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, the trend of storage markets has changed from HDDs that have dominated the markets for the last several decades to flash based SSDs. Corresponding to the drift, various studies have been conducted to adapt traditional DBMS to SSD based storage devices. However, most DBMSs are still more HDD-friendly.**

HDDs and SSDs have inherent features because of their own architecture designs. HDD has a wide gap between the performance of sequential I/O and that of random I/O owing to its mechanical parts. Due to this fact, DBMS usually prefer a full table scan to an index scan except when a selectivity is enough low to take advantage of index scan. Unlike HDDs, SSDs do not contain any mechanical parts and therefore there is a tiny gap between the performance of sequential I/O and that of random I/O. This characteristic of SSDs allows DBMSs to efficiently access a storage based SSD with an index scan. Another feature of SSDs is an internal parallelism thanks to its internal architecture. In spite of a circumstance with the secondary storage based SSD, DBMS are more likely to choose a full table scan rather than an index scan for I/O operations. It is necessary to understand the distinct properties and differences between the two storage devices and make index scans SSD-friendly to improve their performance.

In this paper, we implement an optimization of the index scan, called flash-aware index scan in the paper, by combining two concepts in PostgreSQL. A concept is a sorted index scan that scans tuples in order of record ids. The other is parallel synchronous I/O that is a traditional synchronous I/O with an array of I/O requests per operation. The goal of the paper is making the system aware of the nature of SSDs, thus enhancing the performance of the index scans.

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

I. INTRODUCTION

HDDs have monopolized storage markets for several decades. Most commercial DBMS have been further developed to improve their performance with HDD based storage systems. Flash based SSDs are considered as a substitute of HDDs as storage vendors have been able to supply cheap and high capacity SSDs within the last decade. Corresponding with this trend, various studies have been conducted to adapt traditional DBMS to SSD based storage devices. However, most DBMS are still more HDD-friendly.

HDD and SSD have inherent features due to their respective architecture designs. Mechanical parts in HDDs, which are an outstanding feature of HDDs, cause high I/O response times. In addition, a performance gap between sequential I/O and random I/O is largely owed to the existence of physical parts. Due to this fact, DBMS usually prefer a full table scan to the index scan except when a selectivity is low enough to take advantage of index scans.

Unlike HDDs, SSDs do not have any mechanical parts. Because of this characteristic, SSDs have a tiny gap between the performance of sequential I/O and that of random I/O. This allows DBMS to efficiently access to an SSD based storage system with index scans. Another feature of SSDs is an internal parallelism as a result of its internal architecture. It provides a performance improvement when DBMS exploit the parallelism of SSDs with proper techniques such as multiple I/O requests [1].

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

In this paper, we implement an approach in PostgreSQL that makes index scans flash-aware by combining two concepts: a sorted index scan and parallel synchronous I/O. Sorted index scans fetches tuples in an order of record identifiers [2]–[4] Therefore, It prevents us from reading the same pages repeatedly. 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 requests, but with an array of I/O requests per operation. The flash-aware index scan in PostgreSQL allows the database management system be aware of the nature of SSDs and thus enhance the performance of index scans.

This paper is organized as follows. Section 2 explains the key concepts, covering sorted index scans and parallel synchronous I/O in detail. Section 3 describes the flash-aware index scan with a combination of the two previous ideas, 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 the full details of a few concepts of the flash-aware index scan including a 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 a good influence on performance of an index scan with SSDs.

A. *Index clusteredness*

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

B. *Sorted Index Scan*

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

There is a drawback of a sorted index scan. We receive out-of-order records when we access a table with a sorted index scan since they are read in the ordering of the data pages on the disks. However, this shortcoming can be overcome with sort algorithms in DBMS and sorted index scans with sort algorithms outperform traditional index scans [2]. This is because I/O costs is more than 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*). PostgreSQL provided a tid scan that can work like a sorted index scan, but its implementation has a flaw. We modify the tid scan algorithm in PostgreSQL to work in a way that we desire. We will introduce it in detail in the next section. Furthermore, we need to transform queries to use tid scan because PostgreSQL exploits a tid scan only in a specific query form.

```
/* 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. Query transformation for sorted index scan in PostgreSQL

Fig. 1 describes the query transformation. We write the query with reference to [2]. The query returns tuples from the table *tab* of which have column values in the given range, between min and max. In this example, there is a nonclustered index in 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 a range of column values between *min* and *max*. The array is sorted according to the ordering of the data pages on the disks. We access data records and retrieve them with collected ctids..

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

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

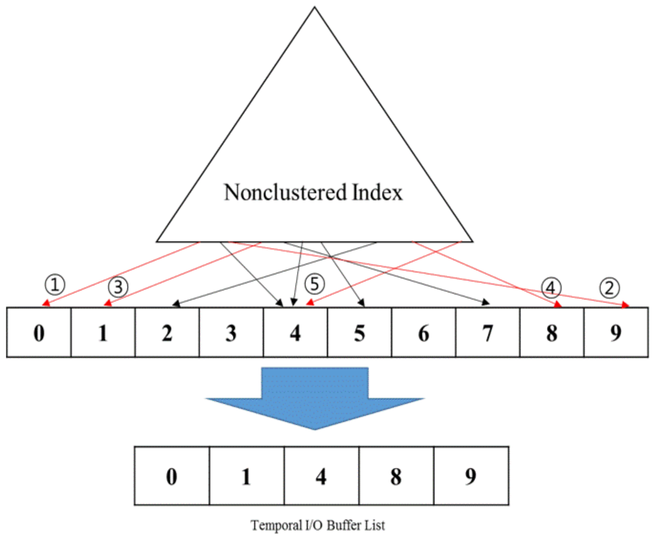


Fig. 2. Flash-aware Index Scan

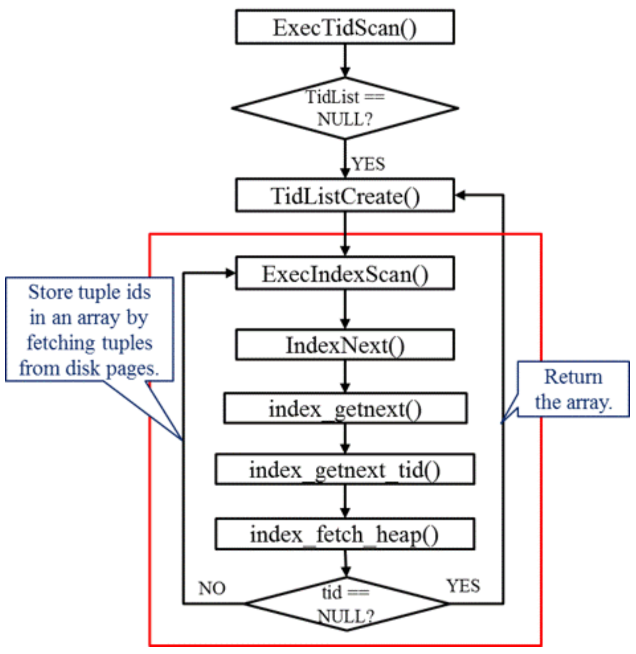


Fig. 3. Flowchart of the original tid scan

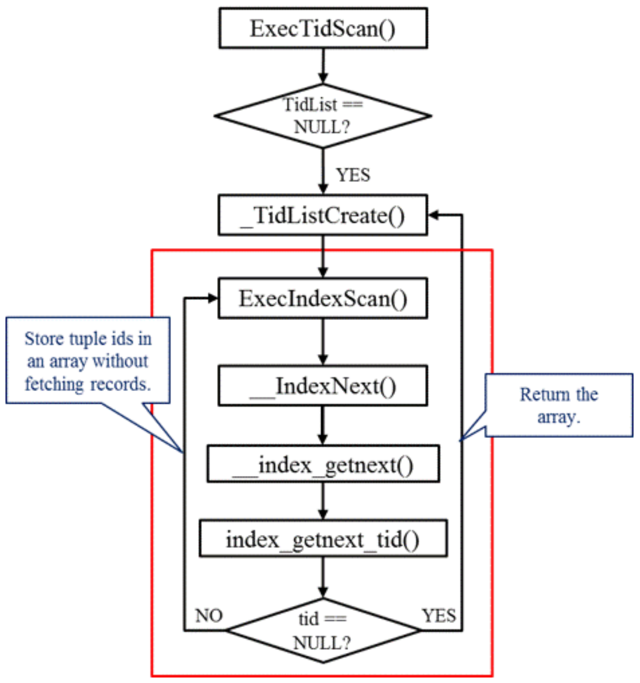


Fig. 4. Flowchart of modified tid scan

The paper has indicates three requirements of P-sync I/O as follows:

- The issue for P-sync I/O: It sends only a set of I/O requests to disks at a time and regains the results all at once. In other words, the other sets of I/O requests should suspend until the previous one receives the request results.
- Processing of P-sync I/O: An array of I/O requests is sent from user space to kernel space as a group and each request in the array should wait until all of them arrives in 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 need to consider how to handle I/O completion events.

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

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 exploit package-level parallelism.
- Create an array of I/O requests in order to utilize the parallelism of SSDs. 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 patterns.

III. Flash-Aware Index Scan

We implement a new index scan that is aware of a flash based SSDs by using a sorted index scan and P-sync I/O on PostgreSQL and explain how to implement our flash-aware index scan in detail.

A. *Flash-aware Index Scan*

We apply a sorted index scan and P-sync I/O to a traditional index scan to take advantage of the internal parallelism of flash based SSDs. Fig. 2 describes how the flash-aware index scan works. For this example, the P-sync factor is five. It means we can send and receive a maximum of five I/O requests at once. A database system requests 5 tuples and the requesting order is the same number with numbers in circles. A sorted index scan first 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 the tid scan algorithm in PostgreSQL as a sorted index scan but we modify several parts of the algorithm because of its implementation flaw. The tid scan algorithm is composed of two phases, like a 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 id order. However, while it collects tuple ids in the first stage, a tid scan fetches useless tuples one by one from disk pages by calling a function, *index_fetch_heap()*. It utilizes the normal index scan algorithm that fetches real data records when creating an array of tuple ids for a tid scan even though we can obtain tuple ids without fetcting data records. This causes extra read requests and thus a tid scan performs even worse than index scans do. We simply change the tid scan algorithm flow to get rid of unnecessary reads. Fig. 3 shows the original flow of a tid scan and Fig. 4 represents that of a tid scan without the previously mentioned defect. These flow charts do not show the exact details of the whole tid scan but they give a simple illustration of creating a tid list. To distinguish between the flow of a normal index scan and that of accumulating tuple ids for a tid scan, we define index related functions that are 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 are then just 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 is not suitable for the sorted index scan. Hence, we fixed the original tid scan not to fetch unnecessary tuples in the first procedure for a sorted index scan. Fig. 4 shows the modified tid scan that works as we expect for a sorted index scan by adding simple functions and slightly changing the tid scan flow.

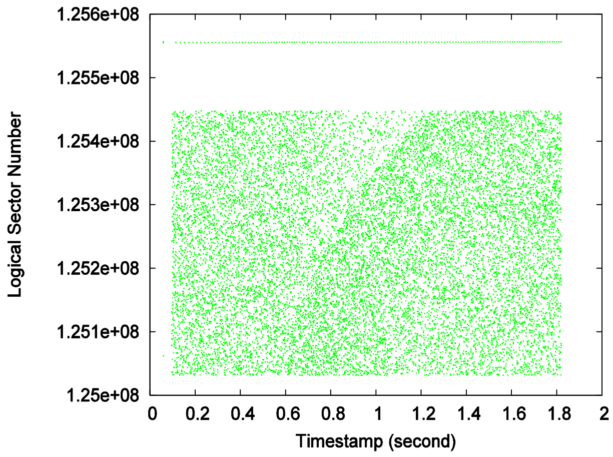
2) Parallel Synchronous I/O: We implement parallel synchronous I/O to deal with multiple I/O requests at once by using direct I/O and a library for asynchronous I/O, *Libaio*. For asynchronous I/O in PostgreSQL, we should also use direct I/O, and thus we add a direct I/O flag in proper positions such as when to open files and create new function pointers and functions for libaio. Also, we add a global variable, *MAX_LIBAIO* as a factor to scale how many I/O requests are sent to disks. We add several functions mimicking the normal functions as well for reading a data page for a buffer in PostgreSQL except that 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 the type of *Buffer pointer* instead of the type *Buffer,* contains multiple buffer pages' information and *mbufNum* always has the value of *MAX_LIBAIO* and lets functions know how many buffer pages will be read. Lastly, we hand over the *tidList* that we obtain in the first step of a tid scan through an *ItemPointerData* pointer parameter. This is in substitute of *BlockNumber* parameter in the original functions which correspond to the new functions. *tidList* has all of the necessary location information of the tuples that we are going to access and put the data pages from disks into *mbuf*. This is in substitute of *BlockNumber* parameter in the original functions corresponding to the new functions. *ReadBufferMultiple_common()* actually plays a role in handling buffer requests. The other functions are wrappers or simple callers. *ReadBufferMultiple_common()* is similar with *ReadBuffer_common()* except that it deals with multiple buffers.

IV. EXPERIMENTAL RESULTS

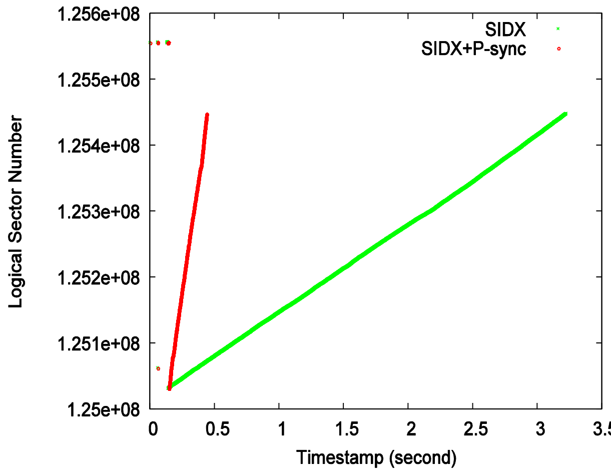
We carried out our experiments imitating the experiments in [2]. In addition, experimental settings as well as the sample table were based on [2].

A. *Experimental background*

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

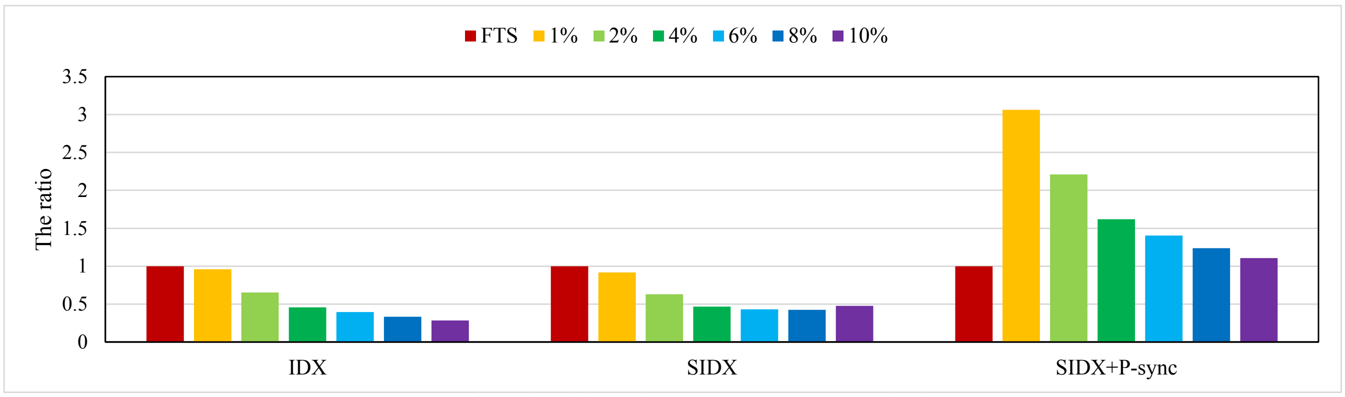


(a) Index Scan

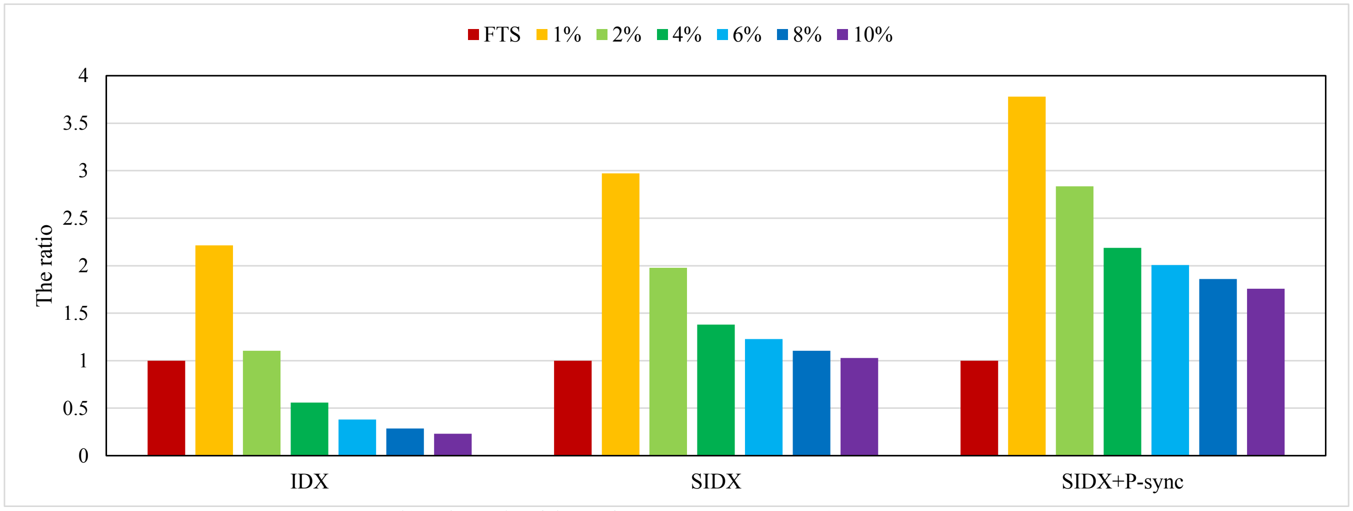


(b) Sorted Index Scan and Flash-aware Index Scan (SIDX + P-sync)

Fig. 5. IO Patterns of Access Methods



(a) The Ratio of the Performance Improvements (A Single User)



(b) The Ratio of the Performance Improvements (20 Concurrent Users)

Fig. 6. The Ratio of the Performance Improvement

The sample table has 2.5 million tuples and its tuple is 50B long. Owing to wanting 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 copied the sample table for each user when we executed queries with multiple users.

In this paper, a selectivity represents the ratio of the requested number of records to the total number of records. We adjusted a selectivity by a range of column *a* in *WHERE* of each query. We carried out experiments with various selectivity less than 10%. This was because the whole table on the disks is accessed with 10% selectivity.

We set data block, data buffer cache, and sort memory to 4 KB, 128MB, and 1 MB respectively. The table size was about 200 MB with 4 KB-data blocks. We did not conduct the same experiments with various buffer sizes. This is because a buffer size doesn't affect the performance of a sorted index scan because it reads the needed data pages only once. In addition, we flush every cache such as the PostgreSQL buffer cache and the OS cache whenever executing a query. Thus, a sorted index scan and a sorted index scan with P-sync I/O will still show a similar performance improvement even if the buffer cache is large.

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 the I/O pattern while carrying out the sample query. The selectivity of the query was 1% and P-sync I/O factor of our flash-aware index scan was 128. Fig. 5(a) plots the I/O pattern of the index scan and Fig. 5(b) plots that of a sorted index scan and the flash-aware index scan. As expected, an index scan randomly accesses data pages on the SSD as shown in Fig. 5(a). Otherwise, in Fig. 5(b), we observed that the sorted index scan (green) and the flash-aware index scan (red) sequentially read data pages on the SSD. Also, we confirm that the flash-aware index scan exploits the internal parallelism of SSD in Fig 5(b).

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

For the range query in Fig. 1 against the sample data with a nonclustered index, we measured the query execution time of the four different methods in seconds; a full table scan(FTS), an index scan (IDX), a sorted index (SIDX) and the flash-aware index scan (SIDX+P-sync) on the SSD. Firstly, We varied the selectivity from 1% to 10% about a single user. Next, we conduct the same query with 20 concurrent users varying the selectivity.

Fig 6 shows the ratio of the performance improvements for the execution time of queries with three access methods compared to that of a full table scan. X-axis is about the access methods and Y-axis is about the ratio of the performances.

First of all, Fig 6(a) shows the values of a ratio that each access method improves with a single user in various selectivity. The ratio for an index scan decreases as the selectivity increases. This is because an index scan randomly accesses the data pages, and thus the amount of the data pages being read keeps growing. The ratio for a sorted index scan shows the same trend with the index scan's. A sorted index scan performs 0.02% and 0.05% worse than the index scan does with 1% and 2 % selectivity respectively. This is because the overhead of the sorted index scan even if a sorted index scan sequentially reads the disk pages. In the case, the sorted index scan cannot fully exploit the internal parallelism of the SSDs with a single user, and thus the burden of creating a tid list is not hidden. However, the performance of a sorted index scan gets better and it works 1.7 times better than an index scan does with 10% selectivity. Lastly, the flash-aware index scan exploits the internal parallelism of the SSDs thanks to P-sync I/O. The flash-aware index scan outperforms from about 1.1 to 3 times against full table scan.

Next, Fig 6(b) depicts the values of a ratio that each access method improves with 20 concurrent users in various selectivity like Fig b(a). The ratio for an index scan decreases as the selectivity increases as well except for the case where the selectivity is 1%. An index scan reads less amount of the data pages than how many data pages a full table scan reads, and thus it outperforms a full table scan about 2 times even if an index scan randomly accesses the data pages. A sorted index scan is able to exploit the internal parallelism of the SSDs with multiple concurrent users. Also, a sorted index scan reads each page only once, and thus a sorted index scan performs 2.97 times better at a maximum than a full table scan does. With 8% and 10% selectivity, a sorted index scan has to read about 97% of the table and the whole table respectively. Despite this fact, a sorted index scan with 8% and with 10% selectivity performs slightly better than a full table scan does. This is because the overhead of a sorted index scan can be hidden despite of utilizing the internal parallelism of the SSD. The flash-aware index scan outperforms a full table scan about from 3.8 to 1.7 times since concurrent users enable the flash-aware index scan to take advantage of the internal parallelism of the SSD at a maximum.

V. Conclusion

In this paper, we implemented a flash-aware index scan that combines two concepts, a sorted index scan and P-sync I/O. It was implemented in PostgreSQL to exploit the internal parallelism of SSD as much as possible. As shown from the experimental results in Section 4, we showed that our flash-aware index scan outperformed two conventional access method, namely an index scan and a full table scan, and a sorted index scan as an approach for optimizing the index scan, from 1.1 to 3.8 times. This is because the flash-aware index scan reduced the amount of required I/O. In addition, it took advantage of the internal parallelism of SSDs better than the three access methods as we expected as shown in Fig. 5 by sequentially reading the required data pages only once and requesting multiple I/Os at once.

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.