Parallel I/O Aware Query Optimization

Pedram Ghodsnia
University of Waterloo
200 University Ave E
Waterloo, Canada
pghodsnia@uwaterloo.ca

Ivan T. Bowman
SAP AG
445 Wes Graham Way
Waterloo, Canada
ivan.bowman@sap.com

Anisoara Nica
SAP AG
445 Wes Graham Way
Waterloo, Canada
anisoara.nica@sap.com

ABSTRACT

New trends in storage industry suggest that in the near future a majority of the hard disk drive-based storage subsystems will be replaced by solid state drives (SSDs). Database management systems can substantially benefit from the superior I/O performance of SSDs.

Although the impact of using SSD in query processing has been studied in the past, exploiting the I/O parallelism of SSDs in query processing and optimization has not received enough attention. In this paper, at first, we show why the query optimizer needs to be aware of the benefit of the I/O parallelism in solid state drives. We characterize the benefit of exploiting I/O parallelism in database scan operators in SAP SQL Anywhere and propose a novel general I/O cost model that considers the impact of device I/O queue depth in I/O cost estimation. We show that using this model, the best plans found by the optimizer would be much closer to optimal. The proposed model is implemented in SAP SQL Anywhere. This model, dynamically defined by a calibration process, summarizes the behavior of the I/O subsystem, without having any prior knowledge about the type and the number of devices which are used in the storage subsystem.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; H.2.2 [**Database Management**]: Physical Design—*Access methods*

Keywords

Parallel I/O; Query Optimization; SSD; I/O Cost Model; Access Path;Index Scan;Full Table Scan; Prefetching

1. INTRODUCTION

For decades hard disk drives have been the dominant solution for storage subsystem in database systems. By advent of flash based solid state drives (SSDs) a revolutionary advancement in storage technology has happened in recent years. As predicted correctly by Jim Grey in 2006, "tape is dead, disk is tape, and flash is disk" [11]. Compared to HDDs, SSDs benefit from a much lower latency and much higher throughput for random I/O. SSDs consume significantly less power and produce much less heat. Due to all these outstanding properties, it can be clearly seen that HDDs are getting gradually replaced by SSDs. Due to the extraordinary business impact of SSDs, in recent years, application of SSDs in I/O intensive applications such as database systems has been studied extensively [2, 7, 8, 12, 13, 14, 16, 17, 18, 23].

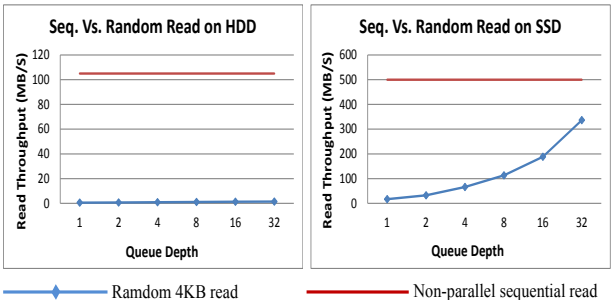


Figure 1: Throughput on non-parallel sequential reads vs. parallel 4KB random reads for different queue depths on HDD and SSD

Although the performance impact of SSDs on different database operators has been studied in the past, the impact of exploiting the I/O parallelism in SSDs has been vastly overlooked. Modern SSDs can substantially benefit from I/O parallelism. The substantial impact of parallelism in I/O performance of SSDs has been studied in [5]. A modern SSD is capable of utilizing multiple levels of parallelism; namely, plane level, channel-level, package level, and die level parallelism. Almost all modern SSDs support command queuing mechanisms (NCQ and TCQ). These mechanisms allow the SSD to accept multiple concurrent I/O requests or a burst of successive I/O requests from the operating system. The received I/O requests are queued up and the host interface will reorder them to make a favorable I/O pattern for the internal parallel organizations in the SSD. In other words, increasing the I/O queue depth of the modern SSDs will result in a significant increase in I/O throughput of the random I/O. The *I/O queue depth* is the average number of outstanding I/Os in the I/O queue at any point of time. The I/O queue depth can be increased by issuing multiple I/O requests at the same time or issuing I/O requests with a rate faster than the rate of handling I/O requests by device.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2595635.

By increasing the I/O queue depth, the outstanding parallel I/O capability of the device can be exploited. Consequently, the performance of random I/O can potentially become very close to the performance of sequential I/O. Therefore, by careful design of the I/O access pattern of the database operators, a parallel random I/O pattern can perform almost as well as a sequential I/O pattern. Fig. 1 shows the difference between the I/O performance of sequential I/O and parallel random I/O. On SSD, by increasing the queue depth to 32 the performance of random I/O becomes about 51.7% of the performance of sequential I/O. However, on HDD, by using a queue depth of 32, the performance of random I/O would be only about 1.3% of the performance of sequential I/O. In this experiment a consumer level PCIe-based SSD and a commodity 7200RPM HDD is used.

Traditionally, most query optimizers are designed by taking into account the substantial disparity between the runtime of random and sequential I/O in HDDs. Moreover, conventionally, query optimizers assume that there is no considerable performance difference between parallel and non-parallel I/O. These assumptions have been proved to work very well for decades of using HDD-based storage subsystems. However, by moving from HDD to SSD, relying on the same assumptions will result in suboptimal optimization decisions.

In this paper we study the impact of exploiting I/O parallelism in query processing and optimization. The contributions of our work can be summarized as follows.

1. We characterize the impact of using I/O parallelism in scan operators in SAP SQL Anywhere. We show how the optimal decision made by query optimizer can be affected when parallel I/O is employed. In [19], it is argued that an SSD-oblivious query optimizer is unlikely to make significant errors in choosing optimal access methods. We show that, when the I/O parallelism is employed in execution of the access methods, this argument is no longer true. In other words, our experiments show that the query optimizer needs to be SSD-aware, and in general, it needs to be aware of the benefit of I/O parallelism in storage device.

2. We perform experiments similar to those performed by others [15, 19] on a different DBMS and over different configurations. We use our experiments to validate the earlier results and compare them to new conditions.

3. Lee et al. [15] studied the impact of inter-query parallelism in exploiting the parallel I/O capability of SSDs. Roh et al. [21] proposed a new approach for executing multiple index scans at the same time. It is shown that their proposed method increases the I/O queue depth of the SSD and consequently improves the performance of the execution of the index scan. We show that intra-query parallelism and effective use of prefetching are two alternative approaches for exploiting the I/O parallelism of SSDs in index scans. We will discuss how these two alternatives eliminate the limitations of the existing approaches.

4. While there are some studies showing approaches to improve I/O parallelism, the query optimization problem has not been fully addressed. In particular, a query optimizer that operates on a range of storage technologies (HDD, RAID HDD, SSD, and even future technologies) must have a principled way to determine what the likely benefit is when using I/O parallelism in order to distribute parallelism opportunities among query operators. We propose a novel, general, and dynamic I/O cost model for accurate I/O cost estimation of those database operators which can benefit from I/O parallelism. The proposed model, dynamically defined by a calibration process, accurately summarizes the behavior of the storage device, no matter how much it can benefit from I/O parallelism. The proposed model is implemented in SAP SQL Anywhere. Our experiments show that leveraging the new model in the query optimizer results in selecting execution plans with up to 20 times shorter runtime, compared to the time the new model is not employed. In other words, when the optimizer is aware of the benefit of the I/O parallelism on SSD, it can make better optimization decisions. We also discuss how the proposed model can be initialized, calibrated and used efficiently.

The remainder of the paper is organized as follows. Section 2 reviews the database access methods and their I/O patterns. Section 3 studies the impact of exploiting parallel I/O in access methods over different configurations. Section 4 describes our proposed I/O cost model in details and shows the impact of the model on query optimizer in SAP SQL Anywhere and Section 5 concludes the paper.

2. DATABASE ACCESS METHODS

Choosing the optimal access method for a given query is one of the fundamental and classic problems in database systems. This problem has been studied since 1970 [9, 22]. Index scan and full table scan (hereafter, IS and FTS, respectively) are two traditional access methods which are implemented in all database systems. When a relevant index is available, the database engine traverses the index to find and fetch only the required rows that satisfy the given scan predicate. Unlike IS, in FTS all rows within a table are fetched and scanned one-by-one to find those rows which satisfy the given predicate.

FTS suffers from unnecessary I/Os because it must read all table pages, no matter whether they are relevant or not. In addition, for any retrieved page, all rows within the page must be evaluated against the given predicate. This results in execution of extra CPU instructions. This problem gets more important when there are a large number of rows which are rejected by the predicate. However, FTS benefits extensively from an efficient sequential I/O pattern. Unlike FTS, in IS, with the guidance of an index, only relevant table pages are fetched. In addition, it is not necessary to scan all rows within every fetched page. Therefore, the number of CPU instructions executed for an index scan could be significantly smaller. However, index scan suffers from an expensive random I/O pattern. Due to the substantial disparity between sequential and random I/O in hard disk drives, the significantly more expensive I/O in index scan plays an important role in preferring FTS over IS over a large selectivity range. Another disadvantage of the index scan is that in this method when the selectivity is large, not only the entire table pages might be fetched but also it is very likely that the same table pages be retrieved over and over again. When the memory buffer pool is small this

Figure 2: Parallel full table scan (PFTS) in SAP SQL Anywhere. Each color represents a different worker.

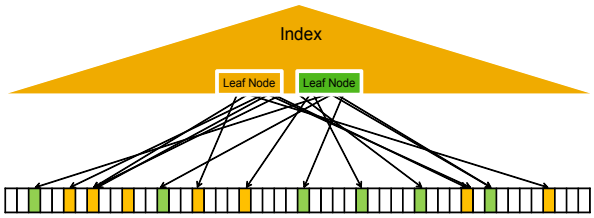


Figure 3: Parallel index scan (PIS) in SAP SQL Anywhere. Each color represents a different worker.

repetitive retrievals result in extra disk I/Os. Therefore, for a large enough selectivity the total number of pages fetched using IS can be potentially even more than the number of pages fetched using FTS.

Traditionally, it is known that the selectivity break-even point between IS and FTS occurs at around 10 percent [20]. It means that when less than 10 percent of the rows in a table satisfy a given predicate, then it is better to use an IS to scan the table. However, recent studies as well as our own experiments show that on todays modern hardware this number is much smaller than 10 percent. The selectivity break-even point depends on two major parameters: the size of memory buffer pool, and the number of records per page. Having larger memory buffer pool improves the performance of the index scan as it is more likely that the re-referenced table pages can be fetched from memory buffer pool rather than disk. If the number of rows per page increases (or equivalently the row size gets smaller) the performance of index scan becomes worse, hence, the break-even point shifts toward smaller selectivities. When a page contains only a single row, the number of rows retrieved is equal to the number of fetched pages. However, as the number of rows per page increases, even at small selectivity, the number of pages that must be fetched quickly approaches 100% of the table pages. When the available memory is small, this number goes beyond 100% of table as some pages might be fetched multiple times. In [26] an analytical formula for the expected number of pages retrieved given the size of the table, number of rows per page, and selectivity is proposed. Many commercial optimizers use their own formulas for the cost model of IS and FTS.

Some modern database management systems like SAP SQL Anywhere support intra-query parallelism [10, 25]. Intra-query parallelism involves having more than one CPU core handle a single query simultaneously, so that portions of the query are computed in parallel on multi-processor hardware. Parallel full table scan and parallel index scan (hereafter, PFTS and PIS, respectively) are two basic algebraic operators that can execute in parallel in SAP SQL Anywhere. Parallel hash join, parallel nested loop join, parallel hash filter and parallel hash groupby are some other operators in SQL Anywhere for which intra-query parallelism is supported. The focus of our study is on PIS and PFTS.

Fig. 2 and Fig. 3 show a schematic view of PFTS and PIS operators, respectively. In PFTS, each worker fetches a table page and starts processing the rows within that page. The workers fetch table pages one by one. As soon as a worker finished processing all rows within a page, it fetches the next available page and starts processing it. A prefetching mechanism is employed that guarantees prefetching up to n blocks ahead of the page which is being currently processed, asynchronously. Therefore, when a worker requests the next available page the page may be already in memory buffer pool. To improve the I/O performance, instead of prefetching pages one by one a large block consists of several consecutive pages is read at a time. In practice, since full table scan is a CPU intensive operator, by employing the prefetching mechanism, usually the CPU computations do not block for I/O, unless the maximum sequential I/O bandwidth of the underlying storage device becomes a bottleneck. Note that the I/O pattern of the table scan is sequential. Thus, even in a commodity hard drive the throughput of the I/O in table scan is very reasonable.

In PFTS, especially on SSD, the I/O queue depth would be usually smaller than the number of workers. That is because in PFTS the speed of processing pages is usually lower than the speed of fetching pages from disk into memory. In other words, to increase the I/O queue depth more CPU cores are required.

In PIS, one worker traverses the index from root to leaf level and finds the range of leaf pages which must be accessed. Then, leaf pages are retrieved and processed by multiple workers one by one. Each leaf page consists of (key, row_id) tuples. Each thread, after retrieving an index leaf page, goes over all row_ids one by one and retrieves corresponding table page for each row_id. By profiling the I/O queue depth of the SSD during the execution of the PIS operator using n workers, a queue depth of n is clearly observable. Since the time interval between issuing consecutive I/O request by each worker is much shorter than the I/O latency of the storage device, at any point of time, during the execution of PIS, the number of outstanding I/Os would be n. This pattern is observable in all cases except in very selective queries in which the number of leaf pages which are required to be retrieved is smaller than the number of workers. Thus, the I/O pattern of PIS with parallel degree n is the parallel random I/O with constant queue depth of n.

3. CHARACTERIZING THE IMPACT OF I/O PARALLELISM IN SCAN OPERATORS

In this section we characterize the impact of using I/O parallelism in scan operators in SAP SQL Anywhere.We will demonstrate how the query optimizer choice for the best access method is affected by the exploitation of the parallel I/O by the IS, FTS, PIS, and PFTS access methods. In particular, we explore the magnitude of the shift from non-parallel to parallel selectivity break-even points in different configurations. Non-parallel and parallel selectivity break-even points refer to particular points in the selectivity range in which the runtime curve of the IS access method crosses the FTS runtime curve, and the runtime curve of the PIS access method crosses the PFTS runtime curve, respectively.

The main goal of the experiments presented in this section is to show why the query optimizer must be aware of the benefit of I/O parallelism for the underlying storage device. As discussed in Sec. 2, intra-query parallelism in PIS and PFTS is one way to increase the I/O queue depth. We will also demonstrate how an effective prefetching mechanism can increase the I/O queue depth in index scans.

Table 1: Experimental configurations

Experiment	Table	Rows per page	Device
E1-HD	T1	1	HDD
E1-SSD	T1	1	SSD
E33-HDD	T33	33	HDD
E33-SSD	T33	33	SSD
E500-HDD	T500	500	HDD
E500-SSD	T500	500	SSD

3.1 Experimental Setup

As mentioned in Sec. 2, the number of rows per page and the size of the available memory buffer pool are two important factors in determining the position of the selectivity break-even point. In order to consider the impact of the number of rows per page, we performed three sets of experiments. The first set of experiments will be performed on a table in which there is only a single row in each table page. This table is named T1. T1 represents an extreme case which is not common in practice but it is a good candidate for observing the impact of very large row size on performance of access methods. The second set of experiments will be performed on a table in which there are 33 rows in each table page. This table is named T33. T33 represents a typical case. The third set of experiments will be performed on a table which has 500 rows per page. This table is named T500. T500 represents another extreme case which is not very common in real workloads but it is a good candidate to study the impact of very small row size in performance of access methods. Each set consists of 2 experiments. The first one is done on HDD and the second one is done on SSD. In order to factor out the impact of memory buffer pool a very small memory buffer pool with size 64MB is used in all experiments. The configuration summary of all experiments has been listed in Table 1.

In all experiments the following query is used:

```
Q:  SELECT MAX(C1) FROM Ti
    WHERE C2 BETWEEN low AND high
```

where *low* and *high* are used to control the selectivity. Tables T1, T33, and T500 include columns C1 and C2 plus some additional columns. The additional columns are used as padding to adjust the target row size. A non-clustered index is created on C2. No index is created on C1. The data type of all columns is integer and the inserted values in each column follow a uniform random distribution. All experiments have been done on a machine with a quad-core Xeon W3530 2.80GHz CPU (with hyper-threading enabled), a 7200 RPM hard disk drive, and a consumer level PCIe SSD drive. The maximum advertised sequential throughput for read and write of the SSD drive are about 1.5GB/s and 1.3GB/s, respectively. The maximum random throughput for read and write of our SSD drive are 230K IOPS and 140k IOPS, respectively. Note that these numbers are the maximum possible numbers while in reality the random performance depends mainly on parameters like queue depth, band size (physical address range in which the random I/Os are issued), compressibility of data, rate of I/O and the size of free space on SSD. In all our experiments the data is random (non-compressible). Band size and queue depth are the variable parameters in our experiments. Free space of the SSD does not have any impact on our experiments as there is no random writes in our experiments.

Some databases support a variation of index scan in which before fetching table pages, row identifiers are sorted in the order of page id [6, 24]. In this way, each table page will be fetched at most once. This access method is usually called sorted index scan. Although this method does not preserve the index key order and needs an additional sorting stage but it can be the optimal choice in a particular selectivity range. Since SAP SQL Anywhere does not support this operator, we could not consider it in our experiments. However, the T1-related experiments will partially make up for this omission.

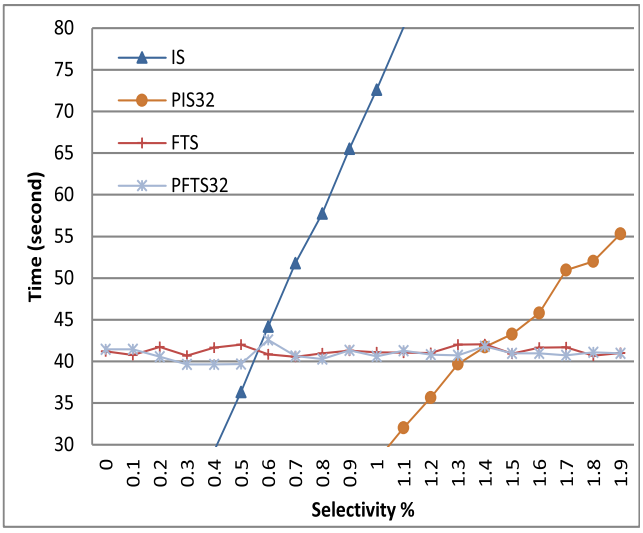
3.2 Experimental Results

In Fig. 4, in all diagrams (a) to (f) the x-axis represents a selectivity range. Note that the selectivity range and the scale of the selectivity range in each diagram is different; the selectivity range was chosen to contain all break-even points for that specific experiment. In each diagram the y-axis represents the total runtime of the execution of the query Q. Each curve is related to the execution of the query with a different access method. PISi and PFTSi refer to PIS and PFTS with a parallel degree of i, respectively. To improve the readability of the diagrams, the curves related to parallel degrees 2, 4, 8, and 16 are emitted from all diagrams.

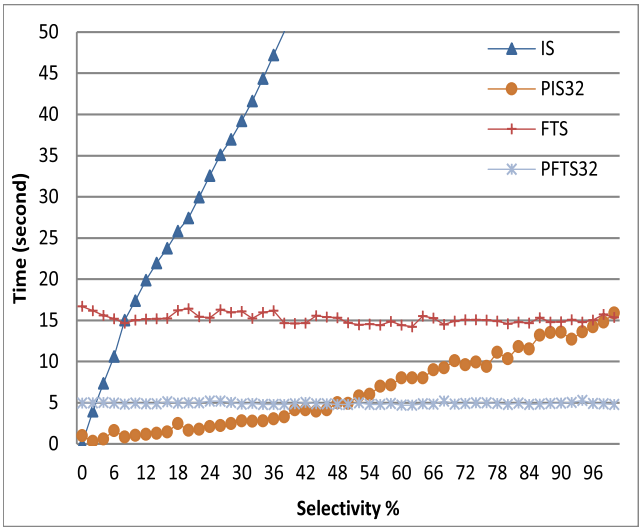
Fig. 4(a) shows the results of the experiment E1-HDD (see Table 1). As mentioned before, this experiment represents an extreme case in which there is only a single large row per table page. Increasing the parallel degree gives an improvement in runtime; however, the amount of this improvement is only moderate. In average the runtime of PIS32 is about 2.37 times faster than the runtime of IS (non-parallel IS). Increasing the parallel degree has almost no effect on performance of PFTS. As expected, HDD does not show any performance improvement with parallel I/O.

Fig. 4(b) shows the results of the same experiment on SSD, E1-SSD. Increasing the parallel degree results in a significant runtime improvement in PIS. The runtime of PIS32 is on average 16.6 times faster than the runtime of IS. For some selectivity ranges this ratio goes up to 21.6 times. Even in FTS, increasing the parallel degree results in improving the runtime: the performance of PFTS32 is in average 3.5 times better than that of FTS. In general, FTS is a CPU intensive operator. However, in this extreme case, there is only one row per page. Therefore, by reading every table page only one row must be processed. That is why even by increasing the parallel degree to a number larger than 8, which is the number of logical processor cores in our machine we can still see some improvements in performance. Unlike in E1-HDD in which we can see a small shift from the non-parallel break-even point to the largest parallel break-even point, i.e., from 0.55% for IS/FTS, to 1.4% for PIS32/PFTS32, in E1-SSD the magnitude of this shift is significantly larger. In E1-SSD the break-even point shifts from 8% to 48%. This shows how important the impact of parallel I/O could be in deciding between PFTS and PIS. Note that, in all experiments, whenever we speak about parallel break-even point, we are referring to the crossing point of the curves PIS32 and PFTS32.

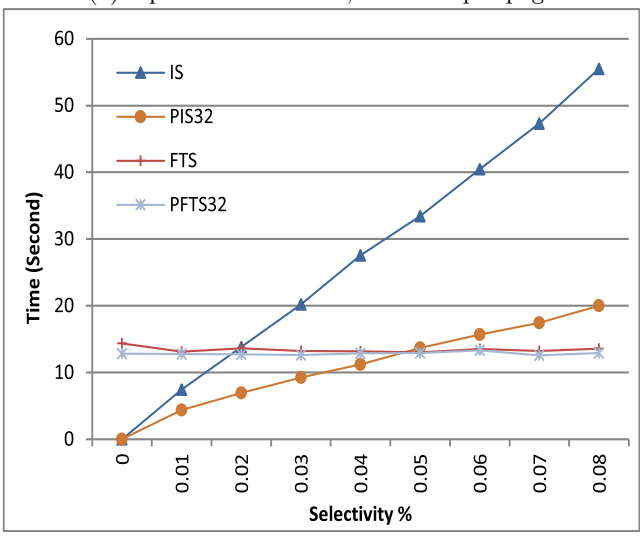
Fig. 4(c) shows the results of experiment E33-HDD. This experiment has been performed on HDD and represents a typical number of rows per page, in this experiment 33. In average only 2.5 times improvement is observable when we compare PIS32 with IS. Here again increasing the paral-



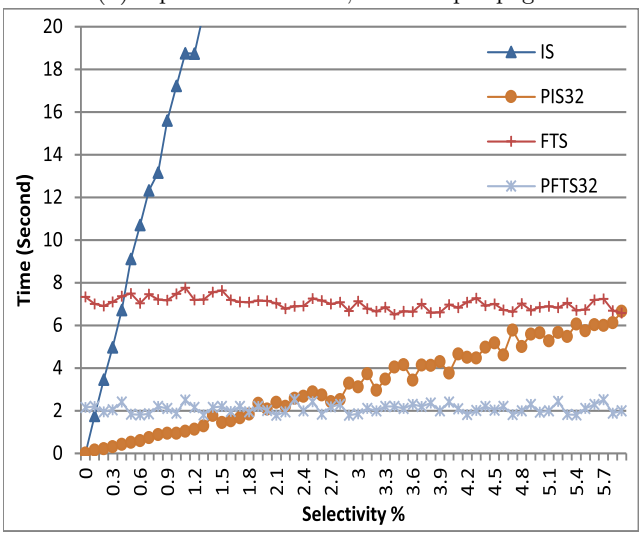
(a)Experiment E1-HDD, one rows per page



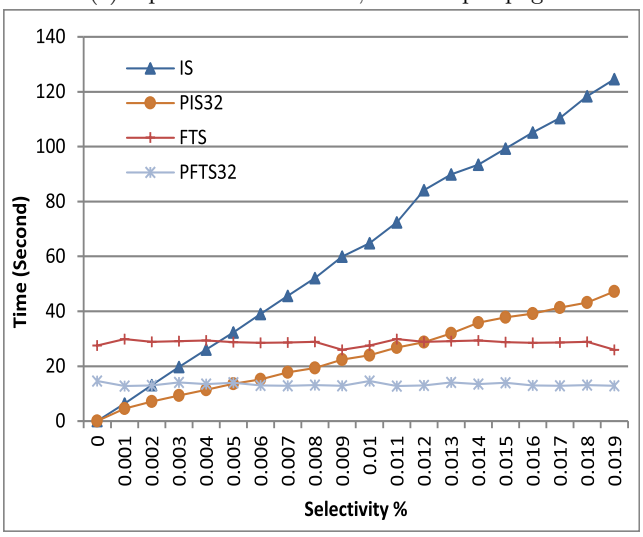
(b)Experiment E1-SSD, one row per page



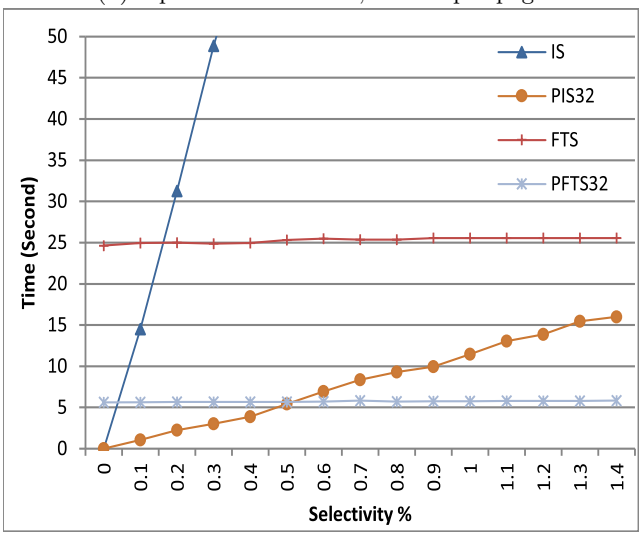
(c)Experiment E33-HDD, 33 rows per page



(d)Experiment E33-SSD, 33 row per page



(e)Experiment E500-HDD, 500 rows per page



(f)Experiment E500-SSD, 500 row per page

Figure 4: Runtime of query Q using the IS, FTS, PIS and PFTS access methods over tables T1, T33 and T500 on HDD and SSD

lel degree in FTS does not show any improvement. Both parallel and non-parallel break-even points are significantly smaller than their counterparts in E1-HDD. The magnitude of shift in break-even point is also much smaller, i.e., from 0.02% for IS/FTS to 0.05% for PIS32/PFTS32.

Fig. 4(d) shows the results of the experiment E33-SSD. This experiment is the same as the previous experiment but on SSD. A very significant improvement from IS to PIS32 is clearly observable. The average improvement is about 19.9 times with a maximum of 34 times in some specific selectivity ranges. Unlike on HDD, on SSD, even in PFTS, increasing the parallel degree results in better performance. We observed that PFTS8 is about 3.13 times faster than FTS in average. Increasing parallel degree to a number larger than 8 does not have any impact on performance. That is because in this experiment we have 33 rows per table page. Therefore, FTS is much more CPU intensive than in E1-SSD. Thus, increasing the parallel degree to a number larger than the number of logical cores would not be helpful anymore.

Fig. 4(e) shows the result of the experiment E500-HDD. This experiment represents another extreme case in which there are a large number of rows per table page. It is expected that in a case like this we observe a very CPU intensive FTS and a very small break-even point. Similar to E33-HDD, in E500-HDD the average improvement of PIS32 over IS is only about 2.5 times. However, unlike in E33-HDD, in this experiment PFTS2 is about 2 times faster than FTS. No more improvement is observable by increasing the parallel degree to a number larger than 2. In this experiment FTS is extremely CPU intensive because a large number of rows, i.e. 500, must be processed after reading every page. Increasing the parallel degree from 1 to 2 gives FTS more CPU power. However, when parallel-degree is 2, the maximum bandwidth of HDD (110MB/s) is already saturated. Therefore, the CPU must wait for I/O. That is why a parallel degree larger than 2 will not help more. In E33-HDD, since the number of rows per page was much smaller than in E500-HDD, even a single CPU core was able to saturate the maximum bandwidth of HDD. Therefore, in E33-HDD we cannot observe any improvement from I/O parallelism. As we expected in this extreme case, both parallel and non-parallel break-even points are extremely small (0.005% and 0.0045%, respectively).

Fig. 4(f) shows the results of the experiment E500-SSD. The average improvement of PIS32 over IS in this experiment is 22.5 times which is even larger than in E33-SSD. Compared to E1-SSD and E33-SSD, in E500-SSD we observed a larger average improvement of PFTS8 over FTS. PFTS8 is in average 4.4 times better than FTS. However, by increasing the parallel degree to a number larger than 8, the PFTS will need more logical CPU cores to be able to saturate the maximum bandwidth of SSD. In this experiment the maximum utilized bandwidth of SSD is only about 250MB/s. This number is much smaller than the maximum bandwidth of our SSD (1.5GB/s). It shows that in this extreme case by increasing the number of CPU cores a smaller runtime would be easily possible. It is expected that even in E33-SSD increasing the number of CPU cores will result in better performance in PFTS. The maximum utilized bandwidth of the SSD in E33-SSD is about 581MB/s.

Table 2 shows the summary of shifts in break-even point in different experiments. As you see the magnitude of shift exposed by SSD is much bigger than that exposed by HDD. For parallel break-even points the crossing point of the PIS and PFTS with maximum parallel degree, i.e., 32, is reported. We repeated all the mentioned experiments with a large memory buffer pool as well and again we observed that the magnitude of shift exposed by SSD is much bigger than that exposed by HDD. Compared to small-memory experiments, in large-memory experiments in some cases, e.g. in RPP=500, the amount of shift in break-event point on SSD even increases by a factor of 2 while on HDD it remains almost constant. This shows that larger memory buffer pool even amplifies the magnitude of shift in break-even point on SSD. At the beginning of each experiment we flushed the memory buffer pool to factor out the impact of pages which are already in memory. Due to the lack of space we omitted the details of the repeated experiments from the paper.

Table 2: Summary of shifts from non-parallel to parallel break-even points on HDD and SSD in different experiments. NP- refers to the crossing point of IS and FTS, and P- refers to the crossing point of PIS32 and PFTS32

Rows per page	NP-HDD	P-HDD	NP-SSD	P-SSD
1	0.55%	1.4%	8%	48%
33	0.02%	0.05%	0.4%	2.1%
500	0.0045%	0.005%	0.15%	0.5%

Table 3: Summary of I/O throughput in PFTS32 and FTS over different experiments.

	PFTS32 I/O throughput	FTS I/O throughput
E1-HDD	100.45MB/s	96.80MB/s
E1-SSD	849.25MB/s	263.33MB/s
Ratio	8.45X	2.72X
E33-HDD	106.47MB/s	100.59MB/s
E33-SSD	581.46MB/s	192.16MB/s
Ratio	5.46X	1.91X
E500-HDD	110.94MB/s	50.77MB/s
E500-SSD	250.69MB/s	57.63MB/s
Ratio	2.25X	1.13X

Table 3 summarizes the average I/O throughput of PFTS32 and FTS on SSD and HDD. As mentioned before, full table scan is a CPU intensive operator. By increasing the number of rows per page, the number of CPU instructions needed to process a single page would be increased. In other words, when the number of rows per page is small, CPU can quickly process a page and request the next page. In this case the CPU must wait for I/O because the processing time would be smaller than I/O latency. In contrast, when there are many rows per page, the time it takes for CPU to process all those rows would be longer than the time it needs to fetch the next page. In this case, I/O must wait for CPU. Therefore, the I/O throughput of the underlying storage device can potentially be underutilized.

In Table 3 the maximum I/O throughput of SSD can be observed in PFTS in E1-SSD. In this extreme-case experiment, since there is only one row per page, the CPU can better utilize the I/O throughput of SSD. By increasing the number of rows per page, as you see in E33-SSD and E500-SSD, the I/O throughput of SSD would be reduced. That is because we do not have enough CPU power to keep the

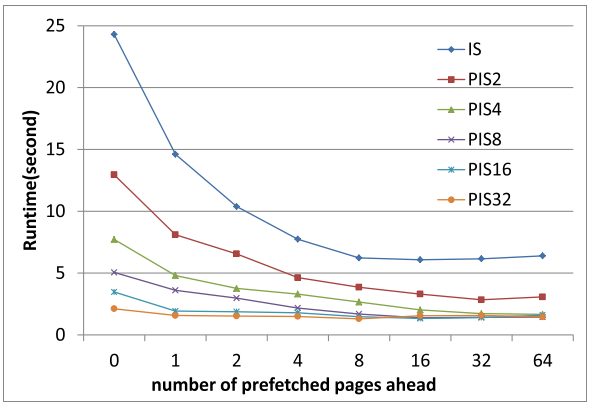


Figure 5: Index scan runtime with different parallel degrees when prefetching is enabled in each worker. Each curve represents a different parallel degree

number of outstanding I/Os in I/O queue of SSD high. By increasing the number of CPU cores or by using more powerful cores the I/O throughput of SSD in PFTS can be improved. By comparing FTS and PFTS on HDD on E1-HDD and E33-HDD no significant improvement in I/O throughput is observable. That is because the maximum possible I/O throughput of our HDD is about 110MB/s. This I/O throughput can be achieved using a single core. Therefore, increasing the parallel degree on HDD does not make any improvement. However, in E500-HDD it can be noticed that the I/O throughput of PFTS on HDD is almost 2 times better than that of FTS. That is because in E500-HDD the PFTS is extremely CPU intensive. Therefore, a single core can utilize up to only about 50MB/s of the I/O throughput. In order to utilize the maximum capacity of the I/O throughput we need at least one additional core. After that, the maximum capacity is reached. Thus, adding more cores would not have any further impact.

3.3 Employing Prefetching

So far we showed that we can increase the I/O queue depth in index scans using intra-query parallelism. However, worker threads are limited resources and synchronization of threads introduces extra overhead. A non-parallel plan is usually preferable to a parallel plan when their estimated costs are close to each other. As mentioned before, index scan is not CPU intensive because of the low performance of random I/O. While we would like to generate a high queue depth to exploit I/O parallelism, doing so with a large number of worker threads is wasteful. An alternative is to employ asynchronous prefetching. We implemented prefetching in index scan of the SAP SQL Anywhere with each of M workers prefetching up to n pages that are expected to be needed in the near future. The expected peak queue depth is Mn. For the sake of simplicity, we only prefetch table pages referenced by a single index leaf page. As a worker gets closer to the last page which is referenced by a leaf index page, the number of prefetched table pages for that worker is reduced until it moves to another index leaf page. Since the number of (key, row_id) tuples in each index page is typically large, this simplification does not have a large impact in the overall runtime of the index scan.

Fig. 5 shows the impact of prefetching in index scans with different parallel degrees (the number of workers used to execute the PIS). In this experiment a range index scan over a large table with 80M rows has been performed. The number of rows in each page is 33. Each curve represents the execution of the index scan with a different parallel degree. The x-axis indicates n (the prefetch requests issued by each worker) and the y-axis shows the total runtime. In this experiment the selectivity of the predicate is 0.03As you can see in Fig. 5, our proposed prefetching mechanism has a significant impact in improving the runtime of the index scan. However, prefetching n pages with 1 worker does not give the same performance as using n workers due to simplifications in the prefetching implementation which prevent it from achieving an average queue depth of n and imperfect overlapping of I/O and CPU resources. Nevertheless, by combining prefetching with intra-query parallelism we can get the maximum with fewer workers. For example, with only 4 workers and a prefetching degree of 32, we can achieve a performance even 35% better than using 32 workers and no prefetching at all . This gives us an excellent performance without the negative impacts of using a large number of workers. Improving the proposed prefetching mechanism can potentially result in even more improvements. Although index scan can extensively benefit from prefetching, the impact of prefetching in table scan is somewhat limited since, the table scan is a CPU intensive operator and usually more threads are needed to keep up with the higher data rate. The experiments related to the impact of prefetching and using larger block reads in table scan are omitted for space.

3.4 Summary of Conclusions

From the experiments presented in this section we can conclude the following facts.

1. Exploiting parallel I/O can have a significant impact on performance of the database operators. This confirms that the I/O queue depth can play an important role in the I/O cost estimation of database operators.

2. The impact of exploiting parallel I/O is more pronounced for SSD than HDD.

3. Parallel I/O on SSD can result in a considerable shift in selectivity break-even point. Therefore, it is not enough to simply use the parallel version of the existing IS and FTS access methods with a constant prefetching degree. The query optimizer must be aware of the impact of the parallel I/O on the selectivity break-even point and adjust its decision about selecting the optimal access method accordingly.

4. Intra-query parallelism and prefetching are two mechanisms to generate higher I/O queue depths and combination of these methods can result in better improvement with a lower impact in the degree of inter-query parallelism in database server.

4. QUEUE DEPTH AWARE DISK TRANSFER TIME MODEL

From the experimental results in the previous section we observed that the selectivity break-even point faces a significant shift when parallel access methods are employed on SSD. Now, the question is how we can employ a mechanism to make the query optimizer aware of the potential benefit

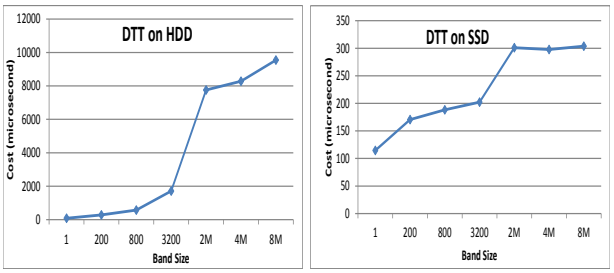


Figure 6: A sample DTT model for HDD and SSD.

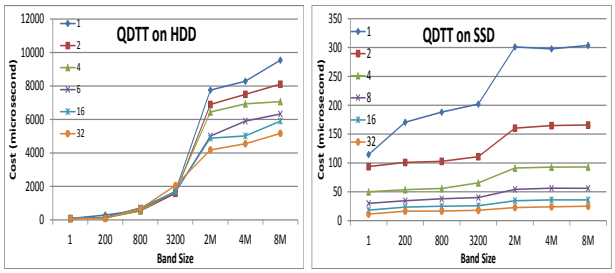


Figure 7: A sample QDTT model for HDD and SSD.

of I/O parallelism and its impact on the I/O cost estimation of the query plans. Based on this knowledge, the cost model should correctly cost parallel access methods such that the optimal access method using the optimal parallel degree would be chosen as the best access method.

4.1 DTT Model

SAP SQL Anywhere employs an I/O cost model called disk transfer time (DTT) model to estimate the cost of I/O [1, 3, 4]. The DTT model is typically calibrated on the customer's specific hardware and it summarizes disk subsystem behavior with respect to an application (in this case, the DBMS). The DTT function models the amortized cost of reading one page randomly over a band size area of the disk. If the band size is 1, the I/O is sequential, otherwise it is random. Band size is basically the size of a disk area (in number of pages) from/to which the random I/Os are going to be issued. In a DTT model calibrated for a hard disk drive, increasing the band size results in significant increase of the I/O cost. When the band size is larger it will span over more cylinders on disk. Therefore, it is more likely that for every retrieval the disk arm needs to be moved from one cylinder to another, resulting in higher overall seek time. By calibrating and using the DTT model, just by knowing the band size in which a database operator is going to perform its I/Os, we can have a fairly accurate estimate of the amortized cost of each individual page I/O. Fig. 6 shows the calibrated DTT model of a hypothetical SSD and HDD. One of the interesting properties of the DTT model is that it can be calibrated easily for any particular hardware at any time. This allows the database optimizer to be adapted to the new hardware at any point of time and perform more accurately after calibration. This eliminates the trouble of using inaccurate hardcoded parameters which are tuned ahead of time and used for any deployment of the database. This is important because SAP SQL Anywhere was designed from the outset to offer self-management features permitting its deployment as an embedded database system.

4.2 QDTT Model

Although the DTT model works very well for modeling the I/O cost of the commodity hard disk drives, it is not accurate enough for modeling the behavior of modern storage devices such as solid state drives. These storage devices will benefit extensively from parallel I/O. Increasing the number of outstanding I/Os in the I/O queue of these storage devices results in a significant improvement in I/O throughput. Therefore, for those database operators which are able to exploit I/O parallelism, the existing DTT model will overestimate the I/O cost because it does not distinguish between parallel and non-parallel I/O. Thus, it is assumed that the cost of parallel I/O is similar to the cost of non-parallel I/O, which is true for commodity hard drives. Consequently, the optimizer will not prefer a parallel plan to a non-parallel plan while the real cost of a parallel plan is much cheaper for solid state drives.

Even if we force the optimizer to always choose a parallel plan over its non-parallel version (assuming that in terms of CPU cost the benefit of parallelism is more than its overhead) it may still choose a suboptimal plan. Suppose for plans P1 and P2 both parallel and non-parallel versions are available. Suppose the optimizer is forced to always prefer a parallel plan over a non-parallel plan but it uses the current DTT model for I/O cost estimation. It is possible that the estimated cost of the non-parallel version of P1 using the DTT model is cheaper than that of P2 while the real cost of the parallel version of P1 is more than the parallel version of P2. In this case the optimizer will choose the parallel version of P1 while the parallel version of P2 is the optimal choice. For example, for a specific selectivity, non-parallel full table scan might be a cheaper access plan compared to non-parallel index scan. However, it is possible that for the same selectivity, parallel index scan be cheaper than parallel full table scan. These inaccurate optimizer decisions are possible because the DTT model does not consider the benefit of doing I/O in parallel at all.

In order to solve this problem, we introduce an extension of the DTT model which considers the I/O queue depth as well as band size. The new model is called queue-depth-aware disk transfer time (QDTT). The DTT model is basically a function that takes the band size as an input parameter and returns the amortized cost of reading randomly a single page within the given band size. Unlike DTT model, the QDTT model is a function that takes two parameters band_size and queue_depth and returns the amortized cost of a random I/O within the given band_size, when the queue depth of the storage device is equal to queue_depth. For database operators which can benefit from I/O parallelism this new model provides a much more accurate estimation of the I/O cost. It is clear that this model will be more beneficial when the data is located on a storage device that can benefit from I/O parallelism (i.e., high queue depth). Solid state drives and multiple-spindle hard disk drives are examples of such devices. Moreover, for single spindle hard disk drives, the QDTT model maintains the same functionality of the DTT model. Therefore, the new QDTT model can be considered as a generalization of the DTT model.

Fig. 7 shows a calibrated QDTT model of an SSD drive and a single spindle HDD drive. Each curve represents a different queue depth. As you see by increasing the queue depth the amortized cost of one I/O (in microsecond) decreases significantly. In a solid state drive there are no moving parts. Therefore, it is expected that band size must not have a considerable impact on I/O performance. How-

ever, it can be seen that in many modern solid state drives the band size is still an important parameter in estimating the cost of random I/O. Nevertheless, this impact is not as serious as what we can see on calibrated models for single-spindle hard disk drives. Moreover, in SSD, as you see, by increasing queue depth, the impact of band size in I/O cost is reduced. In any case, a calibrated QDTT model helps the optimizer to get rid of the internal complexities of the underlying storage subsystem for I/O cost estimation. The only things the optimizer needs to know are band size and queue depth. The QDTT model will take care of the rest.

4.3 Experimenting with QDTT Model

Here we would like to perform some experiments to see the impact of using the new model in the query optimizer. In the cost estimation function of PIS and PFTS operators there is a call to DTT function. The cost estimation function first estimates the band size on disk from which the physical I/O fetches would be issued. This band size would be sent to the DTT model as an input parameter and the DTT model will return the amortized cost of reading a page randomly within the given band size. Then the number of pages needed to be retrieved during the scan is estimated. By multiplying this number with the amortized cost of reading a single page the total I/O cost is calculated. We changed the cost estimation functions of PIS and PFTS such that they use QDTT model instead of DTT model. This time, in addition to band size, parallel degree of the operator would be passed to the model as well. The calibrated QDTT model must know the parallel degree and it will return an accurate cost estimate. By having a more accurate estimation of the I/O cost, the optimizer would be able to make a closer to optimal choice between PIS and PFTS operators. When the optimizer is using DTT model, it would not realize that there might be some benefits (in terms of I/O cost) from executing the operator in parallel. From the perspective of optimizer, the I/O cost of a parallel and non-parallel access method would be similar. Consequently, the optimizer would prefer a parallel access method over a non-parallel access method only in cases in which the CPU cost benefit of doing things in parallel will surpass the overhead of parallelism. By using the new QDTT model, the optimizer will consider the benefit of parallelism not only in CPU cost but also in I/O cost.

Fig. 8 shows the runtime of Q before and after using QDTT model for experiments E1-SSD, E33-SSD and E500-SSD. In each diagram, the *old optimizer* curve represents the runtime of the query when the DTT model is used by optimizer, the *new optimizer* curve represents the runtime of the query when the QDTT model is employed by the optimizer, and the *speedup* curve indicates how many times the query runtime has improved after utilizing the QDTT model. The y-axis in right side of each diagram represents the amount of speedup. As you see by using the new model a significant improvement in runtime of the optimizer is observable. The maximum speedups in E1-SSD, E33-SSD and E500-SSD are 19.7, 16.9, and 13.7, respectively.

The old optimizer uses DTT model and since it does not realize the benefit of parallel I/O it always prefers a non-parallel method over a parallel one for these experiments. Note that in all of the above experiments the optimizer has the knowledge that none of the table pages or index pages is in the memory buffer pool (SQL Anywhere maintains statistics on how many table and index pages are currently

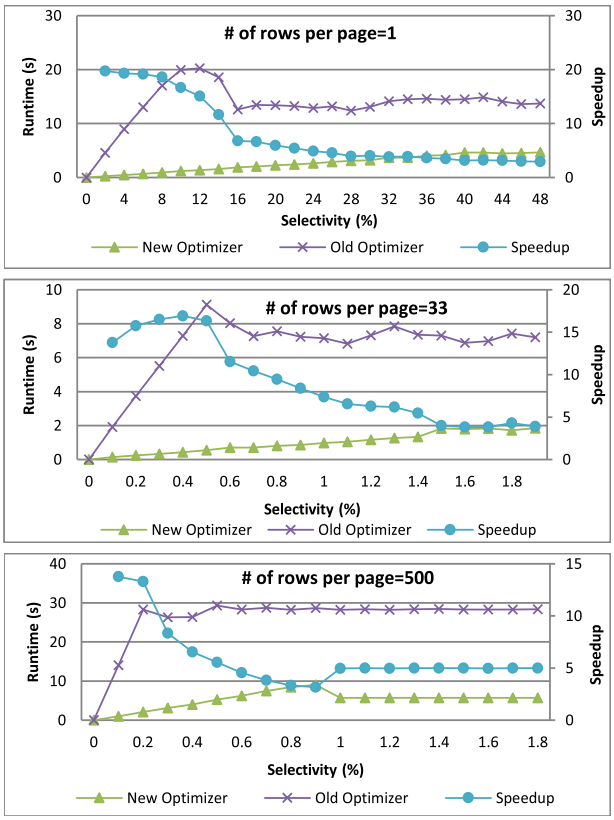
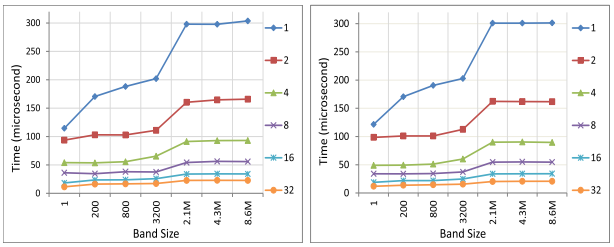


Figure 8: Comparing the performance of DTT-based and QDTT-based optimizers

cached). Since the estimated I/O cost is much more than the estimated CPU cost, and because the parallel plan evolves some extra overhead cost for synchronization and coordination, the CPU benefit of parallel plans does not have any impact on the decision of the optimizer. Therefore, the optimizer prefers a non-parallel plan. By employing the QDTT model, the optimizer would be aware of the benefit of parallel I/O for the SDD used in these experiments. Therefore, parallel plans would be preferred to non-parallel plans based on the correct cost estimations.

In these experiments the maximum allowable parallel degree for a parallel access method is set to 32. In fact after using QDTT in all three experiments a parallel plan with parallel degree 32 is selected. In all three experiments, the amount of improvement for low selectivities is high, then it starts to drop and finally it gets constant. When the amount of improvement gets constant both new and old optimizers are choosing full table scan. By using QDTT, for a very large range of selectivities we can observe at least 3 to 5 times improvement while for a small range of selectivities we can achieve up to 20 times improvement.

In all the performed experiments a single query is running on the system at a time. Therefore, in all cases, in addition to the estimated band size, the optimizer simply needs to pass the maximum beneficial queue depth(here 32), to the QDTT model. When multiple queries are running on the system concurrently, the optimizer needs to pass a lower queue depth number to the QDTT model. The optimal decision of the optimizer about the queue depth parameter depends on the concurrency level of the system and the type of database operators in the query plans. Studying the role



(a) GW method (b) AW method

Figure 9: Calibrated QDTT model using GW (a) and AW (b) methods on SSD

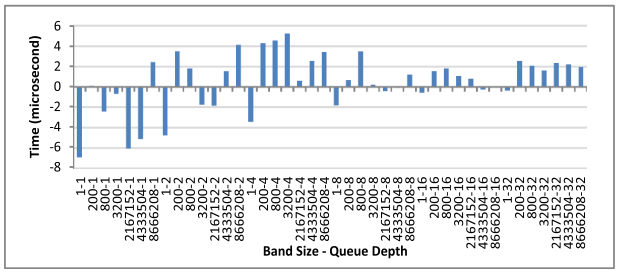


Figure 10: Differences between the costs computed by AW and GW methods on SSD

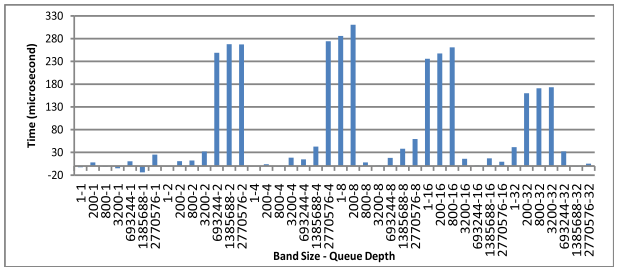


Figure 11: Difference between the costs computed by AW and GW methods on RAID (8-spindles)

of these factors in choosing the proper queue depth is out of the scope of this paper and is considered as a future work.

4.4 Calibrating QDTT Model

In order to calibrate the QDTT model for a band size of b and queue depth of qd we need to measure the amortized cost of a single page I/O, when the I/Os are randomly issued, within a band size of b when the average queue depth is qd. The amortized cost of a single page I/O will then be calculated by dividing the total measured I/O time by the number of issued I/Os. One way of increasing the I/O queue depth is to use multiple threads. Each thread issues a synchronous page I/O and as soon as that synchronous I/O finished, it issues another synchronous I/O. Since the pages are just read, the total CPU time for processing a page is almost zero; hence the CPU time compared to the I/O latency is negligible. Therefore, by using n threads we can keep the average I/O queue depth constantly equal to n.

Another method for increasing queue depth is using a single thread and prefetching pages using asynchronous I/Os. In order to use prefetching to increase the I/O queue depth we have proposed two methods. The first method is called group waiting (GW). In GW, at first n I/O requests would be issued asynchronously. Then the thread would wait for all of them to finish. As soon as all of them are finished, another group of n asynchronous I/Os would be issued. This process would be continued until all pages which are scheduled to be read from within a band size of b are read.

The second proposed method is called active waiting (AW). In AW, n buffers are used. At first a group of n asynchronous I/Os are issued by the thread into the buffers 1 to n. Then, the thread would wait for the I/O associated to the first buffer to finish. As soon as that I/O was finished, the thread issues another asynchronous I/O into buffer 1 and then immediately waits for the I/O associated to the buffer 2 to finish. As soon as the I/O associated to buffer 2 was finished, the thread issues another asynchronous I/O into buffer 2 and then immediately waits for the I/O associated to the buffer 3. This process continues until the I/O associated to nth buffer is finished. Then, the thread issues another I/O into buffer n and then waits for associated I/O to buffer 1 to finish. This circular process continues until all pages are read.

In order to reduce the calibration time for any given calibration point we need to restrict the number of page reads during the calibration. The maximum number of pages read during the calibration is defined by a threshold called M. Currently M is equal to 3200 pages. No matter how big the file or band size are, the total number of page reads for any calibration point would be at most equal to M.

If b is smaller than M, then the file would be divided into multiple blocks of size b. In this case the number of blocks would be equal to $\min(M, (numberofpagesinfile)/b)$, and the number of reads per block would be equal to b.

If b is larger than M then we will have only a single block with size b. In this case, the number of reads from that single block is equal to M and the block size is b and the starting offset of reads is selected randomly. Note that before starting to read from each block a sequence of P non-repetitive random numbers from 0 to b are generated. P is equal to the number of pages read from the block and in practice it is equal to the minimum of b and M. The I/Os would be issued to each block in the same order as the random numbers in the generated sequence. When there are multiple blocks the I/Os would be issued to one block at a time until all blocks are processed.

Fig. 9 (a) and (b) show the calibrated QDTT model on SSD based on GW and AW methods, respectively. In each diagram, each point represents the average of 50 repetitions of the calibration process for that point. It can be clearly seen that the times obtained using GW and AW methods are very similar.

Fig. 10 shows the differences between the costs computed by AW and GW methods on SSD. The maximum observed difference is about 7 microseconds. Compared to standard deviations in each individual method which in some points are up to 40 microseconds, the difference between the costs in the diagrams generated by AW and GW methods is quite negligible. We can conclude that the AW and GW methods show very similar output on SSD. Therefore, either of them can be used for calibrating the QDTT model on SSD.

The similarity between the AW and GW methods on SSD cannot be observed on an HDD. On HDD, AW calibration points are generally smaller than their counterparts obtained using the GW method. Fig. 11 shows the differences between costs obtained by GW and AW methods on a RAID array with eight 15000RPM drive. As you see this time AW shows significantly smaller costs.

The differences between the results using the GW and AW methods on SSD and those obtained by the same methods on HDD can be explained as follows. On SSD, the I/O latency is much lower than on HDD. In addition, due to the SSD high parallel I/O capability, the latency of non-parallel and parallel I/Os (up to a particular parallel degree) are almost the same. For example, the maximum beneficial parallel degree of our SSD is 32. On this SSD, if we issue 32 I/Os, the latency of all 32 I/Os would be almost the same. In other words, on SSD, up to a particular parallel degree, increasing the queue depth does not have any considerable negative impact on I/O latency. Therefore, when using the GW method we wait for the first I/O in a group to be finished, after finishing that I/O, the next I/Os in the group are already finished as well. Therefore, waiting time for the next I/O would be almost zero. Although we wait for the next I/Os, when using the GW method, the waiting time is almost zero. Thus the GW will behave like the AW method on SSDs. On HDD, by increasing the queue depth, although the throughput improves, the latency increases. Therefore, in that case the GW method will not behave like the AW method. Thus, GW would not be able to simulate an I/O queue depth equal to the size of the group. Therefore, in a general calibration method which is supposed to be used on all devices, the AW method must be the method of choice.

4.5 Bilinear Interpolation

Calibrating the QDTT model for all queue depths from 1 to 32 will increase the calibration time substantially. This would be a more serious issue when the model is calibrated on a HDD drive. Therefore, we need to somehow reduce the number of calibration points and use a proper interpolation method to estimate the value of non-calibrated points.

For the original DTT model, a linear interpolation method is used to calculate the cost associated to band sizes for which there is no calibration point in the computed model. We decided to use the same approach to calculate the cost associated with the queue depths for which there is no calibrated point in the QDTT model. Namely, we will first interpolate linearly on the band size and then on the queue depth. This method is also known as bilinear interpolation.

Now, the question is, from 1 to 32, which queue depths can be used by the linear interpolation more accurately. We assumed that queue depths 1, 2, 4, 8, 16, and 32 are the best candidates. In other words, increasing the queue depth in an exponential fashion during calibration will result in a reasonably accurate model for which bilinear interpolation will be used for missing points. To prove the validity of our assumption we performed an extensive set of experiments on different drives.

In Fig. 12 each diagram represents the cost of a random read for a given band size over different queue depths on an 8-spindle RAID. The red square-shaped points represent the observed I/O times associated to queue depths 1, 2, 4, 8, 16 and 32. As you see, calibrating for these points and employing linear interpolation for other points (blue diamond points in Fig. 12) is a fairly accurate approach. This confirms the validity of our assumption. We repeated the same experiments on RAID arrays of different sizes as well as on SSD and observed the same results. The only exception is a single 7200RPM HDD drive. As discussed in Sec. 4.6, for this drive there is no need for interpolation as the calibration will be done only for queue depth 1.

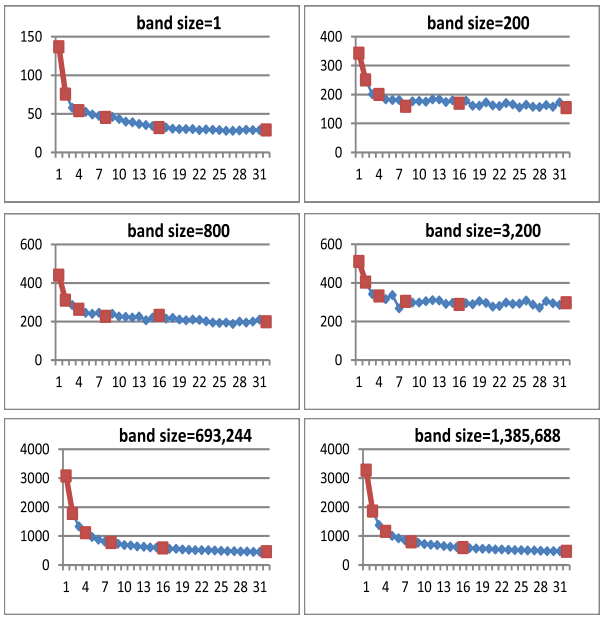


Figure 12: QDTT on RAID (8 spindles). The x-axis represents the queue depth and the y-axis represents the cost of reading a single page in microsecond

4.6 Improving the Calibration Time

Compared to DTT model in the QDTT model calibration time is much longer. That is because there are more calibration points in the QDTT model. For devices which cannot benefit from parallel I/O, performing calibration for high queue depths is pointless. That is because the optimizer will never use the costs associated to those high queue depths.

In order to reduce the calibration time we can take advantage of the mentioned fact and propose a control mechanism that stops calibration when continuing calibration is not beneficial. The mechanism works as follows. The calibration starts from queue depth 1. After calibrating all band sizes for queue depth 1 the queue depth is doubled and the calibration will be performed for queue depth 2. For each queue depth the calibration is done from the largest to the smallest band size. After that the calibration of the largest band size in queue depth 2 is finished, we check the calibration point associated to the largest band size in queue depths 1 and 2. If increasing queue depth has resulted in at least T percent improvement we will continue the calibration. Otherwise, the calibration will be stopped and a default value slightly larger than the measured costs for queue depth one is assigned to the remaining calibration points. If the calibration did not stop, after calibrating the largest band size in next queue depth we check the stop condition again. This approach results in a significant improvement in calibration time especially for devices with weak parallel I/O capability.

Another benefit of this approach is that it adjusts the calibration runtime dynamically based on the parallel I/O capability of the device. We found experimentally that 20 is a reasonable value for T. Reducing the calibration time also increases the feasibility of automatic frequent calibrations during the idle I/O cycles of the system. Investigating the possibility of automatic frequent calibrations during the idle I/O cycles of the system is an interesting avenue for future research.

5. CONCLUSIONS

Previous work (outlined in Section 1) has investigated query execution for data stored on SSDs; however, the impact on query optimization has not been sufficiently explored. In this paper we showed how important it is for the query optimizer to be aware of the benefit of I/O parallelism and we proposed a practical approach to make the query optimizer parallel I/O aware. The QDTT model can be efficiently calibrated and it allows the optimizer to accurately choose among execution alternatives on a range of storage devices.

In the current paper we consider choosing between IS, FTP, PIS and PFTS. Investigating the behavior of more complex database operators and more complex queries is an interesting topic for further research, as is consideration of concurrent requests.

6. REFERENCES

[1] M. Abouzour, I. T. Bowman, P. Bumbulis, D. DeHaan, A. K. Goel, A. Nica, G. N. Paulley, and J. Smirnios. Database self-management: Taming the monster. *IEEE Data Eng. Bull.*, 34(4):3–11, 2011.

[2] D. Bausch, I. Petrov, and A. Buchmann. On the performance of database query processing algorithms on flash solid state disks. In *Database and Expert Systems Applications (DEXA), 2011 22nd International Workshop on*, pages 139–144. IEEE, 2011.

[3] I. T. Bowman, P. Bumbulis, D. Farrar, A. K. Goel, B. Lucier, A. Nica, G. N. Paulley, J. Smirnios, and M. Young-Lai.

[4] I. T. Bowman, P. Bumbulis, D. Farrar, A. K. Goel, B. Lucier, A. Nica, G. N. Paulley, J. Smirnios, and M. Young-Lai. Sql anywhere: An embeddable dbms. *IEEE Data Eng. Bull.*, 30(3):29–36, 2007.

[5] 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. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 266–277. IEEE, 2011.

[6] J. Cheng, D. Haderle, R. Hedges, B. Iyer, T. Messinger, C. Mohan, and Y. Wang. An efficient hybrid join algorithm: a db2 prototype. In *Data Engineering, 1991. Proceedings. Seventh International Conference on*, pages 171–180, Apr 1991.

[7] J. Do and J. M. Patel. Join processing for flash ssds: remembering past lessons. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 1–8. ACM, 2009.

[8] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging dbms buffer pool using ssds. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1113–1124. ACM, 2011.

[9] P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang. Query optimization in the ibm db2 family. *IEEE Data Eng. Bull.*, 16(4):4–18, 1993.

[10] G. Graefe. Volcano-an extensible and parallel query evaluation system. *Knowledge and Data Engineering, IEEE Transactions on*, 6(1):120–135, 1994.

[11] J. Gray and B. Fitzgerald. Flash disk opportunity for server applications. *Queue*, 6(4):18–23, 2008.

[12] W.-H. Kang, S.-W. Lee, and B. Moon. Flash-based extended cache for higher throughput and faster recovery. *Proceedings of the VLDB Endowment*, 5(11):1615–1626, 2012.

[13] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. *Proceedings of the VLDB Endowment*, 1(1):514–525, 2008.

[14] I. Koltsidas and S. D. Viglas. Data management over flash memory. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1209–1212. ACM, 2011.

[15] E.-M. Lee, S.-W. Lee, and S. Park. Optimizing index scans on flash memory ssds. *ACM SIGMOD Record*, 40(4):5–10, 2012.

[16] S.-W. Lee, B. Moon, and C. Park. Advances in flash memory ssd technology for enterprise database applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 863–870. ACM, 2009.

[17] 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 the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086. ACM, 2008.

[18] X. Liu and K. Salem. Hybrid storage management for database systems. *Proceedings of the VLDB Endowment*, 6(8):541–552, 2013.

[19] S. Pelley, T. F. Wenisch, and K. LeFevre. Do query optimizers need to be ssd-aware? *ADMS'11*, 2011.

[20] R. Ramakrishnan and J. Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2000.

[21] 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. *Proceedings of the VLDB Endowment*, 5(4):286–297, 2011.

[22] 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 the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[23] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 59–72. ACM, 2009.

[24] P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, June 1987.

[25] Y. Wang. Db2 query parallelism: Staging and implementation. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 686–691. Morgan Kaufmann Publishers Inc., 1995.

[26] P. Yue and C. Wong. Storage cost considerations in secondary index selection. *International Journal of Computer & Information Sciences*, 4(4):307–327, 1975.