Towards Efficient Concurrent Scans on Flash Disks[⋆]

Chang Xu, Lidan Shou, Gang Chen, Wei Hu, Tianlei Hu, and Ke Chen

College of Computer Science, Zhejiang University, China
chinawraith@163.com, {should,cg,hw,htl,chenk}@zju.edu.cn

Abstract. Flash disk, also known as Solid State Disk (SSD), is widely considered by the database community as a next-generation storage media which will completely or to a large extent replace magnetic disk in data-intensive applications. However, the vast differences on the I/O characteristics between SSD and magnetic disk imply that a considerable part of the existing database techniques need to be modified to gain the best efficiency on flash storage. In this paper, we study the problem of large-scale concurrent disk scans which are frequently used in the decision support systems. We demonstrate that the conventional sharing techniques of mutiple concurrent scans are not suitable for SSDs as they are designed to exploit the characteristics of hard disk drivers (HDD). To leverage the fast random reads on SSD, we propose a new framework named Semi-Sharing Scan (S^3) in this paper. S^3 shares the readings between scans of similar speeds to save the bandwidth utilization. Meanwhile, it compensates the faster scans by executing random I/O requests separately, in order to reduce single scan latency. By utilizing techniques called group splitting and I/O scheduling, S^3 aims at achieving good performance for concurrent scans on various workloads. We implement the S^3 framework on a PostgreSQL database deployed on an enterprise SSD drive. Experiments demonstrate that S^3 outperforms the conventional schemes in both bandwidth utilization and single scan latency.

1 Introduction

During the past decade, flash memory has become a popular medium for storage due to its fast random access, low energy consumption, shock resistance, and silent working. In recent years, the flash disk, also known as Solid State Disk (SSD), has gained momentum in its competition against the conventional magnetic hard disk drive (HDD) in the market. Many software products which previously rely on HDD storage are now being considered to adopt SSD as an alternative. Among these, the database management systems are probably within

⋆ This work was supported in part by the National Science Foundation of China (NSFC Grant No. 60803003, 60970124) and by Chang-Jiang Scholars and Innovative Research Grant (IRT0652) at Zhejiang University.

© Springer-Verlag Berlin Heidelberg 2010

the class which requires the most attention and efforts, due to their vast complexity and numerous legacy applications. Despite the challenges, the prospect of replacing HDD with SSD in DBMS is attractive as the current database applications are desperately confined by the bottleneck of random I/O in HDDs. The most desirable feature of SSD in this regard is its elimination of seek and rotational delay, and the resultant fast random accesses. Table 1 indicates that the state-of-the-art SSD outperforms the conventional HDDs by more than two orders of magnitude in random reads.

Table 1. Performance Comparison of Different Read Patterns

	HDD †	SSD‡
Random Read Latency	9.69ms	0.03ms
Sequential Read Rate	114.7MB/s	up to 250MB/s

† : Western Digital 7200 rpm Digital Black
‡ : Intel X-25 Extremely

Unfortunately, previous studies have found that the performance of DBMS could not gain the expected improvements while they are deployed on SSDs straightforwardly [8] [14] [15]. The main reason for such results is that the storage engines and access methods of the conventional DBMSs are designed for the HDDs. As such, the advantages of SSDs cannot be fully exploited.

Let us consider the disk-based methods to handle *concurrent disk scans*, which are often executed as table scans in many decision support systems. The problem of concurrent table scans on HDD can be described as following: Given a limited buffer size of C and a sufficiently large table, if a running scan operation S_1 *outdistances* another scan operation S_2, meaning that S_1 is accessing a page which is at least $C + 1$ blocks ahead of S_2, S_2 is unlikely to hit any useful pages in the buffer with typical buffer management policies [9] [10] [11] [12] [13]. As a result, scan S_2 will not only cause disk thrashing but also compete the buffer space with S_1, leading to poor performance. In the literature, a few methods [1] [2] [4] [6] [7] have been proposed to address the above problem on HDDs. The main idea behind all these methods is that concurrent disk scans can "share" their footprints on disk pages as well as their memory buffer space. Therefore, multiple scans can share a (hopefully long) period of sequential disk-head movements, avoiding the penalty of chaotic random reads. However, these methods have the following drawbacks when employed on SSDs:

– First, there are almost always speed mismatches among multiple scans sharing the same sequential accesses. A faster scan has to be constrained by the slower ones in order to share the disk arm movement on the HDD. But on SSD this constraint is trivial as it is not mechanic-driven. Therefore, the capacity of SSD does not be fully exploited if the overall disk bandwidth is not saturated.

– Second, the existing methods focus on optimizing the disk accesses only because the queries are I/O-bounded. As SSD provides much larger bandwidth than HDD, some complicated queries are bounded by CPU power other than I/O bandwidth. Therefore, the strategy of the disk accesses needs to be reconsidered to exploit the system capability.

In this work, we leverage the advantage of SSD to speed up concurrent scans. First we introduce a cost model for concurrent scans on SSD, and then we propose a novel framework named *Semi-Sharing Scans (S^3)*. S^3 consists of two main components, namely the *scan scheduler* and the low-level *I/O scheduler*. (1) The *scan scheduler* clusters multiple scans into a set of groups by the similarity between their speeds. The scans in a same group share a stream of sequential block read operations which we refer to as the *unified I/O*s. In addition, faster scans which are not fully feed are allowed to perform *complementary I/O*s at other disk addresses. (2) The *I/O scheduler* dispatches the I/O requests from the scans into two separate queues, namely the *unified* and *complementary* I/O queue. During I/O scheduling, the unified queue enjoys higher priority compared to the complementary one.

Compared to the existing methods, S^3 is more suitable for concurrent scans on SSD as it optimizes both the bandwidth utilization and the average latency of each single scan under various workloads. Additionally, the framework is easy to implement, as only limited modifications are needed in a RDBMS. The contributions of our work include:

– We present a new cost model of the concurrent scans on SSD and demonstrate that the state-of-the-art scan scheduling policies do not fully exploit the I/O capability of SSD in various workloads.
– We propose a multiple-scan scheduling framework S^3 based on the cost analysis. This framework improves the bandwidth utilization via grouping and I/O scheduling. Compared to the existing methods, S^3 reduces the latency of a faster scan by compensating it with a separate stream of disk footprints.
– We implement the S^3 framework in the PostgreSQL DBMS. Our comprehensive experiments based on the *TPCH* benchmark indicate that the proposed approach is very effective for various workload patterns.

The rest of the paper is organized as follows. In Section 2 we define the problem and analyze the existing strategies on SSD. Our S^3 framework is described in Section 3. The experiments in Section 4 demonstrate the efficiency of S^3 and we conclude our work in Section 5.

2 Problem Definition and Cost Analysis

2.1 Problem Definition

A formal definition to concurrent scans problem is as follows. Assume that a table contains N pages stored in the secondary storage and a small buffer with capacity C (C is much smaller than N) in the main memory. If there are k scans denoted by $S_1, S_2..., S_k$, whose speeds are $v_1, v_2..., v_k$ respectively, all beginning at arbitrary times and having to access all the N pages of the table. The problem is how to coordinate the scans to gain the optimal performance.

The performance optimization of concurrent scans relies on the definitions of a few terms as following. The speed v_i of a scan S_i is defined as the number of pages that can be consumed per second when S_i is executed alone. Meanwhile, the response time of S_i, denoted by $rt(S_i)$, is the elapsed time between its start and end. Given sufficiently large disk bandwidth and a specific CPU processing power, the minimum response time of S_i, denoted by $rt_{min}(S_i)$, is the response time when S_i is executed alone. Therefore we have $rt_{min}(S_i) = N/v_i$. As the computational costs between queries may differ significantly, the speeds of different scans may also be very different. For example, in the *TPCH* benchmark, Q_1 generates a slow scan on the *Lineitem* table as it needs many arithmetic computations, while Q_6 is much faster as it only contains an aggregation.

A set of concurrent scans S_i $(i = 1, 2, \dots, n)$ may share disk I/Os if their I/O requests are scheduled properly. In such case, the overall number of pages being transferred per second is defined as the *bandwidth consumption.* However, in real world the bandwidth consumption is always constrained by the physical device bandwidth capacity V. Therefore, we refer to the unconstrained, ideal bandwidth consumption as the *bandwidth demand* or B_{demand}, while the constrained, actual bandwidth consumption seen on the physical disk interface as the *actual bandwidth consumption* or B_{actual}. We denote by $B_{demand}(system)$ the total bandwidth demand of the n scans in the system, and by $B_{demand}(s_i)$ the demand of scan s_i. $B_{actual}(system)$ and $B_{actual}(s_i)$ are defined similarly. We note that $B_{actual}(system)$ must always be no more than $B_{demand}(system)$. The latter must also be no more than the sum of all the speeds of the scans. Therefore, we have

$$B_{actual} \leq B_{demand} \leq \sum v_i.$$

The target of our optimization is to minimize the average response time

$$\frac{1}{n} \sum_{i=1}^{n} rt(S_i) \tag{1}$$

while B_{actual} is constrained by the device bandwidth capacity. It is important to note that the $B_{actual}(system)$ is either V or $B_{demand}(system)$ whichever is smaller.

2.2 Existing Scheduling Schemes

In this subsection, we shall look at a few existing schemes for handling concurrent scans. A naive scheme to handle concurrent scans is to simply rely on the LRU buffer replacement policy. The naive scheme does not allow any page reusing when the distance between the footprints of two concurrent scans is greater than C pages. For clearness we name it as *"no share"*. In such case, each scan needs to read from the secondary storage for every I/O request. This may lead to arm "thrashing" on HDDs, causing serious performance degradation. Using the MRU replacement policy might be slightly better than the LRU because it could reuse some initial pages in the buffer. However, the gain is very limited as C is usually much smaller than N.

A better scheme used by some DBMSs [1] [2] [3] [4] is to save I/O by sharing a stream of reads among multiple scans. When a new scan arrives, the system tries to attach it to an existing scan, which shares its reads of the rest pages with the new one. After a shared scan reaches the end of the table, it resumes from the beginning of the table until arriving at the original starting location. The main problem of this *shared* policy is that it cannot handle the speed mismatch between the scans. Once a scan outdistances another for more than C pages, the situation degenerates to the *no share* scheme. A common solution, namely *"strict share"*, is to stall the fast scans, thus guaranteeing the sequential movement of the disk arm. In this case, the $rt(S_n)$ is between $[N/v_{slowest}, N/v_n]$. To relieve the performance loss of the fast scans, DB2 [6] [7] proposes an improved *"group shared"* scheme in which the scans with similar speeds are grouped together and sharing happens within each group. However, the faster scans still need to wait for the slower ones in a same group.

A novel approach presented in [5] suggests to schedule the buffer by a suite of relevance-intensive functions. However, the computation of the relevance requires subtle dynamic statistics of the table and each scan. This would burden the CPU when the table is large. Another drawback is that it roughens the granularity of the buffer management, which makes it incoincident with accesses of other pattern.

2.3 Cost of Concurrent Scans on SSD

As SSD is non-mechanic driven and offers significantly higher random access speed compared to the HDD, the seek time and rotational delay are both eliminated. Therefore, when the DBMS is deployed on SSDs, the cost of multiple scans must be computed differently. In this subsection, we evaluate the scheduling schemes described in subsection 2.2 using the cost model for SSD. Then we demonstrate that none of the existing scheduling schemes for concurrent scans can adequately exploit the capacity of SSD.

Some assumptions are necessary for introducing the cost model of SSD. (1) We assume that we are equipped with sufficiently large number of CPUs regarding the I/O capacity so that the scans only sleep on I/O requests. (2) We observe that the bandwidth of a SSD device can linearly scale-up before its actual bandwidth consumption reaches the device capacity. Before that, the actual bandwidth being consumed should be equal to the sum of the demands of all scans. That means $B_{actual}(system) = B_{demand}(system) = \sum v_i$ when $B_{demand}(system) < V$. (3) When the bandwidth demand of the whole system $B_{demand}(system)$ is greater than the physical disk capacity V, indicating that the I/O subsystem is overloaded, some requests have to wait for others to complete. Such waiting causes performance degradation to a scan in terms of speed (bandwidth) loss. Without loss of generality, we assume that such speed loss is uniformly distributed among all scans.

To minimize the average response time proposed in Equation 1, we look at the response time of each scan s_k ($k = 1, \ldots, n$) under different scheduling schemes. Based on our second assumption of SSD, we have

$$B_{actual}(system) = min(V, B_{demand}(system)).$$

For the *no share* scheme, $B_{demand}(system)$ is Σv_i. If $B_{demand}(system)$ is less than V, each scan could achieve rt_{min} as no I/O request would be waiting. Otherwise, the actual bandwidth of s_k, $B_{actual}(s_k)$, is only $V * v_k / \Sigma v_i$. Therefore we have $rt(s_k) = N/V * v_k/\Sigma v_i$. For the *strict shared* policy, $B_{demand}(system)$ is equal to the speed of the slowest scan, denoted by v^s. Therefore the response time of each scan is N/v^s if $v^s < V$, and N/V if $v^s > V$. For the *group shared* scheme, the B_{demand} of each group g, denoted by $B_{demand}(g)$, is the slowest speed within group g (we denote it by v^{sg}) and $B_{demand}(system)$ is the sum of all groups in the system, namely Σv_i^{sg}. If $\Sigma v_i^{sg} < V$, then the response time of s_k is given by the slowest scan in its group, namely $rt(s_k) = N/v_k^{sg}$. Otherwise, $rt(s_k) = N/V * v_k^{sg}/\Sigma v_i^{sg}$.

Table 2. Response times of scans on SSD under existing schemes. v^s means the slowest speed among all scans, while v^{sg} means the slowest speed in its group.

	$rt(s_k)$	$B_{actual}(system)$
no share	$N/min(v_k,\ V * v_k/\Sigma v_i)$	$min(V,\ \Sigma v_i)$
strict share	$N/min(v^s,\ V)$	$min(v^s,\ V)$
group share	$N/min(v_k^{sg},\ V * v_k^{sg}/\Sigma v_i^{sg})$	$min(V,\ \Sigma v_i^{sg})$

Table 2 implies that all existing schemes have their limitations on the SSDs. The *no share* scheme could achieve the minimal average response time when $B_{demand}(system) < V$. However, this benefit comes at the expense of large bandwidth consumption when the number of the scans increases. The performance of the *no share* scheme is likely to plunge when $B_{demand}(system)$ exceeds V. In contrast, the *strict shared* scheme aims at minimizing $B_{demand}(system)$. However, faster scans in the system have to be confined by the slower ones. When the disk has available bandwidth, the *strict shared* scheme is unable to fully exploit it to expedite the fast scans. The *group share* scheme compromises between these two schemes. But it still does not make adequate use of the available bandwidths to expedite the faster scans within each group.

According to the analysis above, we design a new framework S^3 which could fully exploit the capacity of the SSD by combining the sharing I/O and separated I/O. S^3 shares part of the I/O reading among similar scans for reducing the bandwidth consumption, while it compensates the faster ones in random I/Os to improve the response time. Compare to the conventional sharing policies, S^3 could gain in both query latencies and bandwidth consumption.

3 The Semi-Sharing Scans Framework

3.1 Overview

The S^3 framework consists of two components, depicted in Figure 1. The *Scan Scheduler* coordinates the multiple scans, possibly in groups, and the *I/O Scheduler* dispatches the low-level I/O requests. Each scan consumes the pages of the table via a common buffer. The buffer is located in the main memory and caches part of the data pages from secondary storage. Once a new page is read in, the buffer manager evicts a victim page according to its priority mechanism, such as *LRU*, *MRU*, *Clock Sweep* etc.

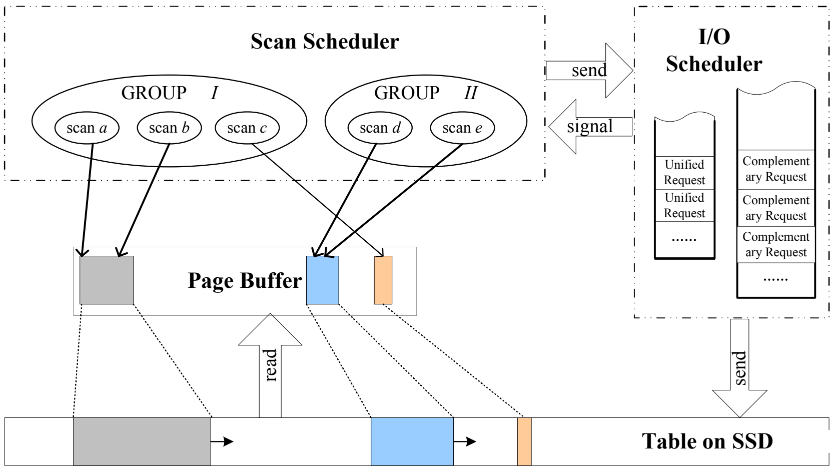


Fig. 1. An Overview of S^3. Two scan groups *I* and *II* are in progress. In group *I*, scan *c* acquires for a page via a *complementary* I/O request, while the group *I* moves forward via a *unified* I/O request.

The *Scan Scheduler* manages a number of scan groups, each of which contains one or several scans. To control the memory footprint, each group is allocated a fix-sized window in the buffer, which contains a number of contiguous data pages of the table. The scans in a same group share the reads of the pages in the window, which moves forward on the table via the cooperation from all the scans in the group. Specifically, the fastest scan reads the next data page and pins it in the memory until the slowest one consumes it. This read is translated to a *unified* I/O request in the low-level. We note that the *scan scheduler* can be easily implemented on the buffer module of any existing DBMSs.

If the fastest scan outdistances the slowest one by the size of the window, it cannot proceed sequentially to the next disk address. In this case we compensate it by allowing complementary I/Os outside the shared stream of disk addresses, by sending *complementary* I/O requests. For example in Figure 1, scan *c* is restricted by the slowest scan a, so c can make a complementary I/O request, which deviates from its conventional sequence.

All I/O requests in S^3 are scheduled by the *I/O Scheduler*. The *unified* I/O requests and the *complementary* I/O requests are stored and manipulated in separate queues. The unified requests are given higher priority. Therefore, if the unified request queue is not empty, no complementary requests will be processed. Once a request is completed, the *I/O Scheduler* evokes the scans which are waiting on it.

In the rest of this section, we shall look at the detailed techniques of scan scheduling and I/O scheduling.

3.2 Scheduling the Scans

Like the typical table scan processes in most existing DBMSs, each scan in the S^3 framework consists of three main steps, namely (1)*BeginScan*, (2) *FetchNext*, (3) *EndScan*. In the *BeginScan* step, a new scan is initialized. Then the scan traverses the table from the first page to the end by invoking the *FetchNext* step iteratively. After all the pages are processed, the *EndScan* step terminates the scan. We shall now elaborate these steps in the following.

Algorithm 1. BeginScan(S)

```
Open($S$);
$S.loc_{begin} \leftarrow 0$;
if there exists no groups then
    create a new group $g$;
    add $S$ to $g$
else
    $g \leftarrow ChooseGroup()$;
    $S.loc_{begin} \leftarrow MAX$;
    for each $S_i$ in $g$ do
        if $S_i.loc_{cur} < S.loc_{begin}$
        then
            $S.loc_{begin} \leftarrow S_i.loc_{cur}$
    add $S$ to $g$;
    if $CheckSplit(g)$ then
        Split $g$;
$S.loc_{cur} \leftarrow S.loc_{begin} + 1$
```

Algorithm 2. EndScan(S)

```
if $S$ is the only scan in $g$ then
    delete $g$;
    for all groups $G_N$ do
        $g_i \leftarrow$
        $PickSplitGroup(G_N)$
    Split $g_i$;
else
    remove $S$ from $g$;
Close(S);
```

Algorithm 3. FetchNext(S)

```
if $S.loc_{cur} = S.loc_{begin}$ then
    return NULL
while true do
    $loc_{acq} \leftarrow S.loc_{cur}$;
    $type \leftarrow unified$;
    $g \leftarrow S$'s group;
    for each $S_i$ in $g$ do
        if $S.loc_{cur} - S_i.loc_{cur} >$
        $|g.Window|$ then
            $loc_{acq} \leftarrow S.loc_{begin}$;
            $type \leftarrow complementary$
    if $type = unified$ then
        if $loc_{acq}$ is in buffer then
            $S.loc_{cur}++$;
            return $loc_{acq}$;
        else
            $SendRequest(loc_{acq}, S)$;
            $Signal(I/OScheduler)$;
    else
        if $loc_{acq}$ is in buffer then
            $S.loc_{begin}--$;
            return $loc_{acq}$;
        else
            $SendRequest(loc_{acq}, S)$;
            $Signal(I/OScheduler)$;
    $Wait()$;
```

The process of *BeginScan* in S^3 is described in Algorithm 1. When a new scan S begins, the scan scheduler creates a new group if there is no group at all. Otherwise, S is added to an existing group according to its speed, which is estimated by the query optimizer module. Then the scan scheduler checks whether a splitting is necessary. The detail of choosing and splitting the group will be described in the next subsection. Once a group is chosen, the beginning location of S is determined. For simplicity, we choose the page being processed by the slowest scan in the group as the beginning location of S.

The $FetchNext$ step returns one page of the table each time. The process of $FetchNext$ is described in Algorithm 3. To synchronize with other scans in the group, A scan S firstly checks its current location. If S has outdistanced the slowest one in its group for more than the window size, it will acquire a complementary page, starting from the beginning location of itself and extending backward. Otherwise it moves to the next page in the forward direction. If the page acquired is not in buffer, S has to send either a *complementary* or a *unified* I/O request to the I/O scheduler and wait for it. More details about the circular location computation, priority updating, and page latching etc. are omitted in our paper due to space limit.

Once a scan S has traversed all the pages in the table, we need to use *EndScan* step to complete the scan. A fast scan is ended when its current location and its beginning location meet. If S is the last scan in its group, the scan scheduler removes this group and checks whether the removal causes splitting of any other groups.

3.3 Grouping the Scans

Analogous to the discussion in section 2.3, the bandwidth demand of scans in S^3 can be calculated as following. As B_{demand} consists of two parts: the *unified bandwidth (Bu)* which is occupied by the *unified* I/O requests and the *complementary bandwidth (Bc)* produced by the *complementary* I/O requests. Generally in each group g, we have

$$Bu_{demand}(g) = v_{slowest}, \forall v_i \in g;$$

and

$$Bc_{demand}(g) = \Sigma v_i - (|g|) * v_{slowest}, \forall v_i \in g.$$

Adding $Bu_{demand}(g)$ to $Bc_{demand}(g)$ gives $B_{demand}(g)$. Given a set of groups $G_N = \{g_1, g_2, \cdots, g_m\}$, $B_{demand}(system)$ can be obtained as $\sum_{g \in G_N} B_{demand}(g)$.

When $B_{demand}(system)$ is smaller than V, the response time of each scan S_i is rt_{min} as each can acquire full bandwidth. We minimize the $B_{demand}(system)$ by grouping the scans, and splitting the groups when necessary. The central idea of the grouping and splitting is that the $Bc_{demand}(system)$ could be reduced significantly if scans with similar speeds are clustered in a same group. As a result, $B_{demand}(system)$ can also be reduced. We illustrate this optimization by giving an example of splitting a group.

Example 1. A group g containing two scans $S_1(v_1 = 1000$ p/s) and $S_2(v_2 = 3000$ p/s) is joined by a new scan $S_3(v_3 = 4000$p/s). This would cause splitting of group g, because the bandwidth consumption after the splitting can decrease by 1000 p/s, as indicated in Table 3.

Table 3. An Example of Splitting

	Scan Groups	Bu_{demand}	Bc_{demand}	B_{demand}
Before Splitting	$< S_1, S_2, S_3 >$	1000	5000	6000
After Splitting	$< S_1 >, < S_2, S_3 >$	4000	1000	5000

On the other hand, when $B_{demand}(system) > V$, some I/O requests have to wait for the others' completion. To improve the average response times, the *unified* I/O requests are always given higher priority by the *I/O scheduler*. Therefore, too many splittings would impair the performance when the system is heavily loaded in I/O, as splitting always increases $Bu_{demand}(system)$. In Example 1, $B_u(system)$ increases by 3000 after splitting. To control the number of the groups, we define two constraints in S^3. First, if the $Bu_{demand}(system)$ is greater than a threshold T (T is usually a bit smaller than V), we do not allow any splitting. Second, when the scan scheduler considers to attach a new scan to a group g, the remain life span of g must be larger than a threshold L_{max}. The reason is that if g will terminate in a short while, it might release its own bandwidth very soon. Therefore, there is no much benefit to attach the new scan to g.

We adopt a number of greedy algorithms for maintaining the groups. Once a scan arrives, the scan scheduler inserts it into the group which, if the new one becomes attached to it, produces the smallest $B_{demand}(system)$. After that, the scheduler checks if the group needs splitting via a routine called *CheckSplit*.

The basic idea of the *CheckSplit* algorithm is to find a splitting plan which minimizes $B_{demand}(system)$ without causing $Bu_{demand}(system) > V$. If this optimal splitting plan causes reduction to the current $B_{demand}(system)$, splitting is beneficial and therefore will be truly undertaken. It can be proven that if there are n scans in a group g, sorted by their speeds in ascending order as $g = (S_1, S_2, \ldots, S_n)$, then choosing the optimal splitting will be equivalent to finding an optimal index $i \in [1, n-1]$ such that vector g is split into two new vectors $(S_1, \ldots, S_i)$ and $(S_{i+1}, \ldots, S_n)$. Therefore, the computational complexity of them is $O(n)$.

When a group is eliminated, its unified bandwidth can be released. The scan scheduler then searches among all existing groups for one which, if being split, causes the largest reduction of $B_{demand}(system)$. This search algorithm is implemented in the *PickSplitGroup* algorithm. The *PickSplitGroup* checks each existing group using the *CheckSplit* routine and chooses the one which could benefit the system most. Again, groups whose remaining life spans are less than L_{max} are not considered. The chosen group is then split according to the respective optimal plan found by the *CheckSplit* algorithm.

It is worthwhile to mention that a group splitting causes changes to the window sizes of all groups. We adopt a simple method which redistributes the available buffer to each group uniformly.

3.4 Scheduling the I/O Requests

The I/O scheduler in S^3 manages all I/O requests in two separate queues, namely $Q_{unified}$ and Q_{comple}. The meanings of the queue names are self-explanatory. Once an I/O request, either a unified or complementary one, for page p is produced by S_i, it is appended to the tail of of the respective queue in the form $< p, S_i >$. The I/O scheduler dispatches the I/O requests as described in Algorithm 4.

Algorithm 4. I/O Scheduler

```
while true do
    if $Q_{unified}$ is not empty then
        $< p, S > \leftarrow GetHeader(Q_{unified})$;
        Read $p$ from SSD into buffer;
        for each $r_i$ in $Q_{comple}$ do
            if $r_i.S$ is in $S$'s group then
                delete $r_i$;
        for each $S_i$ in $S$'s group do
            $Signal(S_i)$;
    else if $Q_{comple}$ is not empty then
        $< p, S > \leftarrow GetHeader(Q_{comple})$;
        Read $p$ from SSD into buffer;
        $Signal(S)$;
    else
        $Wait()$;
```

The I/O scheduler process is evoked whenever a scan needs to access a physical page on the SSD. It first manages the requests on the header of $Q_{unified}$. Once a *unified* request is completed, it eliminates all the *complementary* requests from the same group and evokes all the scans in the group. The requests in the Q_{comple} are only scheduled if $Q_{unified}$ is empty. If there are no requests in both queues, the I/O scheduler sleeps until a new request arrives.

In case of heavy workload so that $B_{demand}(system) > V$, the *unified* I/O requests are still prioritized. The faster scans could succeed in performing *complementary* I/Os only when there are no pending *unified* I/O requests. When there are too many scans being executed concurrently, then S^3 degenerates to the *group share* scheme as described in section 2.3.

4 Experiments

We implement the S^3 framework on the PostgreSQL DBMS and conduct the performance study on a 20-scale *TPCH* database. The machine that we use is an Intel Xeon PC equipped with 8 cores and an Intel X-25E SSD. Our hardware can achieve a maximum random I/O bandwidth of 85 MB/s on Debian with kernel 2.6. Among the *TPCH* queries, we focus on Q_1 and Q_6. The former is a typical CPU-intensive query, while the latter is an I/O-intensive one. Both queries scan the *lineitem* table, which consumes around 18GB space on the SSD. The default buffer size of the DBMS is set to 256MB. For more different query speeds, we also add two new queries, named Q'_1 and Q'_6, which are slightly modified from Q_1 and Q_6. The respective speeds of all four queries have the following relation: $v(Q'_1) < v(Q_1) < v(Q'_6) < v(Q_6)$. Therefore, it can be inferred that Q'_1 is the most CPU-intensive query among the four, while Q_6 is the most I/O-intensive one.

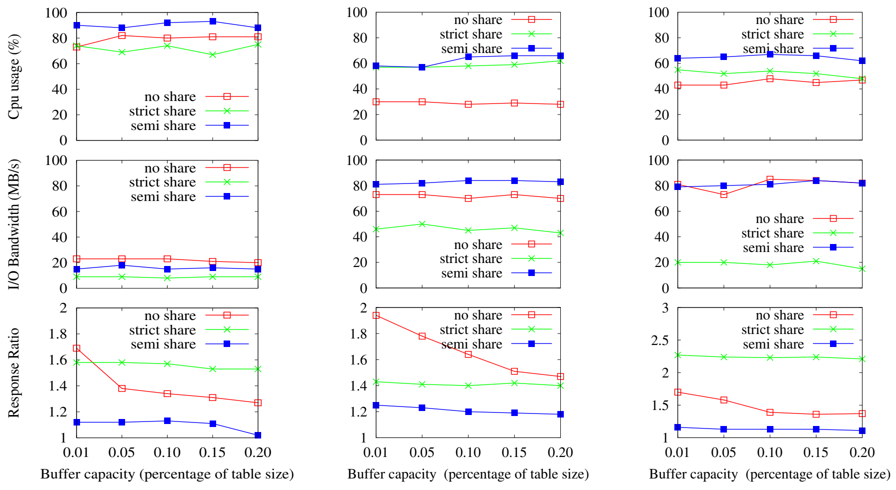
4.1 Results on Two Concurrent Query Streams

First we conduct a set of experiments on two concurrent streams of queries, where each stream issues the same query repetitively. For comparison, we also implement the *strict share* as described in Section 2.2. We do not compare with *group share*, as it apparently degenerates to the *strict share* or *no share* in this set of experiments.

Table 4. Statistics of different schemes when processing two concurrent query streams. Each cell under "response time" contains the two average response time values of the respective query streams.

	no share			strict share			semi share (S^3)		
	response time (s)	I/O throughput (MB/s)	CPU usage	response time (s)	I/O throughput (MB/s)	CPU usage	response time (s)	I/O throughput (MB/s)	CPU usage
Q_1 vs Q'_1	1261 2393.3	23.7	76% 88%	2090.1 2108.6	9.1	37% 100%	1096.7 2228.6	18.6	75% 100%
Q_6 vs Q_1	332.3 1230.1	63.9	20% 43%	906.6 915.3	20.5	4% 100%	225 910.9	80	30% 100%
Q_6 vs Q'_6	439.1 534.8	73.3	15% 44%	357.3 366.8	50.7	14% 99%	229.3 330.2	82.1	32% 82%

Table 4 shows the statistics of different schemes on various query streams. It can be seen that, in most cases, S^3 provides considerably shorter average response time for each query. Under the *no share* scheme, the CPU-intensive workload (Q_1 vs. Q'_1) is completed quickly as they could fully exploit the CPU capacity and the I/O bandwidth. However, for Q_6 vs. Q'_6, the average response times of both streams are poor as the two queries compete on I/Os. The *strict share* scheme performs well when two I/O-intensive queries Q_6 and Q'_6 are executed concurrently. However, it confines the faster scan to the slower one when they run concurrently, as the results of Q_1 vs. Q_6 indicate.



(a) CPU-intensive workloads (Q_1 vs Q_1')

(b) I/O-intensive workloads (Q_6 vs Q_6')

(c) Mixed workloads (Q_1 vs Q_6)

Fig. 2. Statistics of different strategies on varying buffer size, workloads. The *average latency* of the query is normalized.

Figure 2 illustrates the results of varying the buffer size from 1% to 20% of the full table size. We use a smaller dataset of 5-scale data, in which the *lineitem* table is around 4.5GB. The *response ratio* of a scan S_i is defined as $rt(S_i)/rt_{min}(S_i)$. In Figure 2, we can see several interesting results. (1) S^3 always has the highest CPU utilization compared to the other scheduling schemes. The reasons are two-fold. The first reason is that the faster query will never be blocked if the whole I/O bandwidth is not saturated. The second reason is that the slower query can always benefit from the faster one, as the pages loaded from SSD by the latter can be reused by the former. (2) Only the *no share* scheme is sensitive to the buffer size. The *strict share* and S^3 do not appear to vary much by the buffer size as they intentionally share the buffer reading themselves. (3) The response ratio of S^3 is always better than the other two. This confirms the *semi share* as an efficient scheme for concurrent scans on SSDs.

4.2 Results on Many Concurrent Queries

In this subsection we conduct more than two queries concurrently to study the performance and behavior of S^3. Specifically we execute 5 streams of queries, consisting of two streams of Q_1', one stream of Q_1, and two streams of Q_6'. Table 5 presents the results of different scheduling schemes. The average response time of *no share* is slow because the bandwidth demand exceeds V. The *strict share* and *group share* could save the bandwidth efficiently. However, they delay the faster scans like Q_1 or Q_6' to share the whole scan process. Our S^3 scheme outperforms the other three in average response time because it exploits the I/O bandwidth of SSD more efficiently.

Table 5. Statistics of different strategies when processing five concurrent query streams

	no share			strict share			group share			semi share (S^3)		
	response time (s)	I/O thr. (MB/s)	CPU usage	response time (s)	I/O thr. (MB/s)	CPU usage	response time (s)	I/O thr. (MB/s)	CPU usage	response time (s)	I/O thr. (MB/s)	CPU usage
Q'_1	2459.4		80%	2153.2		83%	2196.7		82%	2132.4		94%
Q'_1	2499.4		80%	2144.6		83%	2160.5		83%	2202.8		83%
Q_1	1275.6	82.5	64%	2109	7	45%	2079.1	33.5	42%	1064.4	53	65%
Q'_6	651.8		31%	2094.2		9%	629		35%	545.2		40%
Q'_6	647		26%	2099.1		9%	625.2		35%	589.2		35%

Finally, we conduct a more mircoscopic experiment, in which five queries are started in the order of $Q'_1, Q'_1, Q_1, Q'_6, Q'_6$, each one is started 50 seconds later than its predecessor. Figure 3 plots all disk page IDs being accessed by time. It can be seen that after the first query Q'_1 is started, new queries Q'_1, Q_1, and Q'_6 are started subsequently. All these four are attached to the same group. When the final Q'_6 joins, the group is split into two, namely *Group I* including two Q'_6s, and *Group II* including two Q'_1s and Q_1. Since Q_1 is faster than Q'_1, it performs complementary I/Os to compensate the speed mismatch. We notice that when *Group I* ends, Q_1 stops the complementary I/O for a while, as indicated by the gap in the top-most blue curve. This is because the window size of *Group II* is enlarged due to the elimination of *Group I*. Therefore Q_1 can move forward for a certain period on the unified I/O stream, until reaching the end of the window.

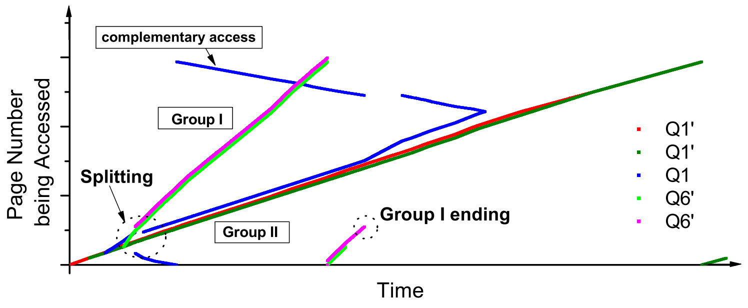


Fig. 3. The process of 5 queries starting at an interval and running concurrently

To summarize, on both CPU-intensive and I/O-intensive workloads, S^3 outperforms the other conventional schemes. Generally, S^3 could improve the query efficiency for about 20% to 100%.

5 Conclusion

In this paper we propose a new framework, namely *Semi-Sharing Scan*, for processing concurrent scans on SSD efficiently. S^3 groups the scans and compensates the faster ones by random I/Os, if the hardware bandwidth is not saturated. Via I/O scheduling, S^3 also improves the bandwidth utilization on I/O-intensive workloads. We implement S^3 on the PostgreSQL DBMS. Experiments based on TPCH benchmark confirm that S^3 is an efficient scheme for concurrent scans on SSD.

References

1. Colby, L.S., et al.: Redbrick vista: Aggregate computation and management. In: Proc. ICDE (1998)
2. Cook., C., et al.: SQL Server Architecture: The Storage Engine. Microsoft Corp., `http://msdn.microsoft.com/library`
3. Jeff Davis Laika, Inc.: Synchronized Sequential Scan in PostgreSQL: `http://j-davis.com/postgresql/syncscan/syncscan.pdf`
4. NCR Corp. Teradata Multi-Value Compression V2R5.0 (2002)
5. Zukowski, M., Héman, S., Nes, N., Boncz, P.A.: Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In: VLDB (2007)
6. Lang, C.A., Bhattacharjee, B., Malkemus, T., Padmanabhan, S., Wong, K.: Increasing buffer-locality for multiple relational table scans through grouping and throttling. In: ICDE (2007)
7. Lang, C.A., Bhattacharjee, B., Malkemus, T., Wong, K.: Increasing Buffer-Locality for Multiple Index Based Scans through Intelligent Placement and Index Scan Speed Control. In: VLDB (2007)
8. Lee, S.-W., Moon, B., Park, C., Kim, J.-M., Kim, S.-W.: A Case for Flash Memory SSD in Enterprise Database Applications. In: Sigmod (2008)
9. O'Neil, E.J., O'Neil, P.E., Weikum, G.: The LRU-K Page Replacement Algorithm For Database Disk Buffering. In: SIGMOD Conference (1993)
10. Johnson, T., Shasha, D.: 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In: VLDB (2004)
11. Nyhcrg, Chris: Disk Scheduling and Cache Replacement for a Database Machine, Master Report, UC Berkeley (July 1984)
12. Robinson, J., Devarakonda, M.: Data cache management using frequency-based replacement. In: Proc. ACM SIGMETRICS Conf. (1990)
13. Lee, D., Choi, J., Kim, J.-H., Noh, S.H., Min, S.L., Cho, Y., Kim, C.-S.: LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. IEEE Trans. Computers (2001)
14. Lee, S.W., Moon, B.: Design of flash-based dbms: an in-page logging approach. In: SIGMOD Conference, pp. 55–66 (2007)
15. Tsirogiannis, D., Harizopoulos, S., Shah, M.A., Wiener, J.L., Graefe, G.: Query processing techniques for solid state drives. In: Sigmod (2009)