divided into three parts: one for the input, one for the output of k buckets excluding the in-memory bucket, and the working space (WS) for the in-memory bucket. The buffer allocation scheme for the second phase is the same as Grace hash join.

3.5 Buffer Allocation

Table 1 shows the optimal buffer allocations for the four join algorithms, for each phase of these algorithms. Note that block nested loops join does not distinguish between the different passes.

In this paper, we use the same buffer allocation method for both flash SSDs and magnetic HDDs. While these allocations may not be optimal for flash SSDs, our goal here is to start with the best allocation strategy for magnetic HDDs and explore what happens if we simply use the same settings when replacing a magnetic HDD with a flash SSD.

Comments	**Values**
Magnetic HDD cost	129.99 $ (0.36 $/GB)
Magnetic HDD average seek time	12 ms
Magnetic HDD latency	5.56 ms
Magnetic HDD transfer rate	34 MB/s
Flash SSD cost	230.99 $ (3.85 $/GB)
Flash SSD latency	0.35 ms
Flash SSD read transfer rate	120 MB/s
Flash SSD write transfer rate	80 MB/s
Page size	2 KB ∼ 32 KB
Buffer pool size	100 MB ∼ 600 MB
Fudge Factor	1.2
orders table size	5 GB
customer table size	730 MB

Table 2: Device characteristics and parameter values

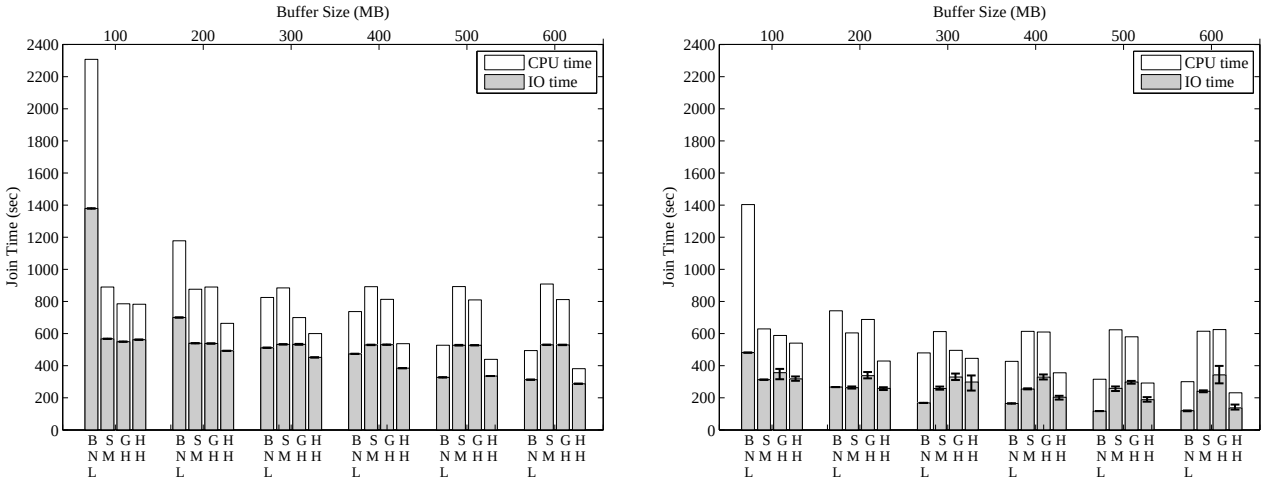
4. EXPERIMENTS

In this section, we compare the performance of the join algorithms using the optimal buffer allocations when using a magnetic HDD and a flash SSD for storing the input data sets. Each data point presented here is the average over three runs.

4.1 Experimental Setup

We implemented a single-thread and light-weight database engine that uses the SQLite3 [1] page format for storing relational data in heap files. Each heap file is stored as a file in the operating system, and the average page utilization is 80%. Our engine has a buffer pool that allows us to control the allocation of pages to the different components of the join algorithms. The engine has no concurrency control or recovery mechanisms.

Our experiments were performed on a Dual Core 3.2GHz Intel Pentium machine with 1 GB of RAM running Red Hat



(a) Magnetic HDD (b) Flash SSD

Figure 1: Varying the size of the buffer pool (8 KB page, blocked I/O)

	Buffer Pool Size											
	100 MB		200 MB		300 MB		400 MB		500 MB		600 MB	
Algorithms	Join	I/O	Join	I/O	Join	I/O	Join	I/O	Join	I/O	Join	I/O
BNL	1.64X	2.86X	1.59X	2.62X	1.72X	3.04X	1.73X	2.87X	1.67X	2.79X	1.65X	2.61X
SM	1.41X	1.81X	1.45X	2.05X	1.44X	2.06X	1.45X	2.08X	1.43X	2.04X	1.48X	2.20X
GH	1.34X	1.54X	1.29X	1.59X	1.41X	1.62X	1.33X	1.62X	1.39x	1.77X	1.30X	1.55X
HH	1.45X	1.77X	1.55X	1.90X	1.35X	1.51X	1.51X	1.89X	1.50x	1.78X	1.65X	2.09X

Table 3: Speedups of total join times and I/O times with flash SSDs

Enterprise 5. For the comparison, we used a 5400 RPM TOSHIBA 320 GB external HDD and a OCZ Core Series 60GB SATA II 2.5 inch flash SSD.

We used wall clock time as a measure of execution time, and calculated the I/O time by subtracting the reported CPU time from the wall clock time. Since *synchronous* I/Os were used for all tests, we assumed that there is no overlap between the I/O and the computation. We also used *direct* I/Os so that the database engine transfers data directly from/to the buffer pool bypassing the OS cache (so there is no prefetching and double buffering). With this setup all join numbers repeated here are "cold" numbers.

4.2 Data Set and Join Query

As our test query, we used a primary/foreign key join between the TPC-H [2] *customer* and the *orders* tables, generated with a scale factor of 30. The *customer* table contains 4,500,000 tuples (730 MB), and the *orders* table has 45,000,000 (5 GB). Each tuple of both tables contains an unsigned 4 byte integer key (the customer key), and an average 130 and 90 bytes of padding for the *customer* and the *orders* tables respectively. The data for both tables were stored in random order in the corresponding heap files.

Characteristics of the magnetic HDD and the flash SSD, and parameter values used in these experiments are shown in Table 2.

4.3 Effect of Varying the Buffer Pool Size

The effect of varying the buffer pool size from 100 MB to 600 MB is shown in Figure 1, for both the magnetic HDD and the flash SSD. We also used *blocked I/O* to sequentially read and write multiple pages in each I/O operation. The size of the I/O block is calculated for each algorithm using the equations shown in Table 1.

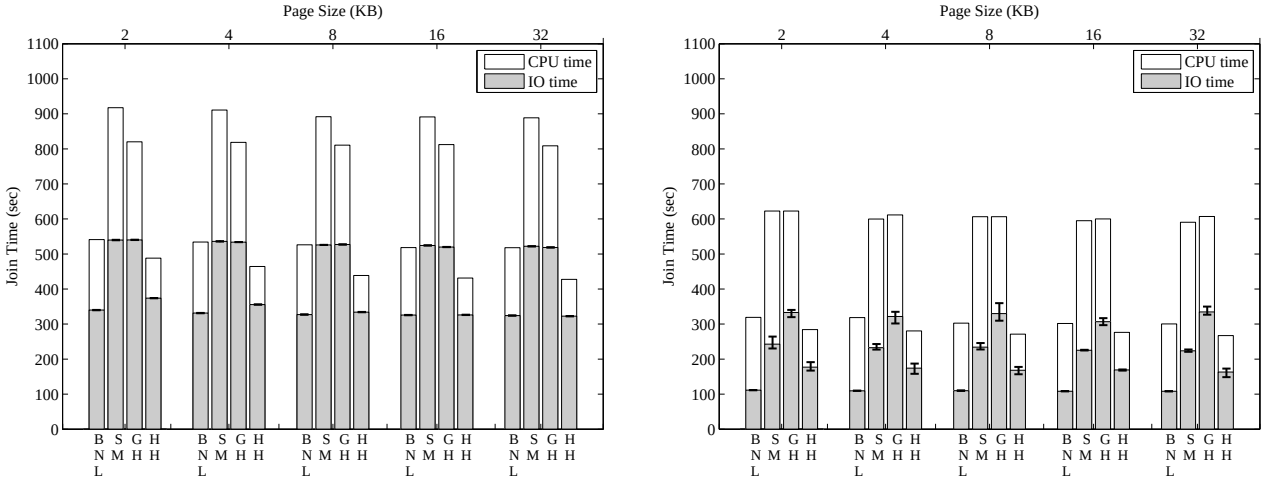
In Figure 1 error bars denote the minimum and the maximum measured I/O times (across the three runs). Note that the error bars for the CPU times are omitted, as their variation is usually less than 1% of the total join time.

Table 3 shows the speedup of the total join times and the I/O times of the four join algorithms under different buffer pool sizes. The results in Table 3 show that replacing the magnetic HDD with the flash SSD benefits all the join methods. The block nested loops join whose I/O pattern is sequential reads shows the biggest performance improvement, with speedup factors between 1.59X to 1.73X. (Interestingly, a case can be made that for sequential reads and writes, comparable or much higher speedups can be achieved with striped magnetic HDDs, for the same $ cost [15].)

Other join algorithms also performed better on the flash SSD compared to the magnetic HDD, with smaller speedup improvements than the block nested loops join. This is because the write transfer rate is slower than the read transfer rate on the flash SSD (See Table 2), and unexpected erase operations might degrade write performance further.

	Algorithms					
	Sort-Merge Join		Grace Hash Join		Hybrid Hash Join	
Buffer Pool Size	First Phase	Second Phase	First Phase	Second Phase	First Phase	Second Phase
100 MB	1.52X	3.00X	1.34X	2.27X	1.57X	2.61X
200 MB	1.83X	2.81X	1.43X	2.19X	1.66X	3.09X
300 MB	1.86X	2.79X	1.47X	2.12X	1.34X	2.32X
400 MB	1.90X	2.63X	1.47X	2.13X	1.70X	2.91X
500 MB	1.81X	2.86X	1.59X	2.44X	1.63X	2.83X
600 MB	2.00X	2.89X	1.31X	2.68X	1.98X	2.84X

Table 4: Speedups of I/O times with flash SSDs, broken down by the first and second phases



(a) Magnetic HDD (b) Flash SSD

Figure 2: Varying the page size (500 MB Buffer Pool, blocked I/O)

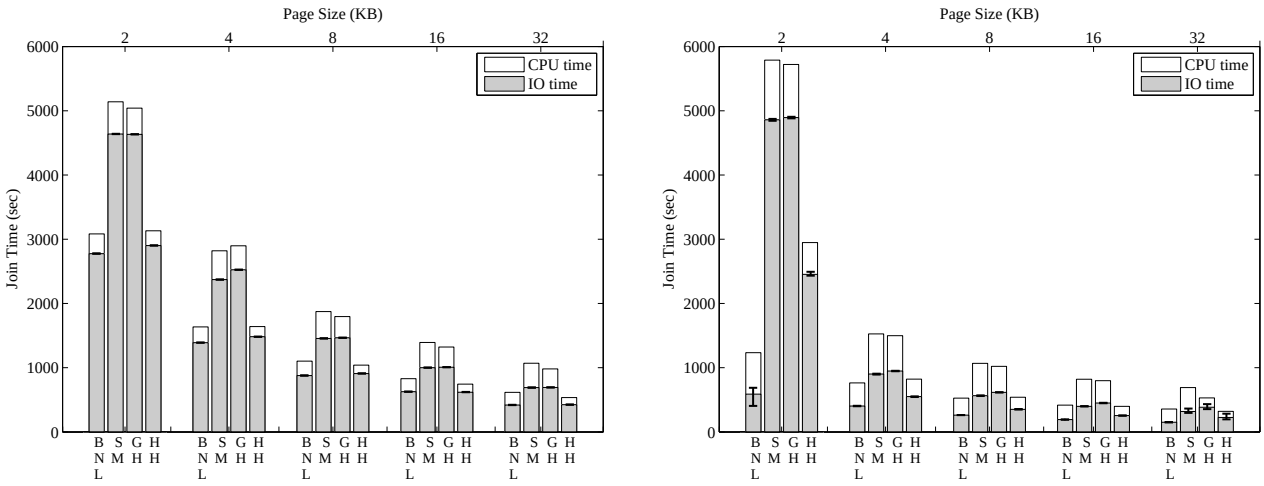
As an example, different I/O speedups were achieved in the first and the second phases of the sort-merge join as shown in Table 4. While the I/O speedup of the second phase was between 2.63X and 3.0X due to faster random reads, the I/O speedup in the first phase (that has sequential writes as the dominant I/O pattern), was only between 1.52X and 2.0X, which reduced the overall speedup for sort-merge join.

In the case of Grace hash join, all the phases were executed with lower I/O speedups than those of the sort-merge join (See Table 4). Note that the dominant I/O pattern of Grace hash join is random writes in the first phase, followed by sequential reads in the second phase. While the I/O speedup between 2.12X and 2.68X was observed for the second phase of Grace hash join, the I/O speedup of its first phase was only between 1.31X and 1.59X due to expensive erase operations. This indicates that algorithms that stress random reads, and avoid random writes as much as possible are likely to see bigger improvements on flash SSDs (over magnetic HDDs).

While there is little variation in the I/O times with the magnetic HDD (See the error bars in Figure 1(a) for the I/O bars), we observed higher variations in the I/O times with the flash SSD (Figure 1(b)), resulting from the varying write performance. Note that since random writes cause more erase operations than sequential writes, hash-based joins show wider range of I/O variations than sort-merge join. On the other hand, there is little variation in the I/O costs for block nested loops join regardless of the buffer pool size, since it does not incur any writes.

Another interesting observation that can be made here (Figure 1) is the relative I/O performance between the sort-merge join and Grace hash join. Both have similar I/O costs with the magnetic HDD, but sort-merge join has lower I/O costs with the flash SSD. This is mainly due to the different output patterns of both join methods. In the first phase of the joins, where both algorithms incur about 80% of the total I/O cost, each writes intermediate results (sorted runs for sort-merge join, and buckets for Grace hash join) to disk in different ways; sort-merge join incurs sequential writes as opposed to the random writes that are incurred by Grace hash join. While this difference in the output patterns has a substantial impact on the join performance with the flash SSD because random writes generate more erase operations than sequential writes, the impact is relatively small with the magnetic HDD.

In addition, we can not argue that the sort-merge join algorithm is better than the Grace hash join algorithm on



(a) Magnetic HDD (b) Flash SSD

Figure 3: Varying the page size (500 MB Buffer Pool, page sized I/O)

flash SSDs based solely on the I/O performance results (as in [16]). While looking at only the I/O characteristics is sometimes okay for magnetic HDDs (when the join algorithm is I/O bound), using flash SSDs makes the CPU costs more prominent. As seen in Figure 1(b), the CPU cost now dominate the I/O cost in most cases for block nested loops join and sort-merge join. In general, with flash SSDs the balance between CPU and I/Os changes, which implies that when building systems with flash SSDs we may want to consider adding more CPU processing power.

From Figure 1, we notice that hybrid hash join outperformed all other algorithms on both the magnetic HDD and the flash SSD, across the entire range of buffer pool sizes that we tested. Hybrid hash join is 2.81X faster than Grace hash join with a 600 MB buffer pool, indicating that comparing only with Grace hash join (as done in [14, 16]) could be misleading. Finally note that in our experiments, with the flash SSD, block nested loops join is faster than sort-merge join and Grace hash join for large buffer pool sizes, but slower than hybrid hash join, which is more CPU efficient (though incurs a higher I/O cost!).

In summary:

1. Joins on flash SSDs have a greater tendency to become CPU-bound (rather than I/O-bound), so ways to improve the CPU performance, such as better cache utilization, is of greater importance with flash SSDs.

2. Trading random reads for random writes is likely a good design choice for flash SSDs.

3. Compared to sequential writes, random writes produce more I/O variations with flash SSDs, which makes the join performance less predictable.

4.4 Effect of Varying the Page Size

In this experiment, we continue to use blocked I/O, but vary the page size from 2 KB to 32 KB. (The maximum page size allowed by SQLite3 is currently 32 KB.) The size of the I/O block is also calculated using the equations shown in Table 1, as in the previous section. The results are shown in Figure 2 for a 500 MB buffer pool.

	Magnetic HDD		Flash SSD	
Page Size	User	Kernel	User	Kernel
2 KB	187.5 sec	108.4 sec	204.4 sec	324.0 sec
4 KB	192.8 sec	49.0 sec	205.5 sec	157.9 sec
8 KB	190.5 sec	27.9 sec	183.6 sec	80.6 sec
16 KB	186.6 sec	15.5 sec	188.8 sec	39.8 sec
32 KB	187.2 sec	8.6 sec	185.9 sec	22.4 sec

Table 5: CPU times for block nested loops join on the magnetic HDD and the flash SSD

As can be seen from Figure 2, when blocked I/O is used, the page size has a small impact on the join performance in both the magnetic HDD and the flash SSD cases.

The key lesson here is that if blocked I/O is used, the database system can likely set the page size based on criteria other than join performance.

4.5 Effect of Blocked I/O

The major reason for using blocked I/O with magnetic HDDs is to amortize the high cost of disk seeks and rotational delays. An interesting question is: Does blocked I/O still make sense for flash SSDs? To answer this question, we repeated the experiment described in the previous section with page sized I/O instead of blocked I/O. These results are shown in Figure 3.

Comparing Figure 3 with Figure 2, we can clearly see that blocked I/O is still valuable for the flash SSD, often improving the performance by 2X. The reasons for this are: a) the software layer of the flash SSD still incur some latency [4], making larger I/O size attractive, and b) the write operations with larger I/O sizes generate fewer erase operations, as the software layer is able to manage a pool of pre-erased blocks more efficiently.

When the page size is 2 KB, the performance of sort-merge

join and Grace hash join on the flash SSD is worse than on the magnetic HDD. When the I/O size is less than the flash page size (4 KB), every write operation is likely to generate an erase operation, which severely degrades performance. This results re-confirms the observation (but for joins) that blocks should be aligned to flash pages [4].

We also observed that the CPU costs on the flash SSD and on the magnetic HDD are different for the same page size. CPU costs are generally larger with the flash SSD, due to the complex nature of FTL on the flash SSD. As described in Section 2, FTL not only provides the ability to map addresses, but also needs to support many other functionalities such as updating map structures, maintaining a pool of pre-erased blocks, wear-leveling and parallelism to improve performance. As an example, Table 5 shows CPU times of the block nested loops join, broken down by the time spent in the user and the kernel modes. (Other join methods showed similar behavior.) From this table, we observe that the kernel CPU times are larger with the flash SSD. This gap between the CPU costs for the flash SSD over the magnetic HDD is larger for smaller page sizes, since the smaller page sizes result in more I/O requests, which keeps the FTL busier with frequent direct map look-ups and indirect map updates. On the other hand, the gap is smaller with larger page sizes, and eventually the CPU costs for the SSD over the magnetic disk are almost the same when blocked I/O is used as in Figure 2.

In summary:

1. Using blocked I/O significantly improves the join performance on flash SSDs over magnetic HDDs.
2. The I/O size should be a multiple of the flash page size.

5. RELATED WORK

There is substantial related work on the use of flash memory for DBMSs. Graefe [6] revisits the famed five-minute rule based on flash and points out several potential uses in DBMSs. Lee *et al.* [12] suggests a log-based storage system that can convert random writes into sequential writes. In that paper they presents a new design for logging updates to the end of each flash erase blocks, rather than doing in-place updates to avoid expensive erase operations. Lee *et al.* [13] observe the characteristics of hash join and sort-merge join in a commercial system and conclude that for that system, sort-merge join is better suited for flash SSDs; however the implementation details of the hash join method in the commercial system is not know. Myer [14] examines join performance on flash SSDs under a set of realistic I/O workloads with Berkeley DB on a fixed-sized buffer pool. Of the hash-based join algorithms, only Grace hash join is considered in this study.

Shah *et al.* [16] show that for flash memory the PAX [3] layout of data pages is better than a row-based layout for scans. They also suggest a new join algorithm for flash SSDs. No direct implementation of the algorithm is presented, but the potential benefits of the new algorithm are presented by using an analytical model, and comparing it to Grace hash join. Tsirogiannis *et al.* [18] presents a new pipelined join algorithm in combination with the PAX layout. A key aspect of their new algorithm is to minimize I/Os by retrieving only required attributes as late as possible. They show that their algorithm is much more efficient on flash SSDs compared to hybrid hash join, especially when either few attributes or few rows are selected in the join result.

More recently, Bouganim *et al.* [4] have provided a collection of nine micro-benchmarks based on various I/O patterns to understand flash device performance.

6. CONCLUSIONS

In this paper, we have presented and discussed the performance characteristics of four well-known ad hoc join algorithms on a magnetic HDD and a flash SSD. Based on our evaluation, we conclude that a) buffer allocation strategy has a critical impact on the performance of join algorithms for both magnetic HDDs and flash SSDs, b) despite the absence of mechanically moving parts, blocked I/O plays an important role for flash SSDs, and c) both CPU times and I/O costs must be considered when comparing the performance of join algorithms as the CPU times can be a larger (and sometimes dominating) proportion of the overall join cost with flash SSDs. Many of these observations are lessons that we have learnt from previous work on optimizing join algorithms for magnetic HDDs, and continue to be important when studying the performance of join algorithms on flash SSDs, though with different emphases.

For future work, we plan to expand the range of hardware that we consider in this study, including using enterprise flash SSDs and other magnetic disk-based I/O configurations. We also plan on deriving detailed analytical models for existing join algorithms on flash SSDs and exploring if the optimal buffer allocations differ from that for magnetic HDDs.

7. ACKNOWLEDGEMENT

We thank Jeff Naughton for providing useful feedback on various parts of this work. This research was supported by a grant from Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Microsoft.

8. REFERENCES

[1] SQLite3. *http://www.sqlite.org/*.
[2] Transaction Processing Performance Council. *http://www.tpc.org/*.
[3] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 169–180, 2001.
[4] L. Bouganim, B. Jonsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2009.
[5] K. Bratbergsengen. Hashing Methods and Relational Algebra Operations. In *proceedings of the 10th International Conference on Very Large Data Bases (VLDB)*, pages 323–333, 1984.
[6] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *proceedings of the 3rd International Workshop on Data Management on New Hardware (DaMoN)*, 2007.

[7] J. Gray and B. Fizgerald. Flash Disk Opportunity for Server-Applications. *ACM QUEUE*, 6(4):18–23, July 2008.

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

[9] C. Hwang. Nanotechnology Enables a New Memory Growth Model. *Proceedings of the IEEE*, 91(11):1765–1771, November 2003.

[10] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of Hash to Database Machine and Its Architecture. *New Generation Computing*, 1(1):63–74, 1983.

[11] D. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass, 1973.

[12] S. Lee and B. Moon. Design of Flash-Based DBMS: An In-Page Logging Approach. In *proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 55–66, 2007.

[13] S. Lee, B. Moon, C. Park, J. Kim, and S. Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1075–1086, 2008.

[14] D. Myers. On the Use of NAND Flash Memory in High-Performance Relational Databases. *Master's Thesis, MIT*, 2008.

[15] M. Polte, J. Simsa, and G. Gibson. Comparing Performance of Solid State Devices and Mechanical Disks. In *proceedings of the 3rd Petascale Data Storage Workshop (PDS Workshop)*, 2008.

[16] M. Shah, S. Harizopoulos, J. Wiener, and G. Graefe. Fast Scans and Joins using Flash Drives. In *proceedings of the 4th International Workshop on Data Management on New Hardware (DaMoN)*, 2008.

[17] L. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[18] D. Tsirogiannis, S. Harizopoulos, M. Shah, J. Wiener, and G. Graefe. Query Processing Techniques for Solid State Drives. In *proceedings of the ACM SIGMOD International Conference on Management of Data*, 2009.