Optimizing Database Operators by Exploiting Internal Parallelism of Solid State Drives

Yulei Fan, Wenyu Lai, and Xiaofeng Meng
School of Information, Renmin University of China, Beijing, China

{fyl815,xiaolai913,xfmeng}@ruc.edu.cn

Abstract

With the development of flash memory technology, flash-based solid state drives (SSDs) are gradually used in more and more devices and applications. In addition to characteristics of flash memory itself, a unique characteristic of SSDs, namely internal parallelism, should also be considered to improve performance of SSDs-based DBMSs, especially query processing. In this paper, we first describe the internal architecture of SSDs and the resulting internal parallelism of SSDs. In the second part, we present a parallel table scan operator, ParaScan, that exploits the internal parallelism of SSDs. Based on ParaScan, we then propose a parallel hash join operator, ParaHashJoin, and a parallel aggregation model, ParaAggr. Experimental results show that ParaScan, ParaHashJoin and ParaAggr on SSDs significantly outperform traditional table scan, hash join and aggregation. Furthermore, sort, as an important basic operator for other complex operators, can also be redesigned based on ParaScan. We design a parallel sort algorithm, ParaSort and then present a parallel ParaSort operator in the third part. Looking forward, database query processing by exploiting internal parallelism of SSDs, can be generalized to other kinds of SSDs with similar internal parallel characteristics.

1 Introduction

As flash memory-based solid state drives (SSDs) improve in price, capacity, reliability and performance, more and more devices and applications gradually adopt SSDs as secondary storage media. Not only do SSDs provide faster access speed, lower power consumption, lighter weight, smaller size and better shock resistance than HDDs, they also have rich internal parallelism that can be used to improve I/O bandwidth[1, 2]. Query processing of HDDs-based DBMSs are mainly designed according to HDDs' mechanical limitations, so they may benefit less or even nothing when SSDs simply replace HDDs for OLAP and OLTP[3]. Beside the characteristics of flash memory, the internal parallelism of SSDs need to be considered when designing query processing algorithms for SSDs-based DBMSs including scan, join, aggregation, sort, etc[4].

This paper explores how to optimize scan, join, aggregation and sort operators by exploiting the internal parallelism of SSDs. We review related work in section 2. In section 3, we outline how to detect the internal parallel architecture of SSDs. And then in section 4, we present parallel table scan ParaScan, parallel hash join ParaHashJoin and parallel aggregation ParaAgg and verify their efficiency by running them on HDD and SSDs. And then we propose a parallel sort ParaSort in section 5. Finally, we offer conclusions in section 6.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Related Work

To achieve higher capacity and better I/O performance, most SSDs adopt a multi-channel and multi-way architecture for connecting flash chips with a flash memory controller. The flash memory controller can access flash chips in parallel[5]. Therefore, it is necessary to understand the impact of this parallelism inside SSDs on the performance of SSDs-based DBMSs. Chen[2] studied how to uncover internal parallelism features of SSDs by detailing an abstract parallel model of SSDs and methods of detecting SSDs. Roh[6] created a new I/O request method in the OS that can be used to generate parallel I/Os to access SSDs by applications. Based on this, the authors designed a new B+-tree variant called PIO B-tree. Hu[7] analyzed the performance impact and interplay of advanced commands, physical-page allocation and data granularity for internal parallelism of SSDs. According to internal parallelism architecture of SSDs, Park[8] designed parallel-aware request processing including request rescheduling and dynamic write request mapping.

Myers[9] and Lee[10] measured the performance of various traditional database algorithms on SSDs. For flash-based DBMSs, a number of query processing algorithms have been designed for flash characteristics, especially excellent random read algorithms[11, 12, 13, 14, 15, 16, 17]. Myers[9] improved the sort-merge join algorithm by optimizing the storage of intermediate results. Graefe[12, 13] focused on speeding up select, project and join operators based on a new page layout, PAX, for SSDs-based DBMSs. Simultaneously, they proposed FlashScan, FlashJoin and RARE-join algorithms and implemented them in PostgreSQL. Liang[14] optimized a join algorithm for column-based DBMSs by reading only needed columns. Li[15, 16, 17] improved no-index join algorithms by reducing intermediate results and optimizing the fetching strategy. Different from previous studies, we try to optimize scan, aggregation, join and sort operators by exploiting the internal parallelism of SSDs.

3 Internal Parallelism of SSD

3.1 Internal Parallel Architecture

The internal architecture of SSDs includes the host interface, SSD controller, RAM buffer and flash memory packages[8]. SSDs connect to the host by a host interface that can be SATA, SAS or PCIe. The SSD controller executes I/O requests and issues commands to flash memory packages via a flash controller. The SSD controller is also in charge of managing the RAM buffer which holds the address mapping table of the flash translation layer (FTL) and other metadata such as ECC etc. SSDs implement internal parallelism by adopting multiple channels that can operate independently and simultaneously. Each channel is shared by a set of flash memory packages, on which operations can also be interleaved, so the bus utilization can be optimized[5, 18]. As shown in[8], channel-level parallelism and package-level parallelism are two typical levels of parallelism. Such rich internal parallelism provides us an opportunity to improve the performance of applications on SSDs.

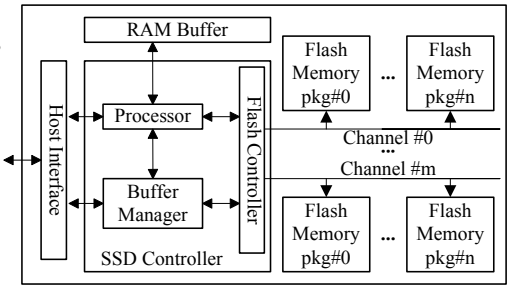

Figure 1: Inside SSD

3.2 Detecting SSD Internals

Before detecting and utilizing the internal parallelism, we must ensure that SSDs have native command queuing (NCQ) mechanisms and AHCI (Advanced Host Controller Interface) mode must be enabled for the host[19]. Multi-threaded processing can be used to produce multiple parallel I/Os. Furthermore, it is necessary to know some key architectural features of SSDs such as the number of channels for setting a proper concurrency level

and avoiding over-parallelization. However, it is hard to obtain such information because these details are often regarded as critical intellectual property of SSD manufacturers. Therefore, we try to detect two representative SSDs as shown in Table 1. Base on publicly available documents, Chen[2] characterized internal organization of SSD as an abstract model including chunk size and the number of domains. A chunk is a unit of data that is continuously allocated within one domain. A domain is a set of flash memories that share a specific set of resources. Guided by the model and the detecting method[2], we obtain chunk size and the number of domains of SSD-S and SSD-M as shown in Table 1 (detailed detecting results are in [21]).

Table 1: Detecting Results

	Manufacturer	Flash	Capacity	Page Size	Interface	NCQ	Chunk Size	Domains No.
SSD-S	Intel	SLC	32 GB	4 KB	SATA	32	4 KB	20
SSD-M	Intel	MLC	160 GB	4 KB	SATA	32	16 KB	20

4 ParaScan, ParaHashJoin and ParaAggr

4.1 ParaScan: Parallel Table Scan

Most SSDs adopt RAID-0 data storage as shown in Fig. 2. In SSDs-based database systems, records within a chunk are organized according to traditional row-based page layout. One chunk is one logical data page with an unique logical address. According to logical addresses, all chunks of a relational table are uniformly placed in all domains. All striped chunks of a relational table are mostly placed in logical consecutive regions in a file. On the assumption of 20 domains, the chunk whose logical address is 20*n (n=0,1,2...), will be in 1st domain, the chunk whose logical address is 20*n+1 (n=0,1,2...), will be in 2nd domain, and so on. Striped domains provide interleaved accesses to reduce sharing of data-bus channels. For example, chunk 0 and chunk 20 in domain 0 cannot be accessed simultaneously, but chunk 0 in domain 0 and chunk 1 in domain 1 can be read in parallel. This provides an opportunity to distribute different data accesses to different domains.

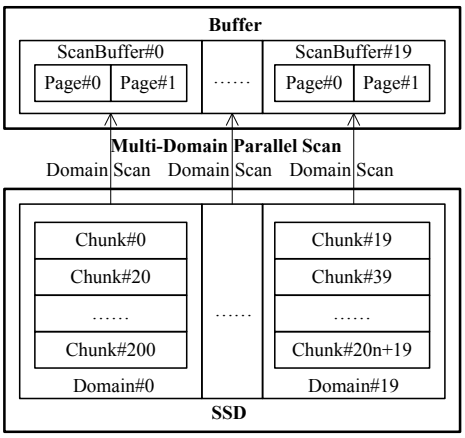

Figure 2: ParaScan

Based on this, we propose a parallel table scan called ParaScan to improve the efficiency of table scan. As shown in Fig. 2, every domain scan reads data chunks one by one from corresponding domain and then puts them into its own ScanBuffer. Multiple domain scan operations can be executed independently and simultaneously. All domain scan and all ScanBuffers compose multi-domain parallel scan named ParaScan. Because chunks are mostly placed in logical consecutive regions, we suggest that all data pages of a relational table should be uniformly distributed into different domains as much as possible to make full use of the internal parallelism of SSDs. Data size in one domain usually exceed ScanBuffer size, so it is necessary to replace processed data pages in ScanBuffer with unprocessed data pages when executing domain scan.

We implement ParaScan by a multi-thread processing technique. Each domain scan corresponds to one thread. The performance of ParaScan depends not only on the number of domains but also on the number of physical threads supported by CPU and NCQ supported by SSDs. We ran ParaScan on an HP PC with Ubuntu 12.10, 8G DDR3 memory, a 500G 7400rpm SATA3 Seagate HDD, Intel Core i5-2400 @ 3.10GHz processor with four cores supporting four physical threads and two SSDs as shown in Table 1. The data set was the ORDERS table(150 million rows, 2G) taken from TPC-H benchmark. Fig. 3 compares the performance of scans while varying the number of threads. Traditional table scan is ParaScan implemented by a single thread.

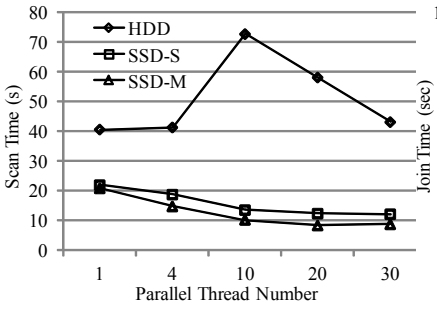

Figure 3: ParaScan

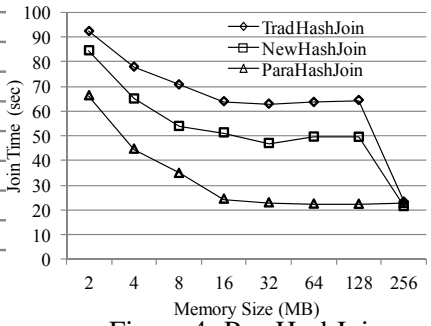

Figure 4: ParaHashJoin

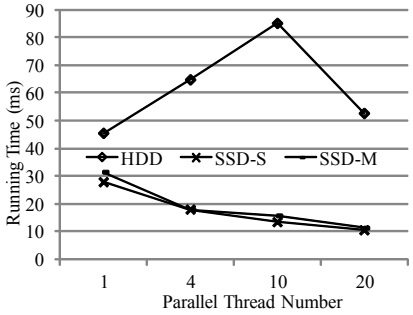

Figure 5: ParaAggr

Experimental results show that ParaScan isn't well suited for HDD but can exploit the internal parallelism of SSDs (detailed experimental results are in [20, 21]).

4.2 ParaHashJoin: Parallel Hash Join

Based on ParaScan, we present a parallel hash join operator called ParaHashJoin. It consists of three phases, ParaHash, MiniJoin and Fetch, as shown in Fig. 6. ParaHash hashes records of different domains in parallel to exploit the internal parallelism of SSDs. The output of ParaHash only contains the row-id (RID) and the join attribute of every record to reduce the amount of data processed in MiniJoin. A RID consists of page id and in-page offset of a record. Finally, Fetch retrieves the needed attributes by RIDs. Because the output of ParaHash and MiniJoin are incomplete results, ParaHashJoin uses less non-volatile storage to materialize intermediate results. However, multi-threaded processing and random reads in ParaHashJoin will cost more CPU. But experimental results show that this tradeoff is worthwhile.

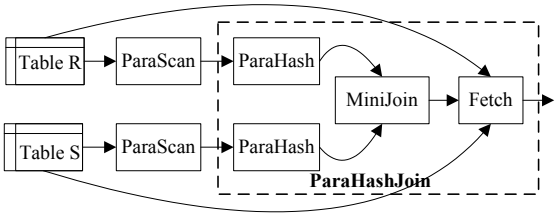

Figure 6: ParaHashJoin

As shown in Fig. 7, each ParaHash instance calculates hash values of records in its ScanBuffer and then adds their join attribute values and RIDs into specific hash buckets. Each hash bucket must maintain a lightweight lock to solve conflicts between hash threads. Hash function is $hash\_value = join\_attr\&(B-1)$. If the results of ParaHash on table R can be maintained in memory, we only need to do ParaScan on table S after ParaScan and ParaHash on table R. And then according to join attribute value of each record of table S in ScanBuffers, MiniJoin directly probe hash buckets of table R and produce join results in the form $\{join\_attr, RID_R, RID_S\}$. Otherwise, MiniJoin gets one hash buckets of table R and corresponding hash bucket of table S into memory to generate join results, and processes other hash buckets in this way.

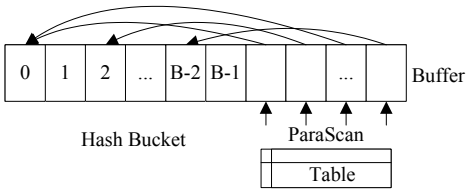

Figure 7: ParaHash

Because results of MiniJoin are incomplete join results, according to RID in join results, Fetch randomly reads necessary attribute values to generate final join results. Designing efficient fetching strategy is very important for minimizing I/Os of reading data pages, but we adopt a sort-based fetching strategy inspired by DigestJoin[15, 16, 17] to avoid reloading pages as much as possible as shown in [20]. Before fetching data pages, we sort results of MiniJoin, so needed data pages can be loaded in order. The sort will cost more CPU but we show that this tradeoff is worthwhile in experimental results.

Implementation technique and running platform of ParaHashJoin are the same as ParaScan. In addition to the ORDERS table, the input data set includes the CUSTOMER table(1.5 million rows, 256M) obtained from TPC-H benchmark. In Fig. 4, TradHashJoin adopts traditional hash join and table scan. NewHashJoin adopts

traditional hash join and ParaScan. We mainly consider equi-join of CUSTOMER and ORDERS on a single attribute. Join selectivity is 1%. The number of threads in ParaScan and ParaHash is fixed at 20. In this paper, we only present experimental results on SSD-S. Fig. 4 evaluate the impact of memory size on three join algorithms as memory size varied from 2MB to 256MB. Both TradHashJoin and NewHashJoin require 256M to execute in one pass while ParaHashJoin only need 16M. Fig. 4 shows that, in the best case, ParaHashJoin is 1-1.5 x faster than NewHashJoin and TradHashJoin (detailed experimental results are in [20]). Our experimental results show that it is worthwhile to pay the extra CPU cost for multi-threaded processing and random reads.

4.3 ParaAggr: Parallel Aggregation

Based on ParaScan, we propose a parallel aggregation algorithm called ParaAggr for implementing some simple aggregation such as *sum*, *max*, *min*, *count*, *average* etc. ParaAggr consists of two phase, SubAggr and TolAggr, as shown in Fig. 8. Each SubAggr instance only processes records in its ScanBuffer and all SubAggr run in parallel. Either every computing result can be forwardly put to TolAggr, or TolAggr forwardly get all computing reseults produced in SubAggr. And then TolAggr calculates all results of SubAggr. The calculations in SubAggr phase and TolAggr phase may be same or different. For example, *count* operator can be divided into count in SubAggr and summation in TolAggr(others are shown in [21]).

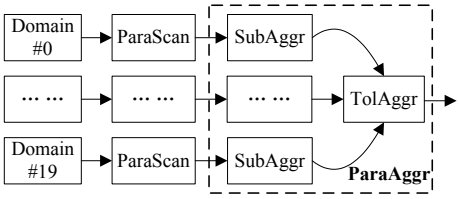

Figure 8: ParaAggr

By exploiting multi-threaded processing we implemented ParaAggr including *sum*, *max*, *min*, *count*, *average*. We ran ParaAggr on a Lenovo K46A with Fedora 14, 2G DDR3 memory, a 500G 5400rpm SATA3 Seagate HDD, Intel Core i5 @450MHZ processor with double cores supporting four physical threads, two SSDs as shown in Table 1 and another SSD as shown in [21]. The data set is the CUSTOMER table(870 thousand rows, 1GB) taken from the TPC-C benchmark. Fig. 5 compares the performance of aggregations while varying the number of SubAggr threads. Traditional aggregation is ParaAggr with 1 SubAggr thread. As shown in Fig. 5, in the best case, ParaAggr is 3-4 x faster than traditional aggregation. Other experimental results are detailed in [21].

5 ParaSort: Parallel Sort for Up-level Operators

Sort is an important operator for up-level operators such as join, group by, having, limit, sub-query etc. In this section, we proposed Parallel Sort Model called ParaSort and present Parallel ParaSort.

5.1 Parallel Sort Model

We now sketch a parallel sort algorithm called ParaSort based on ParaScan. As show in Fig. 9, ParaSort consists of two phases, SubSort and Fetch. SubSort is in charge of getting sorted attribute values and RIDs of all records and then sorting them according to the sort key. SubSort outputs ordered results the sort key and RID and then Fetch randomly reads required attribute values required by up-level operators. The fetching strategy can utilize the strategy mentioned in 4.2. As the critical component of ParaSort, SubSort can be implemented according to traditional sort method to sort any size data, but SubSort can't run until the end of ParaScan if ScanBuffers can store all records; otherwise, SubSort will carry out as long as ScanBuffers is full. What's more, SubSort can be implemented in parallel when sorted attribute values and RIDs of all records fit in memory.

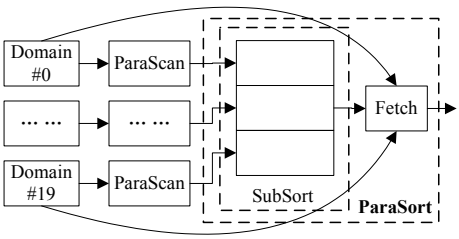

Figure 9: ParaSort

5.2 Parallel ParaSort

When memory is enough, we discuss a parallel ParaSort by implementing SubSort in parallel. As shown in Fig. 10. Every SubSort gets sorted keys and RIDs of data records from corresponding ScanBuffer and then sorts them according to sort key. All SubSorts run in parallel. Results of one SubSort is ordered but results of all SubSorts is semi-ordered. TolSort merges semi-ordered results of all SubSorts to obtain ordered results. Fetch retrieves the necessary attribute values for up-level operators by ordered results of TolSort. The fetching strategy can also use the strategy mentioned in 4.2.

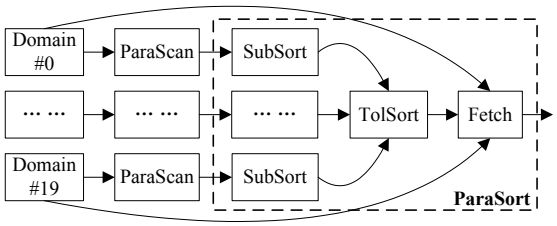

Figure 10: Parallel ParaSort

6 Conclusions

In addition to excellent performance characteristics of flash memory, there is rich parallelism inside SSDs. Previous researchers mainly focus on optimizing algorithms by exploiting properties of flash memory, but the internal parallelism of SSDs also needs to be considered when optimizing SSDs-based DBMSs, especially query processing.

By exploiting the rich internal parallelism of SSDs, we presented ParaScan and then proposed ParaHashJoin and ParaAggr based on ParaScan. Experimental results showed that we were able to speed up performance of ParaScan, ParaHashJoin and ParaAggr by $>$ 1x. And for complex operators, we designed ParaSort. In other words, many database operators can be sped up by exploiting the internal parallelism of SSD. Future research will show whether this is feasible, effective and efficient and how it affect industrial OLAP and OLTP systems.

Acknowledgments

This research was partially supported by the grants from the Natural Science Foundation of China (No. 61379050, 91224008); the National 863 High-tech Program (No. 2013AA013204); Specialized Research Fund for the Doctoral Program of Higher Education(No. 20130004130001), and the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University(No. 11XNL010).

References

[1] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In SIGMETRICS, pages 181-192, 2009.

[2] F. Chen, R. Lee and X. Zhang. Essential Roles of Exploiting Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing. In HPCA, pages 266-277, 2011.

[3] S.-W. Lee and B. Moon. Design of flash-based DBMS: An in-page logging approach. In SIGMOD, pages 55-66, 2007.

[4] R. Ramakrishnan and J. Gehrke. Database Management Systems (3rd Ed.). McGraw Hill, 2002.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In USENIX, pages 57-70, 2008.

[6] H. Roh, S. Park, S. Kim, M. Shin and S.-W. Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. Proc. VLDB, 5(4): 286-297, 2012.

[7] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo and S. P. Zhang. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granu-larity. In ICS, pages 96-107, 2011.

[8] S. Y. Park, E. Seo, J. Y. Shin, S. Maeng and J. Lee. Exploiting internal parallelism of flash-based SSDs. Computer Architecture Letters, 9(1):9-12, 2010.

[9] D. Myers. On the use of NAND flash memory in high-performance relational databases. MIT Msc Thesis, 2008.

[10] W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In SIGMOD, pages 1075C1086, 2008.

[11] J. Do and J. M. Patel. Join processing for flash SSDs: remembering past lessons. In DaMoN, pages 1-8, 2009.

[12] M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe. Fast scans and joins using flash drives. In DaMoN, pages 17-24, 2008.

[13] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In SIGMOD, pages 59-72, 2009.

[14] Zh. Liang, D. Zhou, X. Meng. Sub-Join: Query Optimization Algorithm for Flash-based Database. Journal of Frontiers of Computer Science and Technology, 2010, 4(5): 401-409.

[15] Y. Li, S. T. On, J. Xu, B. Choi, H. Hu. DigestJoin: Exploiting Fast Random Reads for Flash-based Joins. In MDM, pages 152-161, 2009.

[16] Sh. Gao, Y. Li, J. Xu, B. Choi, H. Hu. DigestJoin: Expediting Joins on Solid-State Drives. In DASFAA pages 428-431, 2010.

[17] Y. Li, S. On, J. Xu, B. Choi, H. Hu. Optimizing Nonindexed Join Processing in Flash Storage-Based Systems. IEEE Transactions on Computers, 62(7):1417-1431, 2013.

[18] C. Dirik and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, achitecture, and system organization. In ISCA, pages 279-289, 2009.

[19] S.-W. Lee, B. Moon, and C. Park. Advances in flash memory SSD technology for enterprise database applications. In SIGMOD, pages 863-870, 2009.

[20] W. Lai, Y. Fan, X. Meng. Scan and Join Optimization by Exploiting Internal Parallelism of Flash-Based Solid State Drives. In WAIM, pages 381-392, 2013.

[21] Y. Fan, W. Lai, X. Meng. Database Table Scan and Aggregation by Exploiting Internal Parallelism of SSDs. Chinese Journal of Computers, Vol 35(11):2327-2336, 2012.