Flash-aware Index Scan in PostgreSQL

Da-som Hwang[1], Woon-hak Kang[2], Gihwan Oh[3], Sang-won Lee[4]
College of Info. And Comm. Engineering
Sungkyunkwan University
Suwon, Korea
[1] rhcqnssp32@skku.edu
[2] woonagi319@skku.edu
[3] wurikiji@skku.edu
[4] swlee@skku.edu

***Abstract*— Recently, a trend of storage markets has changed from hard disks (HDD) that have dominated the markets for the last several decades to flash based solid state disks (SSD). Corresponding to the drift, various studies have been conducted to adapt traditional database systems (DBMS) to storage devices based SSD. However, most DBMSs are still more HDD-friendly.**

HDD and SSD have inherent features because of their own architecture designs. HDD has a wide gap between performance of sequential I/O and that of random I/O owing to its mechanical part. Due to the fact, DBMS usually prefers a full table scan to the index scan except when a selectivity is enough low to take an advantage of the index scan. Unlike HDD, SSD without any mechanical parts has a tiny gap between performance of sequential I/O and that of random I/O so that the characteristic of SSD allows DBMSs to efficiently access to a storage based SSD with the index scan. Another feature of SSD is an internal parallelism from its internal architecture. In spite of a circumstance with the secondary storage based SSD, DBMSs are more likely to choose the full table scan rather than the index scan for I/O operations. It is necessary to understand distinct properties and differences of two storage devices and make the index scan SSD-friendly to improve its performance.

In this paper, we simulate an optimization of the index scan, flash-aware index scan by combining two concepts; sorted index scan that scans tuples in order of record ids and parallel synchronous I/O that is a traditional synchronous I/O with an array of I/O requests per operation. We implement them to simulate the flash-aware index scan in PostgreSQL to make the system be aware of SSD nature and enhance the performance of the index scan.

Keywords—flash based SSD; sorted index scan; parallel synchronous I/O;

I. INTRODUCTION

Hard disks (HDD) have monopolized storage markets for several decades. Most commercial database management systems (DBMS) have developed to improve its performance with HDD based storages. Flash based solid state disks (SSD) are considered as a substitute of HDD as storage vendors supply cheap and high-capacity SSDs in the last decade. Corresponding the trend, various studies have been conducted to adapt traditional database systems (DBMS) to storage devices based SSD. However, most DBMSs are still more HDD-friendly.

HDD and SSD have inherent features because of their own architecture designs. Mechanical parts in HDD, which is an outstanding feature of HDD, cause high response time. In addition, a performance gap between sequential I/O and random I/O is big owing to the physical parts. Due to the fact, DBMS usually prefers a full table scan to the index scan except when a selectivity is low enough to take an advantage of the index scan.

Unlike HDD, SSD doesn't have any mechanical parts. Taking an advantage of such a characteristic, SSD has a tiny gap between performance of sequential I/O and that of random I/O. This allows DBMSs to efficiently access to a storage based SSD with the index scan. Another feature of SSD is an internal parallelism from its internal architecture. It provides a performance improvement when DBMS exploits the parallelism of SSD with proper ways such as multiple I/O requests [1].

In spite of a circumstance with the secondary storage based SSD, DBMSs are more likely to choose the full table scan rather than the index scan for I/O operations. It is necessary to understand distinct properties and differences of two storage devices and make the index scan SSD-friendly to improve its performance.

In this paper, we simulate an approach that the index scan can be flash-aware combining two concepts: sorted index scan and parallel synchronous I/O. Sorted index scan fetches tuples in an order of record identifiers [2]–[4]. It prevents us from reading the same pages repetitively. Parallel synchronous I/O is an idea introduced in [5] first. It works like a traditional synchronous I/O not with a single I/O request but with an array of I/O requests per operation. Also we implement the flash-aware index scan in PostgreSQL to let the database management system be aware of SSD nature and enhance the performance of the index scan.

This paper is organized as follows. Section 2 explains key concepts covering sorted index scan and parallel synchronous I/O in details. Section 3 describes the flash-aware index scan with combination of the two idea and how we implement the flash-aware index scan in an open source based DBMS,

PostgreSQL. We present experimental results in Section 4. Section 5 concludes the paper.

II. BACKGROUND

In this section, we give full details of a few concepts of the flash-aware index including sorted index scan and parallel synchronous I/O as we mentioned in the introduction section. With the details, we also explain why each concept has good influence on performance of the index scan with SSDs.

A. Index clusteredness

Index clusteredness is one of factors that has an impact on a performance of the index scan. There are two types of index when it comes to its physical arrangement on a disk: clustered index and nonclustered index. It is a clustered index if data records of a table are settled on disks corresponding with the order of the data entries of an index on the table. Otherwise, it is a nonclustered index. If an index on a table has nonclustered index, then data pages are fetched randomly and even the same data page is more likely to be read several times with limited-size buffer when we access the table with the index scan. It makes the performance of the index scan worse. Even if a flash based SSD works well in a random access pattern, it has an imbalance between a performance of sequential I/O and that of random I/O. Thus, the performance of the index scan on SSDs still is affected by the clusteredness of an index. We can see how much it has an effect on a response time of the index scan in Section 4.

B. Sorted Index Scan

Sorted index scan is an approach to improve execution of the index scan with nonclustered index. It fetches records in the order of page identifiers by sorting record identifiers on index entries into page identifiers order before accessing data records. It allows us to sequentially read the data pages on disks once at most so that we avoid reading the same pages again that we have ever read. In other words, it decreases the total number of I/O requests for the index scan. Thus, we can obtain performance improvement since sequential reads always outperform random reads on hard disks as well as flash based SSDs.

There is a drawback of sorted index scan. We receive out-of-order records when we access a table with sorted index scan since they are read in the ordering of data pages on disks. However, this shortcoming can be overcome with sort algorithms in DBMS and we can show that sorted index scan with external sort outperforms traditional index scan. This is because an I/O cost is more expensive than cost of running a sort algorithm.

We adapt PostgreSQL terms for the paper. A tuple identifier (*tid*) in PostgreSQL is the same term as a record identifier(*rid*). In PostgreSQL, it provides tid scan that can work like sorted index scan but its implementation has a flaw. We modify tid scan algorithm in PostgreSQL to work in a way that we expect. We will introduce it in details in the next section. Furthermore, we need to transform a query to use tid scan because PostgreSQL exploits tid scan only in a specific query form.

Fig. 1. Query transformation for sorted index scan in PostgreSQL

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

/* After transformation */
SELECT   *
FROM     tab
WHERE    ctid = ANY (ARRAY (
                        SELECT      ctid
                        FROM        tab
WHERE    a BETWEEN min and max));
```

Fig. 1 describes the query transformation. We write the query referencing to [2]. The query returns tuples from table tab of which have column values in the given range, between min and max. In this example, there is a nonclustered index on column `a` of table tab. In order to execute the tid scan in PostgreSQL, we need to give PostgreSQL specific current tuple identifiers (*ctid*). The innermost SELECT statement is for gathering ctids as an array which is used for the tid scan in an range of column values between min and max. The array is sorted according to the ordering of data pages on disks. With collected ctids, we access data records and retrieve them.

C. Parallel Synchronous I/O (P-sync I/O)

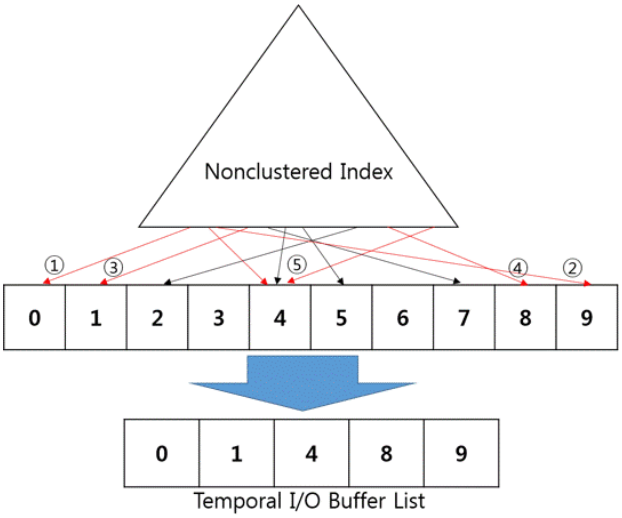
Roh et al. [1] has suggested a new I/O request method, parallel synchronous I/O (P-sync I/O) for the future OS kernel version. P-sync I/O performs still like a conventional synchronous I/O but it works with an I/O array as a unit of operation unlike the sync I/O operating with a single I/O request. P-sync I/O is derived from an attempt to utilize channel-level parallelism better. For taking advantages of channel-level parallelism, many I/O requests are delivered together or in a short interval among requests since a queue span of a queue processing I/O in SSDs is very short.

It has indicated three requirements of P-sync I/O as follow.

- Issue of P-sync I/O: It sends only a set of I/O requests to disks at a time and regains the results at once. In other words, the other sets of I/O requests should suspend until the previous one retrieves request results.
- Processing of P-sync I/O: An array of I/O requests is sent from a user space to a kernel space as a group and each requests in the array should wait until all of them arrives in the kernel space.
- Completion of P-sync I/O: The process is blocked until the I/O request lists are totally dealt with so that it does not consider how to handle the I/O completion events.

It says any I/O processing methods that assure the requirements do not exist. With Linux-native asynchronous I/O API, it emulated P-sync I/O but its implementation did not perfectly satisfy the second requirement. We emulate with the API in PostgreSQL and we will specifically explain how we implement in Section 3.

Fig. 2. Flash-aware Index Scan



Furthermore, it lists three algorithm design principles including principles from previous studies [6], [7] as well as its own experimental results.

- Request I/Os with large granularity to exploits package-level parallelism.
- Create an array of I/Os in order to utilize the parallelism of SSD. Consider using P-sync I/O first in order to issue the array in a single process and save parallel processing for later use in more suitable applications.
- Keep away from making mingled read and write I/O pattern.

III. FLASH-AWARE INDEX SCAN

We implement a new index scan to be aware of a flash based SSD with sorted index scan and P-sync I/O on PostgreSQL and explain how to implement the flash-aware index scan in details.

A. *Flash-aware Index Scan*

We apply sorted index scan and P-sync I/O to a traditional index scan to take advantages of internal parallelism of flash based SSDs at most. Fig. 2 describes how the flash-aware index scan works. For this example, a P-sync factor is five. It means we can send and receive maximum five I/O requests at once. A database system request 5 tuples and the requesting order is the same number with circular numbers. With sorted index scan, first, it makes a temporal list of requested tids by sorting them in an order of data pages and it sequentially accesses data pages in order of tids in the sorted list. With P-sync, the buffer manager in PostgreSQL puts the requested pages in a temporal I/O buffer list and returns the list when it receives five I/O results. Thus, a list, <0, 1, 2, 8, 9 > is returned rather than <0, 9, 1, 8, 4>.

B. *Implementation in PostgreSQL*

1) Sorted Index Scan: We exploit tid scan in PostgreSQL as sorted index scan but we modify several part of the algorithm because of its implementation flaw. Tid scan is composed of two phases like sorted index scan. The first phase is to obtain tuple ids and list them in an order of disk page ids. The second phase is to fetch the data records in sorted tuple ids order. However, while it collects tuple ids in the first stage, tid scan fetches useless tuples one by one from disk pages by calling a function, *index_fetch_heap()*. It just utilizes the normal index scan algorithm that fetches real data records when creating an array of tuple ids for tid scan although we can obtain tuple ids without fetcing data records. This causes extra read requests so that tid scan performs even worse than the index scan does. We simply change the tid scan's algorithm flow to get rid of unnecessary reads. Fig. 3 shows the original flow of tid scan and Fig. 4 represents that of tid scan without the defect that we mentioned. These flow charts do not show exact details of the whole tid scan but they give simple illustration of creating a tid list. To distinguish between the flow for normal index scan and that for accumulating tuple ids for tid scan, we define index related functions similar with existing functions for the index scan in PostgreSQL. The red box is for creating a temporal array of tuple ids for executing the tid scan. As we describe in Fig. 3, while collecting tuple ids, the original tid scan reads real tuples that just are abandoned because they are not needed in this step. It means the original tid scan in PostgreSQL reads the same data twice. It implies that the tid scan doesn't suit for the sorted index scan. Hence, we fixed the original tid scan not to fetch unnecessary tuples in the first procedure for sorted index scan. Fig. 4 shows the modified tid scan that works as we expected for sorted index scan by adding simple functions and changing slightly the tid scan flow.

Fig. 3. Flowchart of the original tid scan

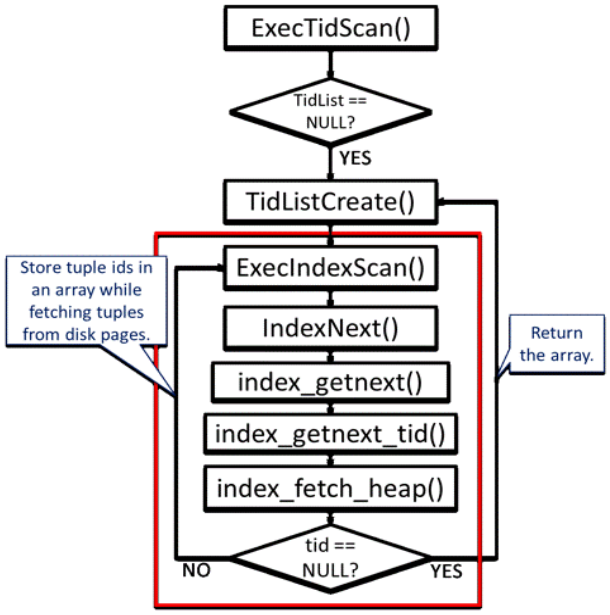
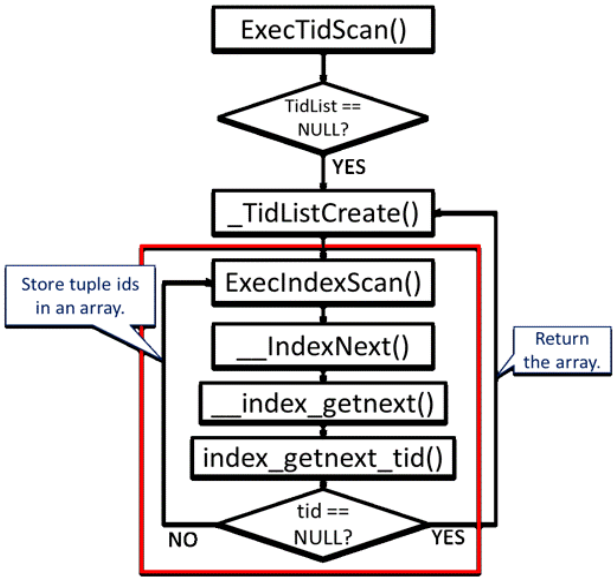


Fig. 4. Flowchart of modified tid scan



2) Parallel Synchronous I/O: We implement parallel synchronous I/O to deal with multiple I/O requests at once by using direct I/O and a library for asynchronous I/O, *Libaio*. For asynchronous I/O in PostgreSQL, we should use direct I/O and we add a direct I/O flag in proper positions such as when to open files and create new function pointers and functions working with libaio. Also, we add a global variable, *MAX_LIBAIO* as a factor to scale how many I/O requests are sent to disks and several functions mimicking the normal functions for reading a data page for a buffer in PostgreSQL but they process multiple buffers at a time. For example, two functions, *md_read_start()* and *md_read_insert()* are related to the first requirement of P-sync I/O in Section 2.C and *md_aio_read()* and *md_aid_end()* are associated with the second requirement and the last requirement respectively. Also, we add three new parameters, *Buffer *mbuf, int mbufNum, ItemPointerData *tidList. mbuf*, which is a *Buffer pointer* type instead of *Buffer* type, contains multiple buffer pages' information and *mbufNum* always has the value of MAX_LIBAIO and let functions know how many buffer pages will be read. Lastly, we hand over *tidList* that we obtain in the first step of tid scan through *ItemPointerData* pointer parameter in substitute of *BlockNumber* parameter in normal functions corresponding to proposed functions. *tidList* has all necessary location information of tuples that we are going to access and put the data pages from disks into *mbuf* in substitute of *BlockNumber* parameter in normal functions corresponding to proposed functions. *ReadBufferMultiple_common()* actually plays a role in handling buffer requests. The other functions are a kind of wrappers or simple callers. *ReadBufferMultiple_common()* is similar with *ReadBuffer_common()* except that it deals with multiple buffers.

IV. Experimental Result

We carry out experiments imitating the experiments in [2] and experimental settings as well as the sample table are based on [2].

A. Experimental background

For the experiments, we used a customized PostgreSQL-9.3 with Intel i5 3.40 GHz quad-core processor and 8 GB RAM. As data tablespace storage, we used Samsung SSD 840pro.

The sample table has 2.5 million tuples and its tuple is 300B long. Owing to build a nonclustered index on the table, we put randomly populated unique integer values between 1 and 2,500,000 to column a of each tuple and created an index on column a. We copy the sample table for each user when we execute queries with multiple users. We adjust a selectivity by a range of column a in where clause for each query Fig. 1.

The logical selectivity represents the ratio of the requested number of records to the total number of records. The physical selectivity means the amount of read data pages to the total data pages.

TABLE I. EXECUTION TIME OF ACCESS METHODS (SINGLE USER)

Access method	5%	10%	20%	30%	40%
IDX	10.35	72.24	145.72	217.98	291.76
SIDX	12.60	32.40	35.37	35.96	36.92
SIDX+Psync	2.21	17.32	19.36	19.93	20.69

TABLE II. EXECUTION TIME OF ACCESS METHODS (12 USER)

Access method	5%	10%	20%	30%	40%
IDX	36.63	19.33	39.89	59.06	79.32
SIDX	24.30	14.98	13.36	14.70	14.97
SIDX+Psync	13.94	3.00	4.04	4.37	4.83

B. IO patterns of Index Scan, Sorted Index Scan and Flash-aware Index Scan

In order to show that the access methods work as we designed, we traced I/O pattern while carrying out the sample query. The logical selectivity of the query is 30% and P-sync I/O factor of flash-aware index scan is 128. Fig. 5(a) plots the I/O pattern of the index scan and Fig. 5(b) plots that of sorted index scan and the flash-aware index scan. As expected, the index scan access randomly data pages on SSD as shown in Fig. 5(a). Otherwise, in Fig. 5(b), we observed that sorted index scan (green) and the flash-aware index scan (red) sequentially read data pages on SSD. Also, we confirm that the flash-aware index scan exploits internal parallelism of SSD in Fig 5(b).

C. *Performance evaluation of Index Scan, Sorted Index Scan and Flash-aware Index Scan*

We set data block size, data buffer cache, and sort memory to 8 KB, 256 MB, and 1 MB respectively. The table size is about 800 MB with 8 KB-data block.

For the range query in Fig. 1 against the sample data with the nonclustered index, we measured the query execution time in seconds of three methods; the index scan (IDX), sorted index (SIDX) and the flash-aware index scan (SIDX+P-sync) on SSD. First of all, we vary the logical selectivity from 5% to 40% about a single user as shown in Table 1. Next, we conduct the same query with 12 users varying the logical selectivity as shown in Table 2.

Table 1 shows that the performance gap between the index scan and sorted index scan become larger as the selectivity increases. This is because the index scan produces random I/O access patterns and repetitive reads on the same disk pages more and more corresponding to the selectivity. Sorted index scan improves the performance about from 1.3 times to 5.3 times against the index scan. Compared to sorted index scan, flash-aware index scan (SIDX+P-sync) outperforms sorted index scan about from 3 times to 5 times since it takes an advantage of internal parallelism of SSD.

We don't conduct the same experiments with various buffer sizes. This is because a buffer size doesn't affect the performance sorted index scan because it reads the needed data pages only once. In addition, we flush every cache such as PostgreSQL buffer cache and OS cache whenever executing a query. Thus, sorted index scan and sorted index scan with P-sync I/O will still show a similar performance improvement even though a buffer cache is large.

Table 2 describes the performance of access methods with 12 users. It represents the similar performance improvement trend with Table 1. We can still obtain the improvements in sorted index scan and flash-aware index scan with multiple users due to internal parallelism of SSD as well.

D. *Performance evaluation of Full Table Scan, Sorted Index Scan and Flash-aware Index Scan*

We set data block size, data buffer cache, and sort memory to 4 KB, 256 MB, and 1 MB respectively. The table size is about 890 MB with 4 KB-data block.

We measured the query execution time in seconds of three methods; the full table scan (FTS), sorted index (SIDX) and the flash-aware index scan (SIDX+P-sync) on SSD for the same query in Section 4. C. We vary the physical selectivity from 10% to 100% and the number of concurrent users for each selectivity. Fig. 6 depicts the performance results.

First, Fig. 6 compares the performance of FTS and SIDX. Sorted index scan always outperforms the full table scan except for the cases only with single user over 20% of the physical selectivity due to the overhead of sorted index scan. However, the burden of sorted index scan is hidden by exploiting the internal parallelism of SSD when the concurrent users increase. Even sorted index scan is 70% faster than the full table scan when it reads the whole table with 24 concurrent users.

Next, Fig. 7 makes a comparison between the performance of the full table scan and that of the flash-aware index scan. Sorted index scan with P-sync I/O utilizes the internal parallelism of SSD better than sorted index scan does. The performance of sorted index scan with P-sync I/O is about from 2 times to 4.8 times faster than the full table scan. Comparing the performance of sorted index scan with that of the sorted index scan with P-sync, the sorted index scan with P-sync outperforms about from 1.5 times to 7 times against the sorted index scan.

V. CONCLUSION

In this paper, we implemented a flash-aware index scan that combines two concepts, sorted index scan and P-sync I/O, in PostgreSQL to exploits internal parallelism of SSD at most. As shown from the experimental results in Section 4, we showed the flash-aware index scan outperforms two conventional access method, the index scan and full table scan and sorted index scan as an approach for optimizing the index scan, at 1.5 times to 7 times. This is because the flash-aware index scan took an advantage of internal parallelism of SSD better than the three access methods as we expected as shown in Fig. 5.

Fig. 6. Performance of Full Table Scan and Sorted Index Scan

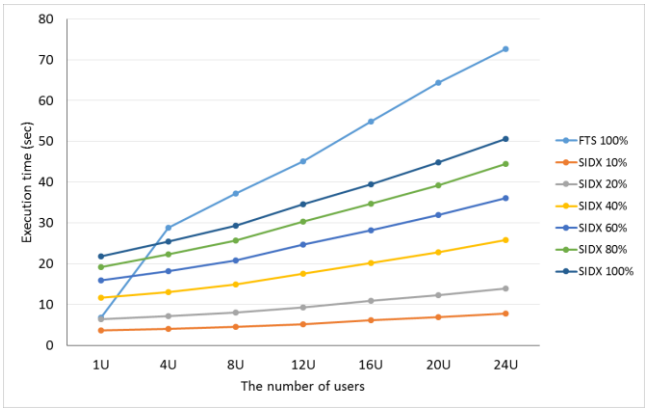


Fig. 7. Performamce of Full Table Scan and Sorted Index Scan with P-sync

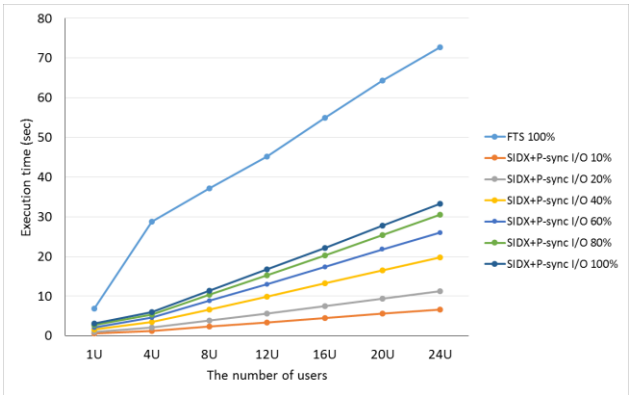
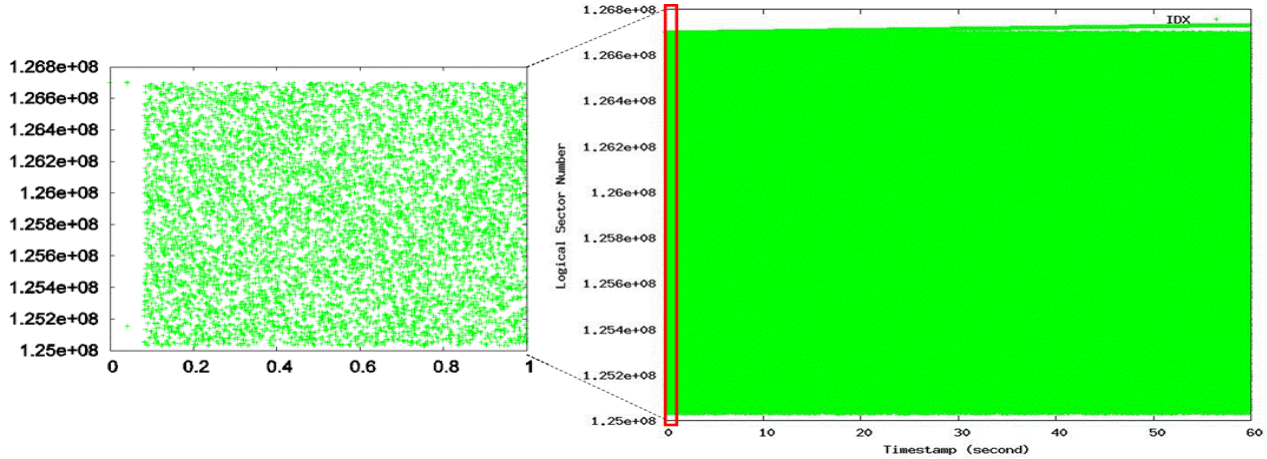
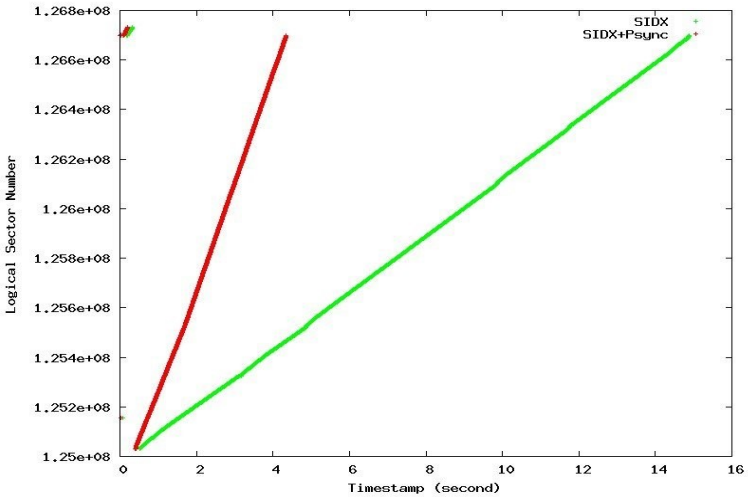


Fig. 5. IO Patterns of Access Method



(a) Index Scan



(b) Sorted Index Scan and Flash-aware Index Scan (SIDX+P-sync)

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2012R1A1A2A10044300). Also, This work was partly supported by the IT R&D program of MKE/KEIT(10041244, SmartTV 2.0 Software Platform).

REFERENCES

[1] Pedram Ghodsnia, Ivan T.Bowman, and Anisoara Nica, "Parallel I/O Aware Query Optimization", SIDMOD'14, June 22-27, 2014, Snowbird, UT, USA

[2] Eun-Mi Lee, Sang-won Lee, and Sang-won Park, "Optimizing Index Scan on Flash Memory SSDs", SIGMOD Record, December 2011, vol. 40, No.4

[3] 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." , Proceedings of ICDE, pages 171–180, 1991.

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

[5] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-won Lee, "B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives", Proceeding of the VLDB Endowment, Vol. 5, No.4

[6] F. Chen, R. Lee, and X. Zhang., "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing.", HPCA, pages 266-277, 2011.

[7] Intel. intel x25-m. http://download.intel.com/design/ash/nand/mainstream/Specification322296.pdf.

[8] Pedram Ghodsnia, Ivan T.Bowman, and Anisoara Nica, "Parallel I/O Aware Query Optimization", SIDMOD'14, June 22-27, 2014, Snowbird, UT, USA