Flashing Databases: Expectations and Limitations

Stephan Baumann
Ilmenau Univ. of Technology, Germany
stephan.baumann@tu-ilmenau.de

Giel de Nijs
VectorWise, Netherlands
giel@vectorwise.com

Michael Strobel
Ilmenau Univ. of Technology, Germany
michael.strobel@tu-ilmenau.de

Kai-Uwe Sattler
Ilmenau Univ. of Technology, Germany
kus@tu-ilmenau.de

ABSTRACT

Flash devices (solid state disks) promise a significant performance improvement for disk-based database processing. However, database storage structures and processing strategies originally designed for magnetic disks prevent the optimal utilization of SSDs. Based on previous work on benchmarking SSDs and a detailed discussion of I/O methods, in this paper, we analyze appropriate execution methods for database processing as well as important parameters and boundaries and present a tool which helps to derive these parameters.

1. INTRODUCTION

Recent progress in the development of flash memory technology has led to an increasing acceptance of solid state disks (SSD) as a replacement for classical magnetic disks. Although SSDs are still more costly and provide less storage capacity than magnetic disks, they offer significant advantages in terms of read access times and energy consumption. Special properties – such as very short read latencies and rather expensive writes (caused by the necessary reprogram/erase operations) and a finite number of reprogram operations per cell – make SSDs especially interesting for read-intensive applications.

Since their beginning days, DBMS have been designed to work with external storage which is much slower than main memory and favors sequential access patterns. However, index structures like the well-known B-tree or clustered data organization such as MDC [10] and ADC [9] offer row- or block-wise access opportunities and thus can help to reduce the data volume in a horizontal way, albeit with the extra costs implied by random reads. Thus, there is usually a tradeoff between the minimal block size for which a magnetic disk still shows acceptable performance for random access and the reduction granularity provided by indexing. The advent of flash memory as external storage has triggered the development and study of algorithms and data structures suitable to the new qualities of this storage type, e.g. variants of B-tree [13], transaction logging [6], self-tuning indexing structures for small devices [8], and special query operator implementations [12].

Based on these observations, two main questions regarding the usage of SSDs in DBMS arise:

- How well suited are the storage and access structures of classic DBMS for the optimal utilization of SSDs?
- More generally – which access patterns and profiles are optimal for SSDs?

Answering these questions requires a detailed analysis of SSD behavior. However, the face that SSDs are not unlike black boxes for DBMS developers complicates this task. These disks implement sophisticated block mapping and wear leveling strategies which may have a great impact on performance and make the modeling of disk behavior more difficult than for magnetic drives. Furthermore, with the improvement of technology new features are continuously added, resulting in significant changes from one generation to the next.

One of the most important contributions on benchmarking flash devices is from Bouganim, Jónsson and Bonnet [1]. Bouganim et al. introduced the uFLIP benchmark, a collection of nine micro-benchmarks which consist of different basic I/O patterns. Based on tests with several flash devices, the authors come to the following conclusions (regarding read operations): There is some overhead per I/O, i.e. some latency, which means that larger I/Os should be preferred. The best block size is 32KB and blocks should be aligned to flash pages. Concurrent access from a few patterns is acceptable, but concurrent or delayed I/O do not help to improve performance.

Another mentionable work is the five-minute rule for flash memory by Graefe [3]. Based on the famous five-minute rule paper by Gray and Putzolu for trading off I/O capacity and memory, Graefe has reconsidered this tradeoff for flash memory instead of magnetic disks. He argues that the optimal page size for B-tree indexes on flash memory is 2KB (much smaller than for traditional disks) and that flash memory shifts the break-even point between table scans and index search from 10,000 to 500,000 rows, satisfying the query predicate for a table of 100M tuples.

Based on these observations, we draw the following two conclusions:

(1) SSDs shift query execution toward index-based query plans

Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana. Copyright 2010 ACM 978-1-4503-0189-3/10/06...$10.00. .

(2) SSDs favor smaller block sizes.

Consequently, clustered storage structures with small clusters are a promising strategy leading to efficient preselection.

In this work, we strive to verify these hypotheses and derive appropriate access profiles. We do not intend to define yet another benchmark, but present a tool to help find the right parameters and analyze important parameters and boundaries needed for designing storage structures supporting efficient clustering and query processing strategies.

2. PRELIMINARIES

Getting a system to exploit the technical advantages and to avoid the disadvantages of solid state disks requires a careful calibration of I/O depending on the properties of the disk. For better understanding, this chapter gives a brief overview of SSD technology and different I/O methods.

2.1 Solid State Disks

SSDs have previously been explained by various sources. In the following, we summarize the internals of these disks. We start with flash memory as this is the base building block of SSDs, followed by the general architecture. The last part of this chapter explains the flash translation layer which manages the requests issued by the system.

2.1.1 Flash Memory

Flash memory is a non-volatile semiconductor memory. A flash chip consists of a large array of flash cells, where different types of chips are distinguished according to the architecture of the array - NOR or NAND - and the type of the cells - Single Level Cell (SLC) or Multi Level Cell (MLC) - each with specific advantages and drawbacks. However, SSDs are throughout based on NAND flash. Therefore, in this paper, flash always refers to NAND flash.

The cells of a chip are organized into a hierarchy of pages, blocks and planes. A flash page is the smallest unit of access, for reads as well as for writes. A flash block is the smallest unit of cells that can be erased. Typical sizes are 4KB for a flash page and 128KB for a flash block. In addition, the chips are divided into a number of planes, for example one for odd and one for even pages, that are independently accessible. Every chip may also contain a cache, which allows for caching of repeatedly read pages.

Flash supports three basic operations: read, program and erase. A read operation is as follows: the page is first loaded into the data register of the corresponding plane and then shifted out over the data bus of the chip. The program operation is essentially the other way around, the page is shifted into the data register, and subsequently stored in flash. Intel states a read latency of $75\mu s$ for its SLC NAND flash SSDs and $85\mu s$ for its MLC NAND flash SSDs, as well as a write latency of $85\mu s$ for its SLC NAND flash SSDs and $115\mu s$ for its MLC NAND flash SSDs [4, 5]. Erasing takes significantly more time than reading or programming, typically a few microseconds.

2.1.2 Architecture

Every solid state disk basically consists of a number of flash packages and a controller, where a flash package is built of one or more flash chips. A set of flash packages has its own channel through which it is connected to the controller supporting independent communication for each channel. Additionally, the controller is equipped with a host interface (e.g. SATA) to communicate with the system. The controller (sometimes also a set of controllers) itself contains logic for managing requests between the flash packages and the host interface.

This general architecture offers two opportunities for parallelization of requests. First, the controller can distribute requests among the flash packages. Second, a flash package supports concurrent processing of requests on its planes. However, parallelization can only be exploited if there are sufficient requests that can be distributed among the different packages and planes. Therefore, size, count and locality of the requests are critical factors for performance.

The higher the number of parallel accessible chips, the higher the number of parallel processable requests which are needed to obtain maximum performance. Thus, the newest generations of SSDs support Native Command Queuing, which facilitates a change of processing order of the requests, so that the number of independently processable requests increases.

2.1.3 Flash Translation Layer

The flash translation layer (FTL) is the software that is run in the controller of the SSD. It translates a request issued to the host interface into (multiple) request(s) to the flash packages with respect to performance and lifetime aspects by implementing sophisticated block mapping and wear leveling mechanisms.

Block Mapping. From the perspective of the operating system, the SSD is a block device, which can be accessed via a block interface. In order to provide this block interface, the FTL needs to implement a mapping function between the logical block addresses (LBAs) of the *logical device* and the physical block addresses (PBAs) of the *physical device*, i.e. the flash packages inside the disk. Since a flash page needs to be in zero-state for programming and must be erased beforehand if it is not, a static mapping would always trigger slow erases for updates. Therefore, a dynamic mapping is used, where updates are handled similar to a log. Every update results in a new page being programmed and in an adaption of the mapping function, while the old page is left outdated. The FTL holds a number of pages for writes in an allocation pool and tries to reclaim the outdated pages during idle times.

Wear Leveling. Another issue is that flash cells wear and become unreliable after a certain number of program and erase cycles. Obviously, unreliable cells need to be handled to ensure data integrity. But as flash cells only have a life span of 10^5 to 10^6 program and erase cycles, depending on the type of flash, more sophisticated solutions are needed. Thus, to maximize the lifetime of each cell, writes are distributed equally between flash cells to ensure equal wear of the cells. This mechanism is also known as wear leveling. It guarantees that write requests always go to one of the least used free pages, and rarely changed pages are transferred to frequently changed ones. Over time, this leads to a random organization of the data, so that even requests of sequential data are executed by a number of randomly distributed page requests.

2.2 I/O execution methods

To understand the need for asynchronous I/O, it is important to first have a good overview of how normal, synchronous I/O works. To get from issuing a read request in an application to receiving the actual data in this application, a number of steps at various layers throughout the system are involved.

2.2.1 Synchronous I/O

The read function call at the application level (in *user space*) is implemented in a standard system library. Such an implementation mainly consists of translating the read function call to a system call. This system call passes all the parameters (e.g. file descriptor, location in the file, number of bytes, the buffer where the data is to be stored) to the operating system. The read request is further executed by the operating system in *kernel space*.

The operating system generally splits up a larger read request into a number of requests so that each request does not exceed the maximum request size of the hardware block device (e.g. 1024 bytes). These requests, together with concurrent requests from (possibly different) applications, are sent to the *block device layer* in the operating system. One of the tasks of the block device layer is scheduling all outstanding requests in such a way that the underlying device can execute them as efficiently as possible. This is done by the *I/O Scheduler* (sometimes called *elevator*). The re-ordered requests are sent to the physical storage device (possibly after passing through a software or hardware RAID system), where they are executed.

The block device layer receives the data for the executed read requests. If the request was split into smaller requests, the results are gathered and assembled. The data is then copied from kernel space memory to the buffer provided by the application in user space memory. The operating system returns from the system call, which in turn returns to the application. The read request is finished.

2.2.2 Native Command Queuing

All the steps described take time to execute. Starting at the lower layers, there is a delay between sending a request to a device and receiving the result. There is also a delay between receiving a result and sending the next request, because the result has to be processed by the operating system. Modern magnetic drives and SSDs have therefore implemented a system called Native Command Queuing (NCQ). This is a queue of I/O requests, implemented in the storage device itself, which allows the device to quickly execute a number of commands in the order it prefers without having to wait for the operating system. It also enables the operating system to quickly send a number of commands to the device without having to wait for the single commands to be finished. To obtain fast transfer speeds, it is important to keep this command queue as full as possible.

Similar to the above described latencies, there is another such delay between user space and kernel space. Not only does the application have to wait for the storage device to execute all commands for a specific read request, it also has to wait for the operating system to copy the result data back from kernel space to user space. The operating system in turn cannot issue the next batch of commands to the device until it has received the next read request from the application and, hence, has to wait for the application to have processed the data of the previous request. Due to this delay, the operating system runs out of requests to send to the device and the command queue in the device becomes empty, so the device is idle for some time.

2.2.3 Massively parallel storage systems

The effect of idle devices on the actual data transfer rate increases as the devices get faster, as a faster device means that more data can be transferred during idle time. With the modern fast SSDs, this effect has a huge impact. If the devices are connected in a RAID array, it becomes even worse, because all devices in the array have to be kept busy. The SSDs can only operate at full speed when there are sufficient commands to work with, due to their parallel nature. Adding this parallelism to a RAID array intensifies this into a massively parallel storage system.

Using synchronous I/O, the only way to minimize this problem is to issue huge read requests (in the order of hundreds of megabytes) so the operating system has enough commands to keep the I/O queues of the devices filled. This is far from a viable solution.

2.2.4 Asynchronous I/O

To solve the issue of the application having to wait on the operating system and the operating system having to wait on the application, read requests can be issued asynchronously. In short: the application can issue a number of requests for data it will need at some time in the future and continue processing. These requests do not have to be continuous, so the size of the individual requests can remain relatively small. This way, the operating system can be provided with enough requests to keep the command queues of the devices full and the devices can operate at full speed. Every time a request is finished, the operating system notifies the application, which then can choose to process the result data.

2.2.5 Parameters

The exact way I/O operations are performed within an application is determined by a number of parameters. These parameters have to be tuned to the system, the workload and the storage devices to be able to obtain highly efficient I/O and thus, high bandwidth.

Block size. The minimum size of a single read request for a certain application is indicated by the *block size*. This does not have any relationship with the actual disk block size (i.e., a sector) and also very little with the memory page size, although it is probably a multiple of both of these. The block size is determined by the internal data layout of the application. Smaller block sizes offer more flexibility, but a minimum size is needed to be able to perform efficient I/O. The block size should not be smaller than the SSD page size and preferably not smaller than the page size. If a RAID array is used, the block size should preferably not be smaller than the page size of one device multiplied by the number of devices in the array to make sure that all drives are used during the execution of a read request.

Queue depth. The maximum number of outstanding requests the operating system can handle will be called the *queue depth*. This is often a configurable parameter. Increasing this number will lead to more possibilities for optimiza-

tion but will also increase the maximum latency of a request. The queue depth has to be a certain size to have enough simultaneously outstanding request to keep the storage device busy while the application is processing results of previous requests. This number depends on the specific properties of the storage device itself as well as the rest of the system.

2.3 Prefetching

The mechanisms described above only work when the I/O subsystem of an application actually needs to read enough data to fill up the asynchronous I/O request queue. The application has to have enough non-I/O work available as well to stay busy until the I/O operation has been performed. This work could consist of processing previously completed read requests. Hence, to be able to use asynchronous I/O effectively, an application needs to be able to predict what data it will need well in advance (depending on the queue depth) and will need to implement a good, application-specific prefetching mechanism.

Prefetching is often already implemented by the operating system or even by a storage device itself. The semi-random nature of read requests issued by a database system are likely to confuse these prefetching mechanisms, as they have no application knowledge. In one possible scenario, an existing pre-fetcher detects sequential I/O because of a number of sequential read requests and starts prefetching. The sequential burst however, is very short and the next requests are in a different location, invalidating the already prefetched data and wasting I/O bandwidth. It is therefore important to disable existing pre-fetchers in favor of one with domain knowledge. The operating system prefetching mechanism can oftentimes be bypassed (with *direct I/O*), but if such a mechanism is implemented in a storage device itself, we have to be very careful with the on-disk data layout. We will elaborate on this in Section 5.

3. DATA LAYOUT ON DISK

As we have seen in the previous section, parameters such as block size and queue depth need to be specified for asynchronous I/O. However, it is well known that the performance of (asynchronous) I/O operations also depends on the data layout on disk. For magnetic disks certainly, choosing the block size in which to organize data is a tradeoff for the database system between the minimal block size that still provides sufficient bandwidth for highly selective random access patterns and the much larger block size that truly optimizes sequential patterns. In Section 5, we will see that this is also the case for solid state disks, however, for smaller block sizes. Another factor to keep in mind is that a sequential data layout on disk requires significant maintenance. For column store systems such as C-Store ([11]) and VectorWise (previously called X100 ([14])) however, storing data fully sequentially is not even beneficial. This can be explained by the way such systems access data. When a query requires a table scan over multiple columns, in most cases the I/O unit will request the first block of more than one of these columns in order to provide data for further processing alongside the query execution tree. That is, in a column store, a sequential scan takes the form of multiple concurrent table scans from the disk access point of view. Additionally, having different data types and thus different column widths can lead to requests of varying sizes. Note that the particular combination of which columns are scanned together varies

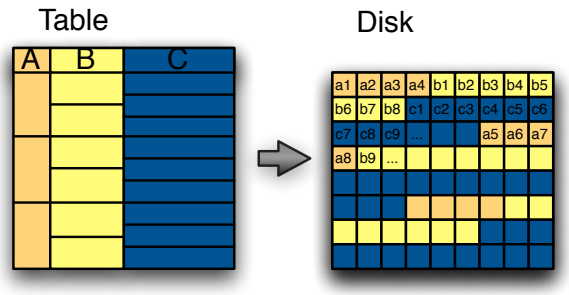

Figure 1: Possible data layout for a table with three columns using a *group size* of four.

from query to query, so, in general, it cannot be anticipated in the physical design. Thus, depending on the data type of a column and the way queries are processed, a request will have more or less opportunities for sequential access but in general only for a few consecutive blocks.

Also, in column oriented systems retrieving a certain number of tuples will generate access to data at a much smaller granularity than in their row oriented counter-parts. As data is stored and accessed column by column and (thus) highly effective compression techniques can be applied, data requests can be order(s) of magnitude smaller than in row stores. This and the previously mentioned tradeoff between physical storage optimized for random or sequential access requires a flexible solution for accessing data on disk. VectorWise tackled this problem by implementing a third I/O parameter, the *group size*. This *group size* defines the minimum number of sequentially stored blocks for a column. To gain a better understanding, Figure 1 shows a table with three columns of different widths. Each horizontal line inside the columns marks the end of a group of four blocks. On the right side we can see how these columns are mapped to disk using a group size of four (blocks are numbered according to each column).

Having grouped data organization has multiple advantages: First, random access on small blocks is not influenced and can be executed the way it would be without grouping. Second, requests of larger blocks automatically result in small sequential access patterns for multiple disk blocks without additional overhead at the I/O unit. Third, storage becomes more flexible in comparison to full sequential storage, as disk space is only partitioned at the granularity of the group size. Also, in comparison to a fully sequential data layout, the grouped layout has the advantage of less maintenance in case of updates or inserts, as the system only has to ensure that *group size* number of blocks are aligned sequentially.

4. EXPECTATIONS

In order to provide fast (random and sequential) access to data, especially multiple concurrent accesses at a time with varying granularity as necessary to exploit structures as ADC or MDC, solid state disks and asynchronous I/O form a great alliance. To fully benefit from such clusterings a database system needs to be able to provide fast access to very small data blocks in addition with high bandwidth

in case multiple requests for very small data blocks are issued. SSDs in principle contain the technology to support these requirements but it is up to asynchronous I/O to fully benefit from this technology.

High Parallelism. As described, a SSD consists of multiple flash chips which are independently accessible via multiple channels. As every flash chip and there every plane is needed to generate high bandwidth, in user space a sufficient amount of data needs to be requested. Using asynchronous I/O there are two basic approaches to reach this goal: First, the queue needs to be long enough, so that there are always pending requests in the SSD controller to be distributed among the parallel units. Or second, the requested blocks are large enough to be split into sufficient device I/O requests which then can be executed in parallel. Concerning clustered storage in database systems, especially the first approach is of interest and there, in particular the minimal size of an I/O unit delivering high bandwidth with an acceptable length of the queue. As described in previous work, a block size of 32KB is a good choice for SSDs to get high performance. But as 32KB still can be considered fairly small, a longer queue should be necessary to achieve high bandwidth. For larger blocks, especially blocks larger than the maximum request size of a disk, the queue depth can be expected to be smaller without loosing performance.

When using SSDs in RAID setups, the degree of parallelism becomes even higher as all disks need to be supplied with a sufficient amount of requests. This should reflect in a (linear) increase in the required number of requests or the required block size of the requests.

Random = Sequential. Although there is a lower request latency for sequential reads in comparison to random reads according to [2], we expect asynchronous I/O to minimize the gap between these two input patterns. Again this expectation is based on the parallel architecture of SSDs. As a disk's performance depends on the utilization of the available parallel units, it should not make a difference if the requests are supplied by asynchronously issued sequential or random requests as long as the device queue has enough requests to keep all units busy. In this scenario NCQ should also contribute to get equally fast random and sequential access as requests can be re-arranged to utilize the parallel architecture of a disk. A second reason for our expectation is the existence of block mapping and wear leveling. As even sequential data is distributed by these functions, a sequential access to a block of data is split into several disk block accesses spread all over the disk making random and sequential access looking equal from a locality aspect. Third, the lack of mechanical parts and, thus equally fast access to each part of the disk, supports this expectation.

Benefits of Grouping. In Section 3 we introduced the concept of grouped storage for column stores. Originally it was developed for magnetic disks and their gap between random and sequential access latencies. Although this gap is small for solid state disks, they may still profit from grouping. However, for a true sequential pattern, it should not make any difference at all, if data is read in groups or not. On the other hand, random access to groups of data may be faster than random access to single blocks, as requesting groups may benefit from sequential latencies. Also, for solid state disks (similar to magnetic disks) we expect performance to decrease as block requests drop below a certain size. Here, grouping may mitigate this decrease, as *group size* block requests are actually sequential block requests. In combination with asynchronous I/O, this advantage of grouping should not be as visible as larger queues, and thus, sufficient parallelism should provide requests for all units inside the disk.

Above we only discussed the benefits of grouping from lower sequential access latencies. Another more interesting benefit of a grouped data layout comes into play, when the I/O unit can request disk blocks of varying sizes. In such a case, the *block size* defines the minimal access granularity and $blocksize * groupsize$ defines the maximal granularity. For every multiple of *block size* between these two values, the I/O unit only requests a single block, as opposed to the first approach, where multiple sequential requests were issued. Such a flexible I/O unit could be the key to having very small random request blocks - as required for column stores but also for indexed storage in row stores - while sequential access or better requests of larger data chunks can be executed at full bandwidth. So from this point of view, the tradeoff between the minimal block size that is fast for random access and the necessary block size that provides high bandwidth for sequential patterns would disappear.

5. ANALYSIS

This section briefly introduces our analysis tool followed by a detailed analysis of I/O parameters for database input operations executed on solid state disks.

5.1 Analysis tool

To validate the theory described above and to realize a method to empirically determine the parameters needed for efficient asynchronous I/O on a specific system with a specific workload, we have implemented a database I/O analysis tool (or *DIAT* for short). This tool is able to iteratively test the I/O bandwidth for all combinations of block size, group size and queue depth within a certain parameter space. We can then utilize these results to optimally tune a real-life system.

After creating a large enough file (4GB by default), the tool runs a read test for each combination of the three parameters in the selected parameter space. The read pattern will consist of reading data in blocks of the specified size, starting at a random location in the file. A number of blocks will be read sequentially, determined by the group size (note here that the group size is only used to define the number of sequential requests, not for combining multiple block requests as described in Section 3). Between groups, the tool will jump to another random location in the file. This way requests are generated for this read pattern and are submitted to the asynchronous I/O facility of the operating system until the number of outstanding requests equals the queue depth. When the tool is notified that one or more requests have been completed, the next requests are generated and submitted, again until the queue depth is reached. A test run ends when the previously specified amount of data has been read. The effective bandwidth is then simply calculated by $test\,size / test\,time$ and output in a format easy for plotting. Then the next set of parameters is tested.

5.2 System Setup

We evaluated on an Intel Xeon E5505 2.00 GHz with 8GB main memory, a 1TB Western Digital Caviar Black for our tool and operating system, a 64 bit Debian, kernel version 2.6.26. Our mainboard is a S5500BC from Intel. As solid state drives we used 80GB Intel X25-M gen2. Three SSDs were connected via an Adaptec 1430SA PCIe x4 RAID controller plugged into one of the boards PCIe x8 slots. The fourth disk was connected directly to one of the onboard SATA ports leading to a maximum bandwidth of about 1GB/s. As each single drive showed a maximum bandwidth of about 270MB/s, a total of 1GB/s for the RAID is very close to the theoretical maximum. This setup was necessary because bandwidth was limited to about 800MB/s when a) all disks were connected to the Adaptec controller, b) the controller was placed into one of the PCIe x4 slots and c) all drives were connected via onboard SATA ports. We first tested a single drive, one of the drives connected to the Adaptec controller, followed by tests on a 4 disk linux software RAID 0 created with *mdadm*[1]. Before we started with the actual experiments we compared different disk schedulers and in line with [2] chose *noop* for our experiments. The disks had been in use prior to the final experiments for this paper. They were formatted before each experiment but no additional care was taken toward an initialization process as, for example, described in [1]. We believe this to be sufficient as we have neither read about nor experienced an influence of the disks state regarding read operations.

5.3 Single SSD

For the single SSD experiments, DIAT created a 32GB file of which 4GB were requested for each run, i.e. a setting for each of the parameters *queue depth*, *group size* and *block size*.

Hight Parallelism. In our expectations we stated that high parallelism is necessary to use the full bandwidth of a SSD. We mentioned two possible ways to achieve this: long queues and large blocks. Figures 2 and 3 present the bandwidth measured by DIAT for multiple block sizes and varying queue length for random patterns with fixed group sizes of 1 and 2. This simulates a data layout where blocks are not particularly organized on disk or grouped by two, respectively. To our surprise, we noticed that a queue depth of 4 is sufficient to reach close to full bandwidth, but (in line with previous results) a minimum block size of 32KB is required. Random reads with blocks smaller than 32KB show better performance for an increasing queue depth of 16 but never deliver full bandwidth. For block sizes larger than 32KB we can see a slight almost linear performance gain up to 1MB, in total this is only about 5%. Reads of blocks larger than 1MB show a sudden performance drop, which at the first sight comes unexpected. However, it can be explained by the way block requests are passed to the device: Using *blktrace*[2] we found that requests of 1MB and larger are split into requests of 512KB before they are actually submitted to the device. So a 2MB read request with group size 1 is in truth 4 sequential read requests of 512KB - in our terms this would be comparable to group size 4 and block size 512KB. This split causes some overhead in the system that could contribute to the performance drop monitored, but we see the main reason for this behavior in the way the disks process sequential reads. We will explain this in the next paragraph about grouping as we also have to deal with this behavior for groups of 4 and larger. One last thing to notice is that random patterns with only one pending request at a time do not perform at the maximum possible speed for the tested block sizes. Patterns with two outstanding requests only perform optimal for 1MB blocks, but this again is in truth a pattern requesting 4 blocks of 512KB in size each and, thus, equivalent to queue depth 4 and block size 512KB.

Sequential patterns, on the other hand, perform more as expected in comparison to random patterns. As presented in Figure 4, increasing the queue depth is sufficient to reach full bandwidth. Also, simply increasing the block size leads to the same result. But even for sequential patterns, a queue depths of 4 is needed to get a good performance for small blocks (32KB). In contrast to random input patterns, increasing the queue depth further will also bring patterns with blocks smaller than 32KB close to full bandwidth. As queues of length 128 are needed for 4KB blocks, this may not be applicable in every system. Finally, for sequential patterns we should notice that the performance drop as seen for random patterns and block sizes larger than 1MB does not show. This is of interest for the explanation given in the next paragraph.

Summing up these observations, a queue depth of at least 4 is needed to get close to full bandwidth for block sizes between 32KB and 1MB.

Benefit from Grouping. First we want to notice that true sequential patterns indeed do not show a dependency on the number of blocks that are requested sequentially, so we omitted a figure for this case. But random patterns on the other hand show some surprising properties. As already mentioned before, block sizes larger than 1MB result in a bandwidth drop. We also explained that a single request of such large blocks is in truth a group of 4 or more sequential requests of size 512KB. Figure 5 shows for a queue depth of 4 that a random input pattern results in a significant penalty for group size 4 for all block sizes, no matter if the group of 4 sequential read requests is generated by asynchronous I/O itself or by splitting larger blocks (see graphs for group size 1 and 2). In Figure 6 we can see that this bandwidth drop is not bound to queue depth 4 but rather shows for all queues that fit into the drive's NCQ queue, i.e. queues of at most 32 elements. This is a very interesting behavior as sequential patterns in general perform well and a larger group size can be compared to (small) sequential patterns that are requested with a random offset. Also, if these small sequential patterns become longer, i.e. increasing group sizes larger than 4, bandwidth slowly recovers. This would be a positive effect of grouped requests if there was not the initial drop at group size 4. Figure 6 also shows that there is almost no difference, but if so, then only a slight decrease in bandwidth from group size 1 to group size 2. So, obviously for a solid state drive it is a difference if a system issues requests using multiple small sequential requests (groups) instead of a single or two requests with the matching block size, when the amount of "sequentially" read data is the same.

An explanation for this is hidden in the way read requests are processed. If the system issues a single read request of

[1] mdadm is a Linux tool used to manage software RAID devices, previously known as mdctl. See also: http://neil.brown.name/blog/mdadm

[2] a tool to trace block I/O in a system created by Jens Axboe

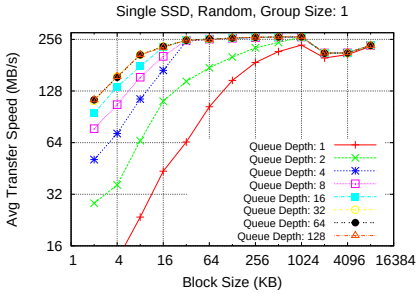

Figure 2: Bandwidth single SSD, group size 1, random access

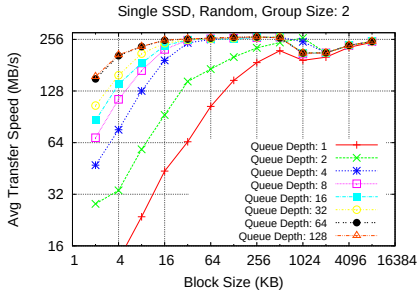

Figure 3: Bandwidth single SSD, group size 2, random access

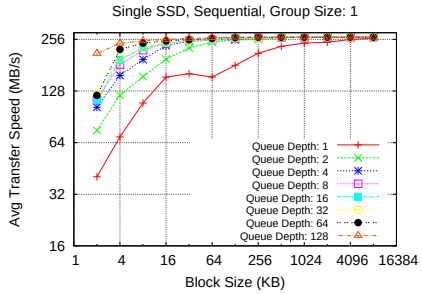

Figure 4: Bandwidth single SSD, group size 1, sequential access

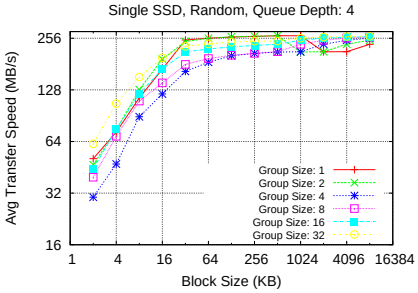

Figure 5: Bandwidth single SSD, queue depth 4, random access

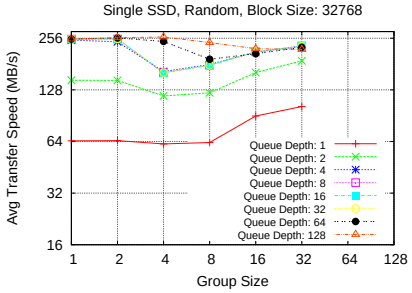

Figure 6: Bandwidth single SSD, block size 32KB, random access

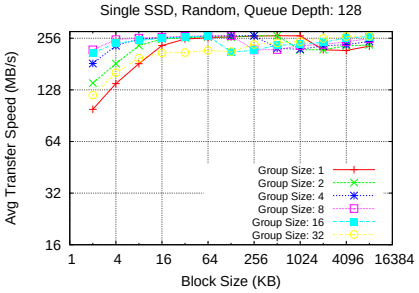

Figure 7: Bandwidth single SSD, queue depth 128, random access

512KB and smaller, the request is directly passed to the device and processed there. In case the request is larger, the request is split into 512KB requests, which are then passed to the device. So any request larger than 512KB is seen by the disk as multiple sequential 512KB requests. Now, if the device gets 4 or more sequential requests, it switches into a sequential read mode, where more blocks are retrieved by a prefetching mechanism. This particular behavior was also discovered by [2]. But, as the sequential pattern only lasts for a small number of blocks, the bandwidth taken by the prefetching mechanism cannot be amortized by the faster processing of the few sequential requests and, thus, results in an overall penalty as (parts of) the prefetched blocks are not used. For larger groups, though, the "wasted" prefetching can be amortized the larger the groups are and the performance of random patterns slowly recovers and becomes comparable to sequential patterns again, following the general expectation. Interestingly, this behavior vanishes for queues longer than 32 as can be seen in Figure 6 or when we compare Figures 5 and 7. However, 1MB again is the maximum request size (block size $*$ group size) with high bandwidth, as larger requests inherit a significant performance penalty comparable to the one seen before. Taking a closer look shows, that the penalty inherited by grouping 4 blocks is compensated by queues of 32, group size 8 by a queue depth of 64, and group size 16 by a queue depth of 128. (Figures for queue size 32 and 64 are not presented.) So the factor of 8 is a constant for each pattern. It seems as if the system is able to detect the right group size if a queue is 8 times as large. Looking at the numbers for the drive where the command queue length is described to be 32 this is hard to believe, though.

In Section 4 we mentioned a second way of how a system can benefit from grouped data layout - a flexible I/O unit. By looking at Figures 2 and 3 we already noticed that 32KB is sufficiently large (for queues of 4 and larger) to get high bandwidth, so 32KB should be the product of block size $*$ group size. The remaining question now is, how small the block size can be chosen for random access.

To answer this question, we need to look at the graphs for queue depth 4. A block size of 2KB gives a bandwidth of about 50MB/s. This is only about 1/5 of the maximal achievable bandwidth. As already mentioned 32KB block size is needed to get this full bandwidth around 250MB/s. however, this is 16 times the block size of 2KB, with only a reduction in bandwidth of 1/5. Thus, 2KB is still beneficial, if only a 2KB block out of the 32KB block is needed and the I/O unit should just request the 2KB block. In fact, as bandwidth always drops less than half, if only half the block size is requested, the I/O unit should always choose the smaller block size for a request. However, if multiple non-consecutive blocks of a 32KB block are requested, additional intelligence is needed. For example if every other 2KB block is required, leading to a total request size of 16KB, the system should request the complete group of 32KB in one piece, as here the gain through higher bandwidth is larger than the penalty for requesting twice the amount of data.

Summing up these observations, a grouped data layout on disk can be beneficial for random patterns, but only if a system provides dynamic block sizes for disk access. In that case of block size of 2KB works well and should be combined with a group size of 16, leading to a combined group block size of 32KB. Sequential patterns are not affected by grouping.

Random=Sequential. A last expectation from the combination of SSDs and asynchronous I/O was that we can find settings where random and sequential patterns equally give

full bandwidth. As we have seen in the previous paragraph, group sizes of 4 and larger result in lower bandwidth. This behavior cannot be monitored for sequential patterns. So in order to achieve high bandwidth for both cases, a group size smaller than 4 needs to be chosen. Looking again at Figures 2 and 4, we can see that the range of potentially interesting (meaning fast) block sizes becomes wider for queues of length 4 and larger. There the interval of 32KB to 1MB for a request can give equally fast random and sequential I/O. For group size 2 (see Figure 3) the maximum block size is limited to 512KB for reasons described above. Also if very small blocks are required by a system, 16KB becomes an interesting block size at the price of larger queues (32 and larger).

Summing up these observations, to achieve high bandwidth, equally for random and sequential patterns, the group size should be chosen to be 1 or 2, the queue depth should at least be 4 and the block size can range from 32KB to 1MB. In the other cases sequential patterns outperform random ones.

5.4 4 Disk Raid

For our 4 disk RAID experiments, DIAT created a 256GB file of which 8GB were requested for each run. The RAID was created using *mdadm* with a *chunk size* of 16KB, 64KB and 256KB leading to stripe sizes of 64KB, 256KB and 1024KB, respectively, as we have four disks. We focus on a chunk size of 64KB, but also present differences to RAID setups using smaller or larger setups where of interest.

Hight Parallelism. As expected the need for parallel requests is even higher if data is stored in a RAID. Figures 8 and 9 show (for stripe sizes 256KB and 1024KB) the bandwidth measured by DIAT for a fixed group size of 1 and varying block sizes and queue depths. We can see that higher queue depth in general leads to higher bandwidth. Where a queue depth of 4 was sufficient for the single SSD, the RAID needs queues of length 16 and up if blocks are smaller than the stripe size. This is not surprising, as read requests smaller than the stripe size are not directed to all devices. So in order to achieve the same degree of parallelism at the device, more requests need to be issued to the RAID the smaller the requests are. Also block sizes smaller than the RAID chunk size are not suitable for fast random reads unless very large queues (64 and up) are used. If we compare Figures 8 and 9, we can see a larger increase of the average transfer speed starting at exactly 64KB and 256KB, respectively. For blocks larger than the stripe size a queue depth of 4 is sufficient which is similar to the single SSD case, as each of these requests is split into at least four device requests which are distributed among the four disks. Similar to the single SSD experiments we can find the bandwidth drop for large block sizes. This time blocks 8MB and larger are affected, as this leaves at least 2MB requests per device. As the I/O scheduler merges the chunk sized requests before sending them to the device (see [7]), we get the same behavior as described for single SSDs, where requests of 1MB and larger are split into 512KB requests and the disk detects the sequential patterns.

Figure 10 shows the same experiment for sequential patterns. Again, increasing queue depth or/and increasing block size leads to better I/O performance. But in order to achieve close to full bandwidth the block size needs to be at least the RAID chunk size or queues need to be reasonably large. Comparable to random patterns, block sizes equivalent to the stripe size or larger provide good performance for shorter queues. Interestingly, and in particular different from the single SSD experiment, block sizes of 4MB and larger show a slightly reduced bandwidth similar to the random patterns.

Summing up these observations, for a RAID of 4 disks a queue depths of 32 in combination with block sizes between the RAID chunk and stripe size will deliver high bandwidth. If blocks are larger than the RAID stripe size, a queue depth 4 is sufficient.

Benefits of Grouping. Again sequential patterns are not affected by the group size. As previously described, a queue depth of 32 delivers good performance for a wide range of block sizes. Figures 11, 12 and 13 show the bandwidth measured by DIAT for this queue depth for the three different RAID setups. Similar to what we have seen for the single SSD, requests of groups of 4 and larger are processed at significantly lower bandwidth, but only for block sizes of at least the RAID stripe size. This is not a surprise, as these requests lead to grouped, a.k.a. sequential, requests of at least four blocks at each device. With an increasing group size this bandwidth drop can be compensated but not fully recovered.

However, for block sizes smaller than the stripe size, a group size of four shows the best performance. Again, the explanation is simple, as grouping for smaller blocks ensures that requests are distributed equally among the disks. For example a grouped request of four blocks at a block size half the stripe size guarantees two requests per device, where in comparison four random requests do not necessarily go to all four devices. However, when grouped requests of blocks smaller than the stripe size are issued, a system needs to make sure that not four or more sequential requests are issued per device. For example, requests with a block size of 128KB are only slowed down by groups of 8 and larger on the RAID with 256KB stripe size and for block sizes of 64KB this effect shows for groups of 16 and larger. So, different from the single SSD case, grouping in combination with sequential block requests can actually be beneficial in a RAID setup.

Similar to the single SSD case, the RAID setup also benefits from a grouped data layout in combination with a dynamic block size. A block size of 4KB for the RAID setup is still efficient, if only data at this granularity is needed, as the ascent of all graphs is less than one. So 4KB could be the choice for the random access block size. However, the appropriate block size for larger requests depends on the degree of parallelism supported by the system and the query load. So for a queue depth of 4 a combined block size larger than the stripe size is required and for queues of length 32 or longer the combined block size only needs to be larger than the RAID chunk size.

Summing up these observations, in a RAID setup with 4 disks I/O can also benefit from grouping in two ways. In case of a fixed block size, a group size of four shows the highest bandwidth for block sizes half or a quarter of the stripe size. In case of a dynamic block size for disk access 4KB can be used for fine granular random access, where disk blocks should be grouped in the order of the RAID stripe size.

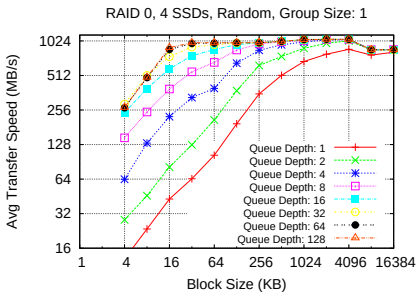

Figure 8: Bandwidth RAID, stripe size 256KB, group size 1, random access

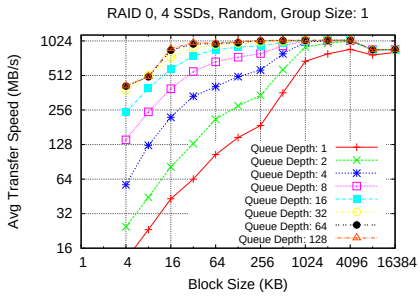

Figure 9: Bandwidth RAID, stripe size 1024KB, group size 1, random access

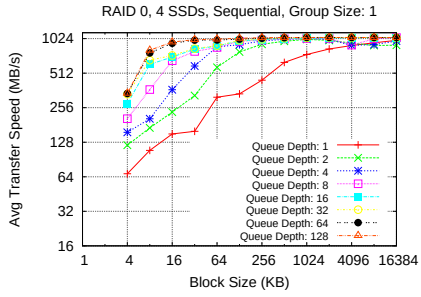

Figure 10: Bandwidth RAID, stripe size 256KB, group size 1, sequential access

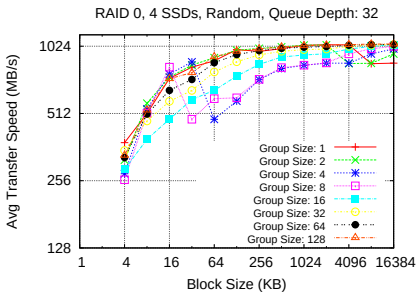

Figure 11: Bandwidth RAID, stripe size 64KB, queue depth 32, random access

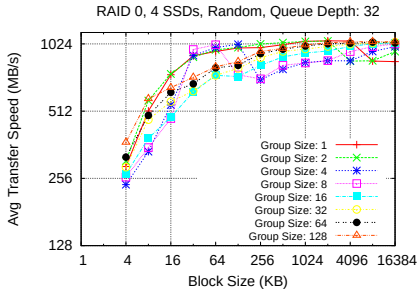

Figure 12: Bandwidth RAID, stripe size 256KB, queue depth 32, random access

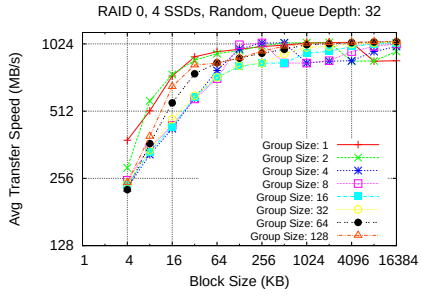

Figure 13: Bandwidth RAID, stripe size 1024KB, queue depth 32, random access

Random=Sequential. Similar to the single SSD case, we can find settings where both patterns show equally good performance. The parameter space for the RAID setup is in principle only restricted by the random pattern, as sequential patterns in general perform slightly better. This leads to the settings found in the first paragraph of this section to achieve equally high bandwidth for random and sequential patterns.

6. CONCLUSIONS

Concluding we can say that much of the potential hidden in solid state disks can be extracted for database use by applying asynchronous I/O. In particular the combination asynchronous I/O with the SSD property of fast random reads opens the door for very fine grained clustering structures. In Section 3 we introduced grouping for column stores. In our evaluation we show, that this grouping approach in combination with dynamic block sizes for disk access can provide very fine granular access without loosing performance for more coarse data requests. However, our analysis also pointed out, that the necessary parameters need to be chosen with care. As we only tested the X-25M, we have to leave the question open if our results are portable to other types of SSDs, especially as the market is still growing and SSD technology is still under heavy development.

7. REFERENCES

[1] L. Bouganim, B. T. Jónsson, and P. Bonnet. uflip: Understanding flash io patterns. In *CIDR'09*, 2009.

[2] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS '09*, pages 181–192, New York, NY, USA, 2009. ACM.

[3] G. Graefe. The Five-minute Rule 20 Years Later (and how Flash Memory Changes the Rules). *Communications of the ACM*, 52(7):48–59, 2009.

[4] Intel Corporation. *Intel X18-M/X25-M SATA Solid State Drive - Product Manual.* http://download.intel.com/design/flash/nand/mainstream/mainstream-sata-ssd-datasheet.pdf.

[5] Intel Corporation. *Intel X25-E SATA Solid State Drive - Product Manual.* http://download.intel.com/design/flash/nand/extreme/319984.pdf.

[6] S.-W. Lee and B. Moon. Design of Flash-based DBMS: an In-Page Logging Approach. In *SIGMOD'07*, pages 55–66, 2007.

[7] R. Love. *Linux Kernel Development (2nd Edition) (Novell Press)*. Novell Press, 2005.

[8] S. Nath and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *6th Int. Conf. on Information Processing in Sensor Networks (IPSN) 2007*, pages 410–419, 2007.

[9] P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In R. O. Nambiar and M. Poess, editors, *TPCTC*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2009.

[10] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, and M. Huras. Multi-dimensional Clustering: a New Data Layout Scheme in DB2. In *SIGMOD '03*, pages 637–641, New York, NY, USA, 2003. ACM.

[11] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[12] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query Processing Techniques for Solid State Drives. In *SIGMOD'09*, pages 59–72, 2009.

[13] C.-H. Wu, T.-W. Kuo, and L.-P. Chang. An Efficient B-tree Layer Implementation for Flash-memory Storage Systems. *ACM Trans. Embedded Comput. Syst.*, 6(3), 2007.

[14] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. Monetdb/x100 - a dbms in the cpu cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.