Optimizing Index Scans on Flash Memory SSDs

Eun-Mi Lee† Sang-Won Lee† Sangwon Park‡

†School of Information & Communications Engineering
Sungkyunkwan University
Suwon, 440-746, Korea
{bbhammer,swlee}@skku.edu

‡Department of Information Communication Engineering
Hankook University of Foreign Studies
Yongin, 449-791, Korea
{swpark}@hufs.ac.kr

ABSTRACT

Unlike harddisks, flash memory SSDs have very fast latency in random reads and thus the relative bandwidth gap between sequential and random read is quite small, though not negligible. For this reason, it has been believed that index scan would become more attractive access method in flash memory storage devices. In reality, however, the existing index scan can outperform the full table scan only in very selective predicates.

In this paper, we investigate how to optimize the index scan on flash memory SSDs. First, we empirically show that the index scan underperforms the full table scan even when the selectivity of selection predicate is less than 5% and explain its reason. Second, we revisit the idea of sorted index scan and demonstrate that it can outperform the full table scan even when the selectivity is larger than 30%. However, one drawback of the sorted index scan is that it loses the sortedness of the retrieved records. Third, in order to efficiently resort the result from the sorted index scan, we propose a new external index-based sort algorithm, *partitioned sort*, which exploits the information of key value distribution in the index leaf nodes. It can sort data in one pass regardless of the available sort memory size.

Keywords-flash memory SSDs; sorted index scan; partitioned sort

1. INTRODUCTION

Magnetic harddisks have dominated the storage market for more than three decades, and most enterprise database systems assume harddisks as secondary storage. By the way, harddisk has the inevitable mechanical latency, and the access time for a small data page (e.g. 4KB) is thus dominated by the latency. In contrast, the bandwidth of sequential access in contemporary enterprise class harddisks is enormous (e.g. 200MB per second). From the historical perspective, we have witnessed that the improvement of the access latency in harddisk is lagging far behind that of the sequential access, and thus the performance gap between random access and sequential access is widening [3]. This bandwidth imbalance between random and sequential accesses in harddisk had made query optimizers prefer the full table scan to the index scan, except only when the selectivity of the given selection predicate is very low (e.g. 1%).

Meanwhile, flash memory based solid stated drives (hereafter, SSDs) are becoming popular as alternative storage to harddisks. In contrast to harddisk, SSD has very fast latency in random IOs since it has no mechanical part, and thus the performance gap between sequential and random access to all the data pages in a table is very small, though not negligible. Therefore, we can anticipate that, on top of SSDs, the index scan can outperform the full table scan even when the ratio of randomly accessed data pages in a table is very high (e.g. more than 50%). In reality, however, the existing index scan can outperform the full table scan only in very selective predicates.

In this paper, we investigate how to optimize the index scan on flash memory SSDs. First, we empirically demonstrate that the index scan underperforms the full table scan even when the selectivity of selection predicate is less than 10%, and explain that it is mainly due to repetitive reads of same data pages during the index scan with limited size of buffer cache. Second, we revisit the idea of sorted index scan which first sorts the index entries in the order of record identifier before fetching data pages and thus can avoid the repetitive reads of the same pages. Our experimental result shows that it can outperform the full table scan when the selectivity is larger than 30%. However, one drawback of the sorted index scan is that the retrieved records is not sorted any more in the index key order. Third, in order to efficiently re-sort the result from the sorted index scan, we propose a new index-based sort algorithm, *partitioned sort*, which exploits the information of key value distribution in the index leaf nodes while partitioning the data to be sorted. It can sort data in one pass regardless of the available sort memory size. In contrast, the traditional external sort might require multiple passes depending on the data size and the sort memory size.

2. BACKGROUND

In this section, we compare the performance characteristics of harddisks and SSDs according to the access pattern to data, then explain the parallelism adopted by contemporary SSDs and its implications on the performance of access methods, and briefly review two popular access methods, the full table scan and the index scan.

2.1 Harddisks and SSDs

Table 1: Harddisk vs. Flash SSD

Media	Harddisk†	Flash SSD‡
Latency (4KB)	3.5ms	0.2ms
IOPS (4KB)	600	35,000
Sequential bandwidth	125MB/sec	200MB/sec
Ratio between random and sequential bandwidth	0.02	0.7

†Seagate Cheetah 15K.5 ST373455SS
‡A Commercial SSD (SLC flash chip)

Table 1 presents the key performance metrics of a harddisk and an SSD used in our experiment. Because this paper focuses on read-intensive queries, Table 1 contains only the read-related performance metrics. The first row represents the access latency (i.e. response time) when a single process reads a page of 4KB. The second row shows maximum IOPS (IO per second) when concurrent read requests are made to each storage media. In other words, the second row represents the random read bandwidth of each storage media. Please note that throughput is not inverse of response time because each product has the native command queuing (NCQ) feature which can handle multiple requests at a time [3]. To be concrete, harddisks use the well-known elevator algorithm while SSDs distribute multiple requests to different flash chips so that each flash chip can handle each request in parallel. The third row represents the maximum bandwidth of each product when we read data in a purely sequential way. The fourth row shows the ratio between random and sequential bandwidth in each product. The ratio is calculated using the formula, *(maximum IOPS x 4KB) / sequential bandwidth*. Please note that while there is huge imbalance between sequential and random access in harddisk, the relative bandwidth gap between sequential and random read is quite small, though not negligible.

2.2 Parallelism inside SSDs

In order to achieve high bandwidth and better IOPS, every modern SSD adopts multi-channel and multi-way architecture and flash memory controller can read and write flash chips in parallel [1]. From the perspective of query throughput, we need to understand the impact of this parallelism inside SSDs. For simplicity, let us assume that the database tables are uniformly striped across multiple flash memory chips, and also that the data pages from a table are also evenly distributed over the chips. In fact, this assumption about the data distribution across multiple flash chips is similar to the RAID-0 striping. When a single query is accessing a table using the full table scan, all the flash chips would become active because each flash chip contains some data pages from the table, thus maximizing the bandwidth. In contrast, when a single query is accessing the table using the index scan which is implemented using *synchronous IO* call in most DBMS, only one of the flash chips is active while the others are idle at a point of time. Namely, the parallelism inside SSDs is under-utilized.

In order to fairly compare the performance of the full table scan and the index scan in SSDs, we need to run multiple queries at least as many as the parallelism degree inside SSDs, that is, the number of channels. We will illustrate the effect of parallelism in SSDs on query processing in Section 3.2.

2.3 Full Table Scan and Index Scan

Most DBMS provides two primitive access methods to tables, the full table scan and the index scan (hereafter, FTS and IDX, respectively) [5]. When harddisk is used as the database storage, the database query optimizer prefers FTS to IDX as the access method, except when the selection predicate for the target table is very selective. This is mainly due to the slow random IO latency, and IDX would, taking into account the widening performance gap between the sequential and random access, remain as the access method only when accessing a few records. In contrast, SSDs do not have any mechanical component and thus the access time in SSDs is nearly proportional to the data size being accessed. Therefore, an index based access method is expected to be promising when SSDs are used as database storage.

Despite using SSDs as database storage, however, there are two issues which might limit the efficacy of the traditional IDX: *index clusteredness* and *buffer size*. According to the clusteredness, there are clustered index and non-clustered index. If the physical ordering of data records of a table is the same or very close to the ordering of data entries in an index created on the table, the index is said to be clustered; otherwise non-clustered [5]. In case of clustered index, IDX would be very efficient because every data pages is read into buffer only once during index scan. In case of non-clustered index, IDX might read the same data pages from the storage into buffer cache several times because the size of buffer cache is limited and the data page can contain several data records to be retrieved by IDX. In fact, as we will illustrate in Section 3.2, IDX underperforms FTS even when the predicate selectivity is below 5% on SSDs.

3. SORTED INDEX SCAN

3.1 Sorted Index Scan

To improve the efficiency of the non-clustered index scan, there have been some approaches to sort the record identifiers (*rid*) in index entries in the order of page id before accessing the data records using the rids [2, 7]. By taking the *sorted index scan* with a non-clustered index, each data page is fetched once from the storage into buffer even with the limited buffer size. In harddisk based databases, however, the sorted index scan (hereafter, SIDX) has not been such popular and will become less effective because of the ever widening imbalance between sequential and random access. In case of SSDs, however, SIDX could be much more effective. For example, as will be shown later, SIDX outperforms FTS in the selectivity of 40% while pure IDX outperforms FTS in the selectivity of 5%. Therefore, we can make use of non-clustered indexes in a wider range of predicate selectivity.

The commercial database server used in our experiment does not directly support SIDX as its access method, and thus, in order to simulate the logical steps in SIDX

using the DBMS, we used a query transformation approach. The two SQL queries in Figure 1 illustrate our query transformation approach using a sample range query which retrieves and aggregates tuples from table `tab`, each of whose `a` column value is between `min` and `max`. In the example, we assume that a non-clustered index exists on column `a` in table `tab`.

```
/* Before transformation */
SELECT *
FROM   tab
WHERE  a BETWEEN min AND max;

/* After transformation */
SELECT *
FROM  (SELECT /* use_nested_loop */ t1.*
       FROM (SELECT /* use_index */ rowid
             FROM   tab
             WHERE  a BETWEEN min AND max
             ORDER BY rowid) t1, tab t2
       WHERE t1.rowid = t2.rowid );
```

Figure 1: Query transformation for simulating the sorted index scan

The innermost `SELECT` statement in the transformed query finds all the index entries satisfying the predicate `a BETWEEN min AND max`, sorts them, and returns the result as a virtual table to the next outer query. The second inner `SELECT` statement accesses all the relevant data pages in table `tab` in the order of their page_ids, and returns the records satisfying the given predicate. And the outermost statement aggregates `b` values of all the records. In fact, this approach of query transformation will have run-time overhead over the natively implemented SIDX, but the overhead is not such big enough to change the main argument made in this paper.

3.2 Performance Evaluation

For the experiment, we used a commercial DBMS on Linux 2.6.18 with AMD 3.0GHz hexa-core processor and 8GB RAM. As data tablespace storage, we used the harddisk and the SSD from Table 1. In order to hide the interference on the data tablespace by the IOs in temporary tablespace, another harddisk was designated as the temporary tablespace device. We set database block size, buffer cache, and sort memory to 8KB, 256MB, and 1MB, respectively. The sort memory represents the memory area designated to each process for sorting. When either harddisk or SSD was used as stable storage, it was bound as a raw device to minimize the interference from data caching by the file system.

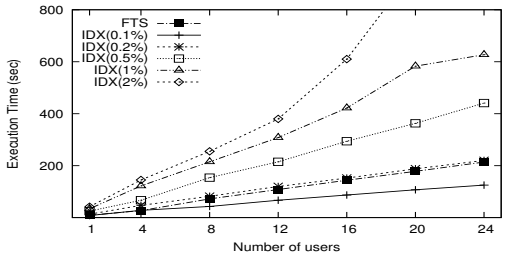
The sample table has 2.5 million tuples, and each tuple is 300B long. In order to create a non-clustered index on the table, we assigned a randomly generated distinct integer value between 1 and 2,500,000 to column `a` of each tuple, and created an index on column `a`.

For the sample range query in Figure 1 against the sample table and the non-clustered index, we measured the query execution time of three access methods, FTS, IDX, and SIDX, using harddisk and SSD, respectively, by varying the query selectivity and the number of concurrent users. When varying the number of concurrent users, we made 24 copies of both the sample table and the non-clustered index on column `a` and allowed each session to run the same sample query in Figure 1 against its own separate table and index. And, the performance result is plotted in Figure 2.

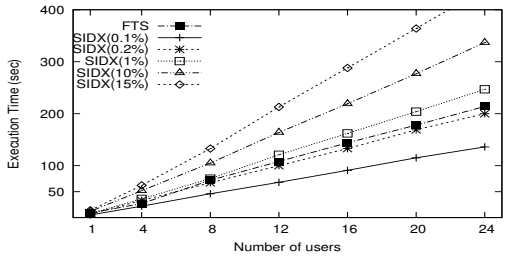
First, Figure 2(a) compares the performance of FTS and IDX in harddisk. As expected, FTS always outperforms IDX, except when the query predicate is very selective (less than 0.2%). This is mainly because the number of random reads increases as the query predicate becomes less selective and because the performance gap between sequential and random read operations in harddisk, as shown in Table 1, is large. Next, Figure 2(b) compares the performance of SIDX and FTS in harddisk. By comparing the performance of IDX and SIDX in Figure 2(a) and Figure 2(b), respectively, we know that SIDX improves IDX considerably. And this is mainly because SIDX reads each page to be accessed by queries only once while IDX might read each page several times. Despite the performance benefit of SIDX, we should note that SIDX, in case of harddisks, underperforms FTS except for very selective case (e.g. less than 1%) at which 6% of total pages in the table is accessed.

Now, let us compare the performance of three access methods on SSD. Figure 2(c) compares the performance of FTS and IDX on SSD. From Figure 2(c), we know that the break-even point between FTS and IDX in SSD moves to the selectivity of 5% at which 17% of total blocks in the table is accessed. Nevertheless, it is still disappointing to observe that the repetitive reads of the same pages make IDX to underperform FTS in SSD even when the selectivity is below 10%. In contrast, the performance of SIDX on SSD can outperform FTS for a wider range of selectivity. As plotted in Figure 2(d), the break-even point between SIDX and FTS moves up to the 40% selectivity when sixteen concurrent queries are running. In our sample table and non-clustered index, the number of distinct pages to be accessed by SIDX is about 60% of total page number in the table.

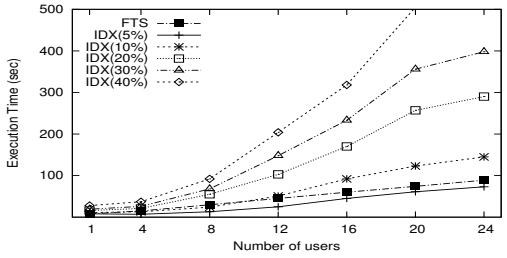
One interesting observation in Figure 2(d) is that the break-even selectivity between FTS and SIDX goes higher as the number of concurrent users increases. In order to understand this performance phenomenon in Figure 2(d), we need to remind the effect of parallelism inside SSD already explained in Section 2.2. When a single query runs in SIDX, SSD is under-utilized since only one flash chip is active at any point of time due to the synchronous IO call in SIDX. In contrast, in case of FTS, even if a single query is running, almost flash chips would be active because of the sequential access pattern in FTS. Therefore, when a single query runs, SIDX could underperform FTS even at the selectivity of 5%. Meanwhile, as more concurrent queries are running in SIDX, more flash chips inside SSD would be active because multiple random read operations are issued by concurrent queries, and thus the query throughput of SSDs gets higher. In contrast, in case of FTS, because of its sequential access pattern, the query execution time would increase in proportion to the number of concurrent queries. Thus, SIDX outperforms FTS at the selectivity of 40% when sixteen or more concurrent queries are running, at which case all flash chips inside SSD would be active in SIDX.



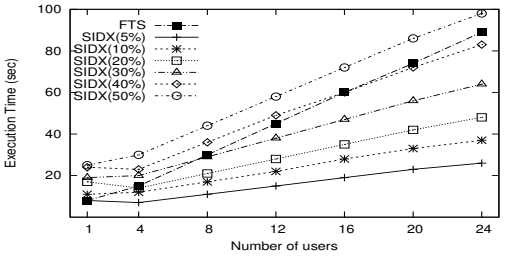
(a) Index Scan on HDD



(b) Sorted Index Scan on HDD



(c) Index Scan on SSD



(d) Sorted Index Scan on SSD

Figure 2: The break-even point between FTS, IDX and SIDX

4. INDEX-BASED PARTITIONED SORT

In the previous section, we have shown that SIDX is an effective access method on SSD. One downside of SIDX is, however, that the tuples produced by SIDX is not any more in the order of index key. Because the *interesting order* is not preserved [6], the output tuples from SIDX should be re-sorted to be used for `order by` or `group by` clause or as an input to the sort merge join.

In this case, the sorting overhead would be non-trivial, although it does not diminish the benefit of SIDX as an access method. But, fortunately we can efficiently sort the output of SIDX by exploiting the index itself. For this, we propose a new index-based sort algorithm, *partitioned sort (PS)* in this section.

4.1 Partitioned Sort

The key idea of PS is to map the records evenly into small partitions based on the ranges of values of the partitioning key so that all the records in each partition could fit into the available sort memory. Because each partition can fit in the sort memory, we can sort each partition using in-memory sort algorithm after reading it from SSD. And because all the partitions are range partitioned, we can simply concatenate the sorted output of each partition without any separate merge step such as in the existing external merge sort.

Then, the remaining key question is how to calculate the partitioning key values. In SIDX, we can calculate them just by scanning the relevant index entries in the leaf nodes because the index entries in the B+-tree leaf nodes are already sorted in key order. Meanwhile, without index on the key attribute, the only way to calculate the partitioning key values is, as far as we can imagine, to sort the whole records.

Below is listed the algorithm of our partitioned sort, and its overall process is depicted in Figure 3.

1. While scanning the index entries in SIDX, we calculate the partitioning key values so that the whole records of each partition can fit into the available sort memory. At the end of this step, we know how many partitions are necessary and the value range of each partition.
2. We create the partitions as necessary and specify the value range of each partition.
3. While retrieving the records using SIDX, we insert each record to its corresponding partition (*partitioning phase*).
4. For each partition (in the ascending order of partition range), we read all the records into the sort memory, sort them, and output the sorted records. (*sorting phase*).

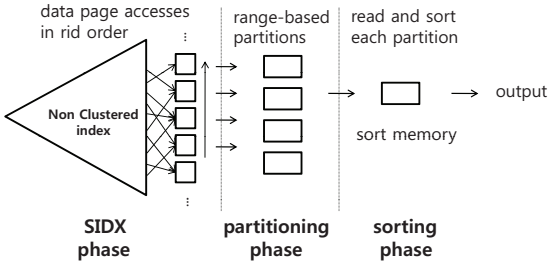


Figure 3: SIDX and partitioned sort

Please note that the step 3 writes all the records once and the step 4 reads all the records once, and thus our PS can, *regardless of the size of the available sort memory*, sort the retrieved records in *one pass* (i.e. one write

and one read). Also note that this one-pass sort is possible because the partitions generated in step 3 is *skewless*, that is, the size of each partition is uniform. In comparison, the existing external merge sort [5] may require multiple passes depending on the size of the sort memory, especially when the sort memory is small compared to the size of records to be sorted.

Now let us illustrate the IO pattern generated from our PS with SIDX, and compare it with the IO pattern from the traditional external sorting (ES) with the full table scan (FTS). By the way, the commercial DBMS used in our experiment does not support either SIDX or PS, we again take the query transformation approach to simulate SIDX and PS, as shown in Figure 4. The first `CREATE` statement in Figure 4 creates range-based partitions and inserts the retrieved records using the transformed `SELECT` statement simulating SIDX in Figure 1. In other words, it represents the partitioning phase of our PS algorithm while retrieving the records using SIDX. The second `SELECT` statement in Figure 4 corresponds to the sorting phase of our PS algorithm. That is, it reads each partition, sorts its records using the `ORDER BY` clause, and then merges all the sorted records using the `UNION ALL` clause.

```
/* Before transformation */
SELECT *
FROM   tab
WHERE  a BETWEEN min AND max
ORDER BY a;

/* After transformation */
CREATE TABLE partioned PARTITION BY RANGE (a) (
  PARTITION p_1 VALUES LESS THAN (val1),
            ...
  PARTITION p_n VALUES LESS THAN (maxvalue)
) AS
SELECT statement in Figure 2;

SELECT * FROM ( /*+ use_no_merge */
  SELECT * FROM part_tab PARTITION (p_1) ORDER BY a
  UNION ALL
            ...
  SELECT * FROM part_tab PARTITION (p_n) ORDER BY a)
```

Figure 4: Query transformation for simulating partitioned sort

In order to validate that the query transformation in Figure 4 works as we intended, we traced the IO pattern while executing the transformed query and plotted them in Figure 5(a) [4]. The selectivity of the query was 30% and the size of the sort memory was set to 2 MB. A clear separation of two phases was observed in Figure 5(a). When partitions are created and populated during the partitioning phase, the data pages for the partitions were written sequentially to the data tablespace. In the second sorting phase, on the other hand, the data pages of each partition were read in randomly scattered manner, leading to random reads spread over the whole region of the time-address space corresponding to each partition. Figure 5(a) confirms that the PS algorithm, as we intended, can sort the records in one pass. To better explain the difference between PS and ES, we also traced the IO operations in the temporary

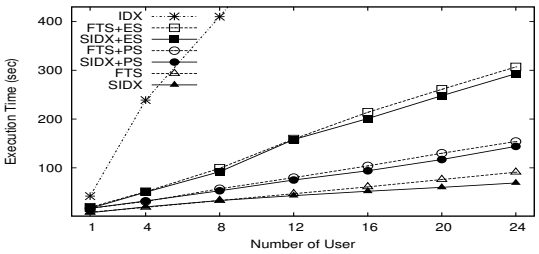


Figure 6: Performance of various query plans

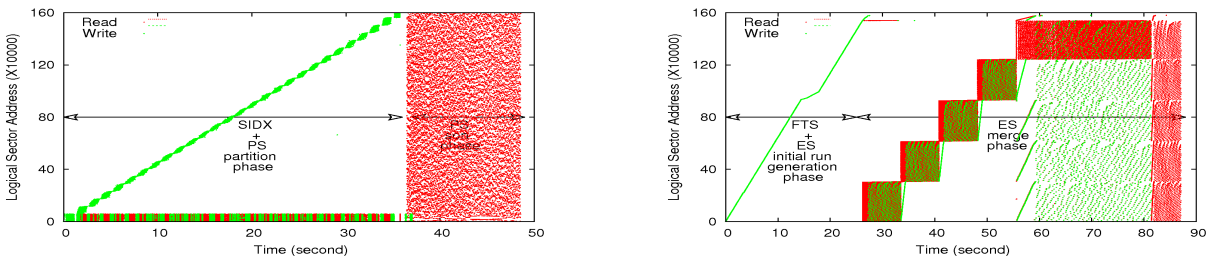
tablespace while executing the original `ORDER BY` query before the transformation in Figure 4 and plotted them in Figure 5(b). The table was accessed in FTS and the selected records were sorted using ES provided by the commercial DBMS. Figure 5(b) also has two phases. The first phase generates the initial runs while scanning the table from the data tablespace, and the second phase merges the runs to generate the final sorted output. In the merge phase, multiple passes are required to merge the runs, and this is because the sort memory is too small to merge all the runs at once. From this, we can observe that our PS can sort the data just in one pass regardless of the size of the available sort memory while ES may require multiple passes depending on the size of sort memory and the data size to be sorted.

The IO pattern of PS consists of sequential writes and random reads, which is preferable in SSD [4], and thus PS can provide good performance. In case of harddisk, however, as we confirmed throughout a separate set of experiments using the harddisk, the small random reads in the sort phase makes PS underperform ES.

One interesting point is that our index-based partitioned sort can be used in combination with FTS as well as with SIDX. Thus, when the selectivity is higher (e.g. larger than 40%) and an index is available on the sorting attribute(s), PS combined with FTS (FTS + PS) can outperform the traditional ES combined with FTS (FTS + ES).

4.2 Performance Evaluation

The experimental setting here is same as that in Section 3.2. In order to compare the performance of various query execution plans for the `ORDER BY` query in Figure 4, we ran the query in five different ways, including IDX, SIDX+PS, SIDX+ES, FTS+PS and FTS+ES. The idea of PS can be, as stated earlier, applied to FTS as well as SIDX. In this experiment, the query selectivity was set to 30%, the size of the sort memory was 32 MB, the buffer caches was 128MB, and the number of concurrent users was varied from 1 to 24. The performance result is presented in Figure 6. In Figure 6, we also included the performance of SIDX and FTS for the query without `order by` clause so as to easily estimate the sort overhead in each query plan. From Figure 6, we can make a few important observations. First, although IDX preserves the sortedness of the result records and thus the separate sorting phase is not required, it performs much worse than the other four plans. Its overhead from the repetitive reads for the same pages is too



(a) SIDX+PS (b) FTS+ES

Figure 5: IO patterns of PS and ES

large to be compensated by avoiding the explicit sorting step for the `order by` clause. The second observation is that our PS algorithms (SIDX+PS and FTS+PS) outperform the existing ES algorithms (SIDX+ES and FTS+ES). In particular, taking only the sort time into account, it can be roughly said that PS can sort the records two or three times faster than ES.

In order to test whether PS algorithm is effective in harddisk, we carried out a separate set of experiments using harddisk, but FTS + ES outperforms both SIDX + PS and FTS + PS consistently over almost every experimental configuration except only when the selectivity is lower than than 1%. This is because the overhead of frequent mechanical harddisk head movements for the random writes to multiple partitions during the partitioning phase in PS algorithm can not be compensated by the one pass sort.

Table 2: The effect of sort memory size

Access	2MB	4MB	8MB	16MB	32MB
FTS+ES	340	274	269	269	263
SIDX+PS	122	124	119	122	121
Ratio	2.8	2.2	2.3	2.2	2.2

Now, let us explain the effect of the sort memory size on the performance of two query plans, SIDX+PS and FTS+ES. We ran the transformed query in Figure 4 by varying the size of the sort memory from 2, 4, 8, 16 to 32 MB with the selectivity of 30% with 20 concurrent users. For comparison, we also ran the query in FTS+ES mode with the same configuration. The performance result is presented in Table 2. As expected, while the performance of FTS+ES varies considerably depending on the sort memory size, the performance of SIDX+PS is almost consistent regardless of the sort memory size.

5. CONCLUSION

In this paper, we revisited SIDX (sorted index scan) as a database access method when SSDs are used as the database storage, and have empirically shown that the break-even point between SIDX and FTS (full table scan) moves to higher selectivity (e.g. 40%) especially when multiple queries are concurrently running and thus the intrinsic parallelism inside SSDs are exploited. And, in order to efficiently re-sort the unsorted result of SIDX, we proposed an index-based partitioned sort algorithm, PS, and have experimentally shown that it can outperform the existing external merge sort (ES) and in particular its performance is, in contrast to ES, not sensitive to the size of available sort memory.

In this paper, we simulated SIDX and PS by modifying queries in a commercial DBMS because it does not support those operators directly. In fact, as far as we know, any commercial or open source DBMS is not equipped with either algorithm. However, considering that both SIDX and PS are effective in SSDs and that SSDs would be more actively adopted in the database market, both algorithms need to be incorporated into future DBMSs as first-class citizens; they should be implemented as built-in operators and could be chosen by query optimizers in a cost-based manner.

ACKNOWLEDGMENTS

This work was supported by ITRC NIPA-2011-(C1090-1121-0008) and NRF Grant NRF-2011-0026492.

6. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference*, 2008.

[2] J. M. Cheng, D. J. Haderle, R. Hedges, B. R. Iyer, T. Messinger, C. Mohan, and Y. Wang. An Efficient Hybrid Join Algorithm: A DB2 Prototype. In *Proceedings of ICDE*, pages 171–180, 1991.

[3] S.-W. Lee, B. Moon, and C. Park. Advances in Flash Memory SSD Technology for Enterprise Database Applications. In *Proceedings of SIGMOD*, pages 863–870, 2009.

[4] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *Proceedings of SIGMOD*, pages 1075–1086, 2008.

[5] R. Ramakrishnan and J. Gehrke. *Database Management Systems(3rd ed.)*. McGraw Hill, 2002.

[6] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of SIGMOD*, pages 23–34, 1979.

[7] P. Valduriez. Join Indices. In *ACM Transactions on Database Systems*, pages 218–246, 1987.