Query Processing on Smart SSDs: Opportunities and Challenges

Jaeyoung Do[+,#], Yang-Suk Kee[*], Jignesh M. Patel[+],
Chanik Park[*], Kwanghyun Park[+], David J. DeWitt[#]

[+]University of Wisconsin – Madison; [*]Samsung Electronics Corp.; [#]Microsoft Corp.

ABSTRACT

Data storage devices are getting "smarter." Smart Flash storage devices (a.k.a. "Smart SSD") are on the horizon and will package CPU processing and DRAM storage inside a Smart SSD, and make that available to run user programs inside a Smart SSD. The focus of this paper is on exploring the opportunities and challenges associated with exploiting this functionality of Smart SSDs for relational analytic query processing. We have implemented an initial prototype of Microsoft SQL Server running on a Samsung Smart SSD. Our results demonstrate that significant performance and energy gains can be achieved by pushing selected query processing components inside the Smart SSDs. We also identify various changes that SSD device manufacturers can make to increase the benefits of using Smart SSDs for data processing applications, and also suggest possible research opportunities for the database community.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – Query Processing

General Terms

Design, Performance, Experimentation.

Keywords

Smart SSD.

1. INTRODUCTION

It has generally been recognized that for data intensive applications, moving code to data is far more efficient than moving data to code. Thus, data processing systems try to push code as far below in the query processing pipeline as possible by using techniques such as early selection pushdown and early (pre-)aggregation, and parallel/distributed data processing systems run as much of the query close to the node that holds the data.

Traditionally these "code pushdown" techniques have been implemented in systems with rigid hardware boundaries that have largely stayed static since the start of the computing era. Data is pulled from an underlying I/O subsystem into the main memory, and query processing code is run in the CPUs (which pulls data from the main memory through various levels of processor caches). Various areas of computer science have focused on making this data flow efficient using techniques such as prefetching, prioritizing sequential access (for both fetching data to the main memory, and/or to the processor caches), and pipelined query execution.

However, the boundary between persistent storage, volatile storage, and processing is increasingly getting blurrier. For example, mobile devices today integrate many of these features into a single chip (the SoC trend). We are now on the cusp of this hardware trend sweeping over into the server world. The focus of this project is the integration of processing power and non-volatile storage in a new class of storage products known as *Smart SSDs*. Smart SSDs are flash storage devices (like regular SSDs), but ones that incorporate memory and computing inside the SSD device. While SSD devices have always contained these resources for managing the device for many years (e.g., for running the FTL logic), with Smart SSDs some of the computing resources inside the SSD could be made available to run general user-defined programs.

The focus of this paper is to explore the opportunities and challenges associated with running selected database operations inside a Smart SSD. The potential opportunities here are threefold.

First, SSDs generally have a far larger aggregate internal bandwidth than the bandwidth supported by common host I/O interfaces (typically SAS or SATA). Today, the internal aggregate I/O bandwidth of high-end Samsung SSDs is about 5X that of the fastest SAS or SATA interface, and this gap is likely to grow to more than 10X (see Figure 1) in the near future. Thus, pushing operations, especially highly selective ones that return few result rows, could allow the query to run at the speed at which data is getting pulled from the internal (NAND) flash chips. We note that similar techniques have been used in IBM Netezza and Oracle Exadata appliances, but these approaches use additional or specialized hardware that is added right into or next to the I/O subsystem (FPGA for Netezza [12], and Intel Xeon processors in Exadata [1]). In contrast, Smart SSDs have this processing in-built into the I/O device itself, essentially providing the opportunity to "commoditize" a new style of data processing where operations are opportunistically pushed down into the I/O layer using commodity Smart SSDs.

Second, offloading work to the Smart SSDs may change the way in which we build balanced database servers and database appliances. If some of computation is done inside the Smart SSD, then one can reduce the processing power that is needed in the host machine, or increase the effective computing power of the servers or appliances. Smart SSDs use simpler processors, like ARM, that are generally cheaper (from the $/MHz perspective)

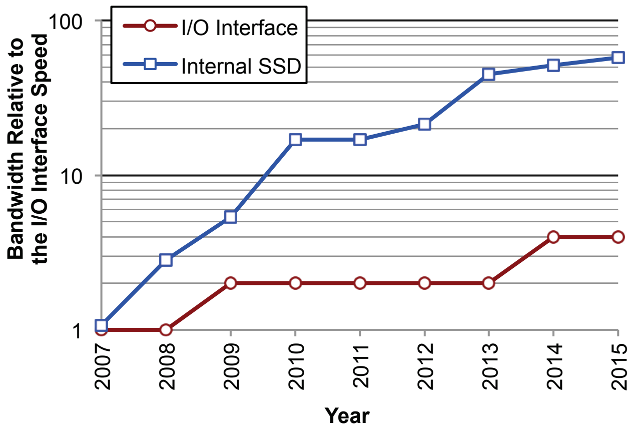

Figure 1: Bandwidth trends for the host I/O interface (i.e., SAS/SATA standards), and aggregate internal bandwidth available in high-end enterprise Samsung SSDs. Numbers here are relative to the I/O interface speed in 2007 (375 MB/s). Data beyond 2012 are internal projections by Samsung.

than the traditional processors that are used in servers. Thus, database servers and appliances that use Smart SSDs could be more efficient from the overall price/performance perspective.

Finally, pushing processing into the Smart SSDs can reduce the energy consumption of the overall database server/appliance. The energy efficiency of query processing can be improved by reducing its running time and/or by running processing on the low power processors that are typically packaged inside the Smart SSDs. Lower energy consumption is not only environmentally friendly, but often leads to a reduction in the total cost of operating the database system. In addition, with the trend towards database appliances, energy starts becoming an important deployment consideration when the database appliances are installed in private clouds on premises where getting additional (many kilowatts of) power is challenging.

To explore and quantify these potential advantages of using Smart SSDs for DBMSs, we have started an exploratory project to extend Microsoft SQL Server to offload database operations onto a Samsung Smart SSD. We wrote simple selection and aggregation operators that are compiled into the firmware of the SSD. We also extended the execution framework of SQL Server to develop a simple (but with limited functionality) working prototype in which we could run simple selection and aggregation queries end-to-end.

Our results show that for this class of queries, we observed up to 2.7X improvement in end-to-end performance compared to using the same SSDs but without the "Smart" functionality, and up to a 3.0X reduction in energy consumption. These early results, admittedly on queries using a limited subset of SQL (e.g., no joins), demonstrate that there are potential opportunities for using Smart SSDs even in mature commercial and well-optimized relational DBMSs.

Our results also point out that there are a number of challenges, and hence research opportunities, in this new area of running data processing programs inside the Smart SSDs.

First, the processing capabilities available inside the Smart SSD that we used are very limited by design. It is clear from our results that adding more computing power into the Smart SSD (and making it available for query processing) could further increase

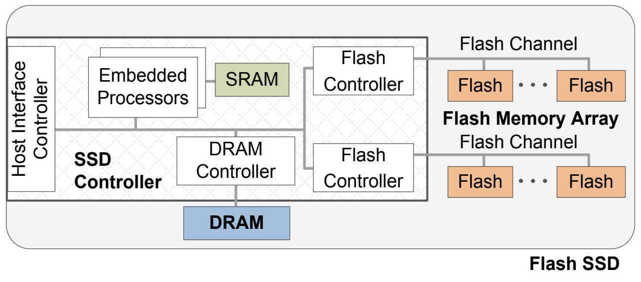

Figure 2: Internal architecture of a modern SSD

both performance and energy savings. However, the SSD manufacturers will need to determine if it is economical and technically feasible to add more processing power – issues such as the additional cost per device and changes in the device energy profile must be considered. In a sense, this is a chicken-and-egg problem since the SSD manufacturers will add more processing power only if more software makes use of an SSD's "smart" features while the software vendors need to become confident in the potential benefits before investing the necessary engineering resources. We hope that our work provides a starting point for such deliberations.

Second, the firmware development process we followed to run user code in the Smart SSDs is rudimentary. This can be a potential challenge for general application developers. Before Smart SSDs can be broadly adopted, the existing development and debugging tools and runtime system (Section 3) need to be much more user-friendly. Further, the ecosystem around the Smart SSDs including communication protocols and the programming, runtime, and usage models need to be investigated in-depth.

Finally, the query execution engine and query optimizer of the DBMS must be extended to determine when to push an operation to the SSD. Implications of running operations in the Smart SSDs also extend out to query optimization, DBMS buffer pool caching policies, transaction processing, and may require re-examining how aspects such as database compression are used. In other words, the DBMS internals have to be modified to make use of Smart SSDs in a production setting.

The remainder of this paper is organized as follows: The architecture of a modern SSD is presented in Section 2. In Section 3 we describe how Smart SSDs work. Experimental results are presented in Section 4. Related work is discussed in Section 5. Finally, Section 6 contains our concluding remarks and points to some directions for future work.

2. BACKGROUND: SSD ARCHITECTURE

Figure 2 illustrates the general internal architecture of a modern SSD. There are three major components: SSD controller, flash memory array, and DRAM.

The SSD controller has four key subcomponents: host interface controller, embedded processors, DRAM controller, and flash memory controllers. The host interface controller implements a bus interface protocol such as SATA, SAS, or PCI Express (PCIe). The embedded processors are used to execute the SSD firmware code that runs the host interface protocol, and also runs the Flash Translation Layer (FTL), which maps Logical Block Address (LBA) in the host OS to the Physical Block Address (PBA) in the flash memory. Time-critical data and program code are stored in the SRAM. Today, the processor of choice is typically a low-powered 32-bit RISC processor, like an ARM series processor,

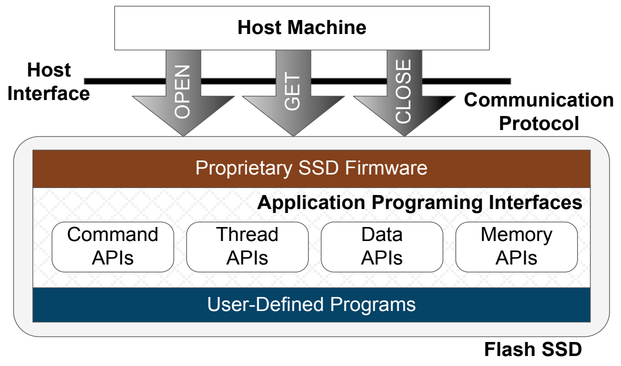

Figure 3: Smart SSD runtime framework

which typically has multiple cores. The controller also has on-board DRAM memory that has higher capacity (but also higher access latency) than the SRAM.

The flash memory controller is in charge of data transfer between the flash memory and DRAM. Its key functions include running the Error Correction Code (ECC) logic, and the Direct Memory Access (DMA). To obtain higher I/O performance from the flash memory array, the flash controller uses chip-level and channel-level interleaving techniques. All the flash channels share access to the DRAM. Hence, data transfers from the flash channels to the DRAM (via DMA) are serialized.

The NAND flash memory array is the persistent storage medium. Each flash chip has multiple blocks, each of which holds multiple pages. The unit of erasure is a block, while the read and write operations in the firmware are done at the granularity of pages.

3. SMART SSDs FOR QUERY PROCESSING

The Smart SSD runtime framework (shown in Figure 3) implements the core of the software ecosystem that is needed to run user-defined programs in the Smart SSDs.

3.1 Communication Protocol

Since the key concept of the Smart SSD (that we explore in this paper) is to convert a regular SSD into a combined computing and storage device, we needed a standard mechanism to enable the processing capabilities of the device at run-time. We have developed a simple session-based protocol that is compatible with the standard SATA/SAS interfaces (but could be extended for PCIe). The protocol consists of three commands – OPEN, GET, and CLOSE.

- OPEN, CLOSE: A session starts with an OPEN command and terminates with a CLOSE command. Once the session starts, runtime resources including threads and memory (see Thread and Memory APIs in Section 3.2) that are required to run a user-defined program are granted, and a unique session id is then returned to the host. Note that when one of the other Smart SSD commands (i.e., GET and CLOSE) is invoked by the host, the session id must be provided to find the corresponding session before the command is executed in the Smart SSDs. The CLOSE command closes the session associated with the session id; it terminates any running program and releases all resources that are used by the program. Once the session is closed, the corresponding session id is invalid, and can be recycled.
- GET: The host can monitor the status of the program and retrieve results that the program generates via a GET command. This command is mainly designed for the traditional block devices (based on SATA/SAS interfaces), in which case the storage device is a passive entity and responds only when the host initiates a request. For PCIe, a more efficient command (such as PULL) could be introduced to directly leverage device-initiated capabilities (e.g., interrupts). A single GET command retrieves both the running status of the program and the results if the output is ready. With different session ids, multiple user-defined programs can be executed in parallel. Note that the programs can be blocked if no resource is available in the Smart SSD. Therefore, the polling interval should be adaptive so that it does not introduce a large polling overhead or hinder the progress of the Smart SSD operations. In our experiments, the polling interval was set to 10 msec.

3.2 Application Programming Interface (API)

Once a command has been successfully delivered to the device through the Smart SSD communication protocol (Section 3.1), the Smart SSD runtime system drives the user-defined program in an event-driven fashion. The user program can use the Smart SSD APIs for command management, thread management, memory management, and data management. The design philosophy of the APIs is to give more flexibility to the program, so that it is easier for the end-user programs to use these APIs. These APIs are briefly described below.

- ***Command APIs***: Whenever a Smart SSD command (i.e., OPEN, GET, and CLOSE) is passed to the device, the Smart SSD runtime system invokes the corresponding callback function(s) registered by the user-defined program. For instance, the OPEN and CLOSE commands trigger user-defined *open* and *close* functions respectively. In contrast, the GET command calls functions to fill the running status of the program and to transfer results to the host if available.
- ***Thread APIs***: Once a session is opened, the Smart SSD runtime system creates a set of worker threads and a master thread per core dedicated to the session. All threads managed by the runtime system are non-preemptive. A worker thread is scheduled when a Smart SSD command arrives (see *Command APIs* above), or when a data page (8KB) is loaded from flash to DRAM (see *Data APIs* below). Once scheduled, a user-registered callback function for that event is invoked on the thread (e.g., an *open* function in the event of the OPEN command). Since callback functions are designed to be "quick" functions, long-running operations that are required for each page (such as filtering) are handled by a special function that is executed in the master thread. We note that the current version of the runtime system does not support a "yield" command that gives up the processor to other threads. To simulate this behavior when the master thread is scheduled, the operation processes only a few pages, before the master thread is rescheduled to deal with the next task (which could be to process the next set of pages for the first task).
- ***Memory APIs***: Smart SSD devices typically have two types of memory modules – a small fast SRAM (e.g., ARM's Tightly Coupled Memory), and a large slow DRAM. In a typical scenario, the DRAM is mainly used to store data pages while the SRAM is used for frequently accessed metadata such as the database table schema. Once a session

is open, a pre-defined amount of memory is assigned to the session, and this memory is returned back to the Smart SSD runtime system when the session is closed (i.e., dynamic memory allocation using *malloc* and *free* is not allowed.)

- ***Data APIs***: Multiple data pages can be loaded from flash to DRAM in parallel. Here, the degree of parallelism depends on the number of flash channels employed in the Smart SSD. Once loaded, the pages are pinned to ensure that they are not evicted from the DRAM. After processing a page, it must be unpinned to release the memory required to hold the page back to the device. Otherwise, Smart SSD operations might be blocked until enough memory is available for the subsequent operations.

4. EVALUATION

In this section, we present results from an empirical evaluation of Smart SSD with Microsoft SQL Server.

4.1 Experimental Setup

4.1.1 Workloads

For our experiments, we used the LINEITEM table defined in the TPC-H benchmark [30] and three synthetic tables (Synthetic4, Synthetic16, and Synthetic64) that consist of 4 integer columns, 16 integer columns, and 64 integer columns respectively.

Our modifications to the original LINEITEM table specifications are as follows:

1) We used a fixed-length char string for the variable-length column, L_COMMENT,
2) All decimal numbers were multiplied by 100 and stored as integers,
3) All date values were converted to the numbers of days since the last epoch.

These changes resulted in 148 byte-sized tuples. The LINEITEM data was populated at a scale factor of 100 (600M tuples, ~90GB).

In addition, we created three synthetic tables, called Synthetic4, Synthetic16, and Synthetic64, each of which has 400M tuples. The sizes of these tables are 10GB, 30GB and 110GB for the Synthetic4, the Synthetic16, and the Synthetic64 tables respectively.

The data in the LINEITEM table and the synthetic tables was inserted into a SQL Server heap table (without a clustered index). By default, the tuples in these tables were stored in slotted pages using the traditional N-ary Storage Model (NSM). For the Smart SSDs, we also implemented the PAX layout [3] in which all the values of a column are grouped together within a page.

4.1.2 Hardware/Software Setup

All experiments were performed on a system running 64bit Windows 7 with 32GB of DRAM (24 GB of memory is dedicated to the DBMS). The system has two Intel Xeon E5430 2.66GHz quad core processors, each of which has a 32KB L1 cache, and two 6MB L2 caches shared by two cores. For the OS and the transactional log, we used two 7.5K RPM SATA HDDs, respectively. In addition, we used a LSI four-port SATA/SAS 6Gbps HBA (host bus adapter) [22] for the three storage devices that we used in our experiments. These three devices are:

1) A 146GB 10K RPM SAS HDD,
2) A 400GB SAS SSD, and
3) A Smart SSD prototyped on the same SSD as above.

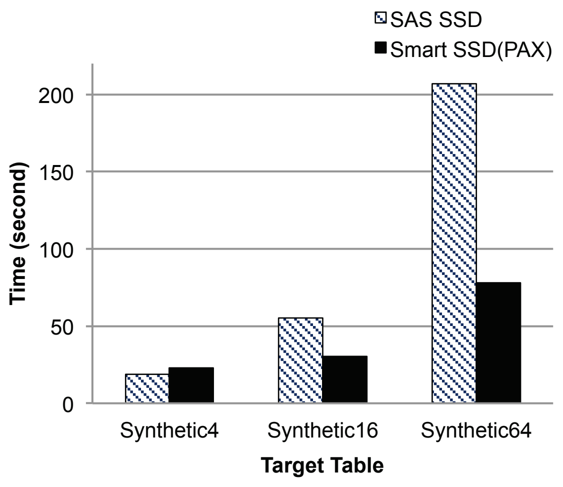

Figure 4: End-to-end elapsed time for a selection query at a selectivity of 0.1% with the three synthetic tables Synthetic4, Synthetic16, and Synthetic64.

Only one of three devices is connected to the HBA at a time for each experiment. Finally, the power drawn by the system was measured using a Yokogawa WT210 unit (as suggested in [26]). We used this server hardware since it was compatible with the LSI HBA card that was needed to run the extended host interface protocol described in Section 3.1.

We recognize that this box has a very high base energy profile (235W in the idle state) for our setting in which we use a single data drive; hence, we expect the energy gains to be bigger when the Smart SSD is used with a more balanced hardware configuration. But, this configuration allowed us to get initial end-to-end results.

We implemented simple selection and selection with aggregation queries in the Smart SSD by using the Smart SSD APIs (Section 3.2). We also modified some components in SQL Server 2012 [23] to recognize and communicate with the Smart SSD through the Smart SSD communication protocol (Section 3.1). For each test, we measured the elapsed wall-clock time, and calculated the disk energy consumption by summing the time discretized real energy values over the elapsed time. After each test run, we dropped the pages in the main-memory buffer pool to start with a cold buffer cache on each run. Thus, all the results presented here are for *cold* experiments; i.e., there is no data cached in the buffer pool prior to running each query.

4.2 Experimental Results

To aid the analysis of the results that are presented below, the I/O characteristics of the HDD, SSD, and Smart SSD are shown in Table 1. The bandwidth of the HDD and the SSD was obtained using Iometer [15]. For the Smart SSD internal bandwidth, we implemented a simple program (by using the Smart SSD APIs introduced in Section 3.2) to measure the wall clock time to sequentially fetch a 100GB dummy data file from flash to the on-board DRAM. Note that for this experiment, there was no data

Table 1: Maximum sequential read bandwidth with 32-page (256KB) I/Os.

	SAS HDD	*SAS SSD*	*(Internal) Smart SSD*
Seq. Read (MB/sec)	80	550	1,560

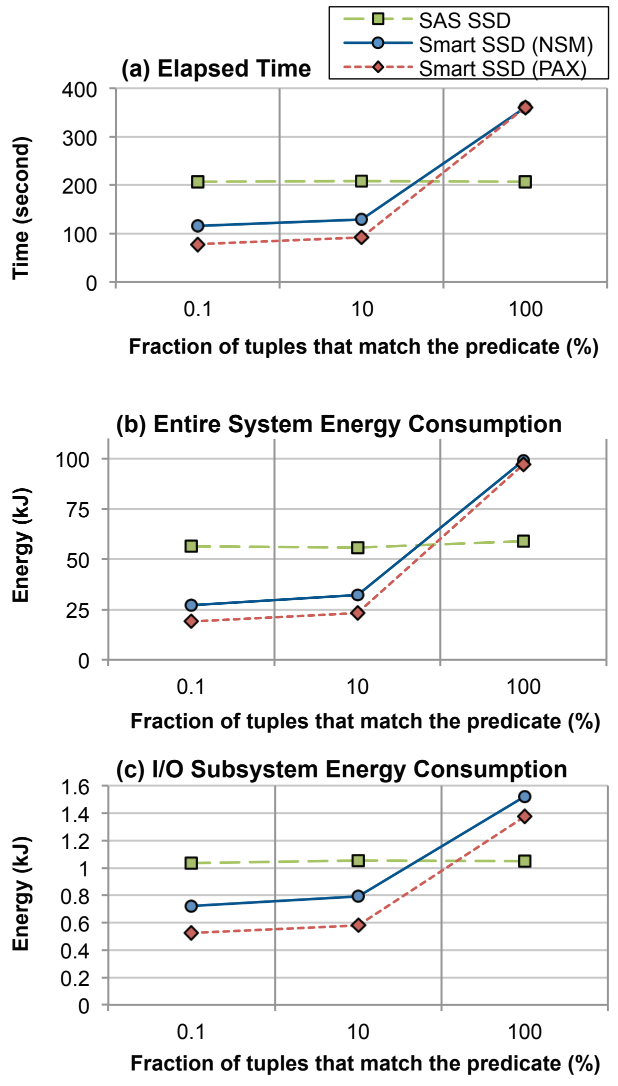

Figure 5: End-to-end (a) query execution time, (b) entire system energy consumption, and (c) I/O subsystem energy consumption for a selection query on the Synthetic64 table at various selectivity factors.

transfer between the SSD and the host. The only traffic between the host and the Smart SSD was the communication associated with issuing the Smart SSD commands (i.e., OPEN, GET, and CLOSE) to control the program.

As can be seen in Table 1, the internal sequential read bandwidth of the Smart SSD is 19.5X and 2.8X faster than that of the HDD and the SSD, respectively. This value can be used as the upper bound of the performance gains that this Smart SSD could potentially deliver. As described in Figure 1, over time it is likely that the gap between the SSD and the Smart SSD will grow to a much larger number than 2.8X.

We also note that the improvement here (of 2.8X) is far smaller than the gap shown in Figure 1 (about 10X). The reason for this gap is that the access to the DRAM is shared by all the flash channels, and currently in this SSD device only one channel can be active at a time (recall the discussion in Section 2), which becomes the bottleneck. One could potentially address this bottleneck by increasing the bandwidth to the DRAM or adding more DRAM busses. As we discuss below, this and other issues must be addressed to realize the full potential of the Smart SSD vision.

Table 2: Results for the SAS HDD: End-to-end query execution time, entire system energy consumption, and I/O subsystem energy consumption for a selection query on the Synthetic64 table at various selectivity factors.

	0.1%	***10%***	***100%***
Elapse time (seconds)	1,494	1,486	1,485
Entire System Energy (kJ)	357	358	358
I/O Subsystem Energy (kJ)	13	13	13

4.2.1 Selection Query

For this experiment, we used three synthetic tables and the following SQL query:

```
SELECT SecondColumn
FROM SyntheticTable
WHERE FirstColumn < [VALUE]
```

Effect of Tuple Size: Figure 4 shows the end-to-end elapsed time to execute the selection query at a selectivity of 0.1% with the three synthetic tables (Synthetic4, Synthetic16, and Synthetic64). As can be seen in this figure, the Smart SSD, with a PAX layout, executes the selection query on the Synthetic64 table **2.6X** faster than the regular SSD, whereas the selection on the Synthetic4 table is slower than the regular SSD. The performance improvement of the Smart SSD comes from the faster internal I/O, whereas the low computation power of the ARM core in the Smart SSD saturates its performance. In this experiment, in all the three cases, the Smart SSD improves the I/O component of fetching data from the flash chips. But, compared to the regular SSD case, the Smart SSD has to compute on the data in the pages that are fetched from the flash chips before sending it to the host. With the Synthetic64 data set, this computation cost (measured as cycles/page) is low as there are only 29 tuples on each page. However, with the Synthetic4 table, there are 323 tuples on each data page, and the Smart SSD-based execution strategy now has to spend far more processing cycles per page, which saturates the CPU. Now, the query (on the Synthetic4 table) in the Smart SSD is bottlenecked on the CPU resource. In the case of this SSD device, for the Synthetic4 data set, the throughput of the computation that can be pushed "through the CPU" in the Smart SSD is lower than the host IO interface. Consequently, the performance of this query (with 0.1% selectivity) is faster with the regular SSD.

Effect of Varying the Selectivity Factor: Figures 5 (a), 5 (b), and 5 (c) present the end-to-end elapsed time and the energy consumed when executing the selection query at various selectivity factors on the Synthetic64 table, using the regular SSD, and the Smart SSD with the default NSM layout and the PAX layout. The energy consumption is shown for the entire system in Figure 5 (b), and for just the I/O subsystem in Figure 5 (c).

To improve the presentation of these figures, we do not show the measurements for the HDD case, as it was significantly higher than the SSD cases. Rather, we show the measurements for the HDD case in Table 2.

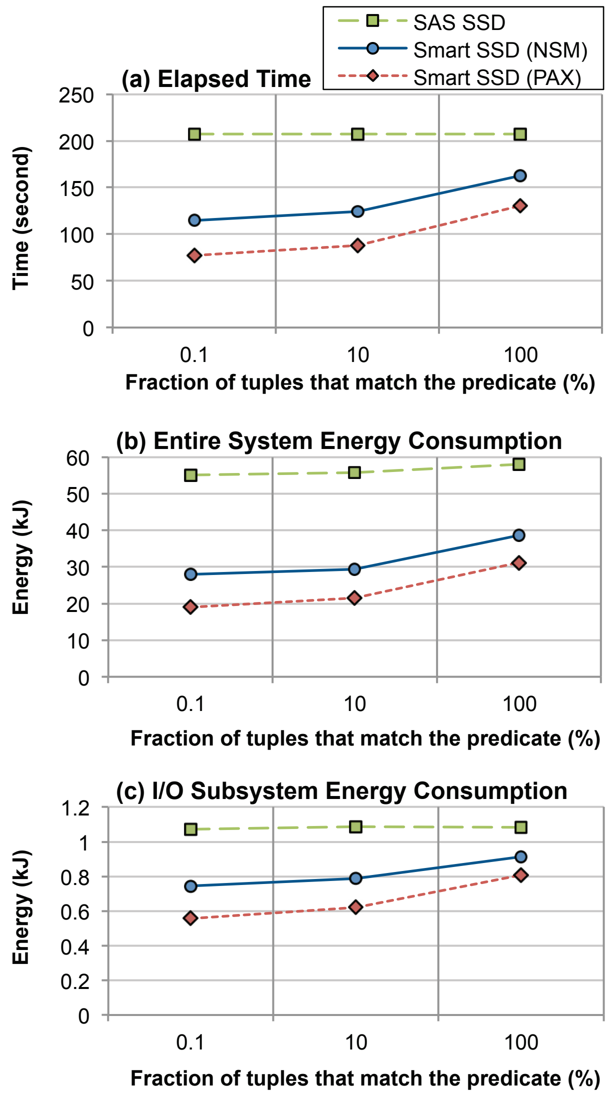

Figure 6: End-to-end (a) query execution time, (b) entire system energy consumption, and (c) I/O subsystem energy consumption for a selection with aggregate query on the Synthetic64 table at various selectivity factors.

As can be observed from Figure 5 (a) and Table 2, the Smart SSD provides significant improvements in performance for the highly selective queries (i.e. when few tuples match the selection predicate). The improvements are **19X** and **2.6X** over the HDD and the SSD, respectively when 0.1% of the tuples satisfy the selection predicate.

One interesting observation from Figure 5 (a) is that for the Smart SSD case, using the PAX layout provides better performance than the NSM layout, by up to 32%. As an example, for the 0.1% selection query, the elapsed times when using NSM and PAX are about 115 seconds and 78 seconds, respectively. Unlike the host processor that has L1/L2 caches, the embedded processor in our Smart SSD does not have these caches. Instead, it provides an efficient way to move consecutive bytes from the memory to the processor registers in a single instruction, called the LDM instruction [4][1]. Since all the values of a column in a page are stored contiguously in the case of the PAX layout, we were able to use the LDM instruction to load multiple values at once, reducing the number of (slow) DRAM accesses. Given the high DRAM latency in the SSD, the columnar PAX layout is more efficient than a row-based layout.

Table 3: Results for the SAS HDD: End-to-end query execution time, entire system energy consumption, and I/O subsystem energy consumption for a selection with aggregation query on the Synthetic64 table at various selectivity factors.

	0.1%	*10%*	*100%*
Elapse time (seconds)	1,485	1,486	1,488
Entire System Energy (kJ)	354	353	355
I/O Subsystem Energy (kJ)	13	13	13

In addition, from Figure 5 (b) and Table 2, we observe that the Smart SSD provides a big energy efficiency benefits – up to **18.8X** and **3.0X** over the HDD and the SSD respectively, with 0.1% selectivity. Furthermore, from Figure 5 (c) and Table 2, we observe that the Smart SSD achieves a substantial I/O subsystem energy efficiency improvement. For example it reduces the energy consumption by **24.9X** and **2.0X** over the HDD and the SSD cases respectively, at 0.1% selectivity. The interesting observation for the I/O subsystem energy consumption is that the Smart SSD energy efficiency benefit over the SSD is not proportional to the elapsed time. In other words, the elapsed times at 0.1% selectivity when using the Smart SSD with a PAX layout and the regular SSD are about 78 seconds and 207 seconds, which shows 2.6X performance improvement. However, the I/O subsystem energy efficiency improvement is only 2.0X. That is because the Smart SSD consumes additional computation power compared to the regular SSD.

With the Synthetic4 and the Synthetic16 tables, the Smart SSD is usually slower than the regular SSD for the select query at 0.1% selectivity, and in the worst case about 2.6X slower with the NSM format. As above, the PAX format works better with the Smart SSD, and in the worst case the Smart SSD is 22% slower than the regular SSD. The reasons for this behavior are similar to the case of the Synthetic4 table shown in Figure 4 (See Section 4.2.1).

4.2.2 Selection with Aggregation Query

For this experiment, we used the following SQL aggregate query:

```
SELECT AVG (SecondColumn)
FROM SyntheticTable
WHERE FirstColumn < [VALUE]
```

The results for this experiment for the Synthetic64 dataset are shown in Figure 6. The HDD results for this experiment are shown in Table 3. From Figure 6 and Table 3, we note that compared to the Smart SSD case with PAX, the HDD case takes 19.2X longer to execute the query, consumes 18.7X more energy at the whole server/system level, and about 23.3X more energy in just the I/O subsystem.

[1] The load multiple instruction (LDM) allows loading data into any subset of the 16 general-purpose processor registers from memory, using a single instruction.

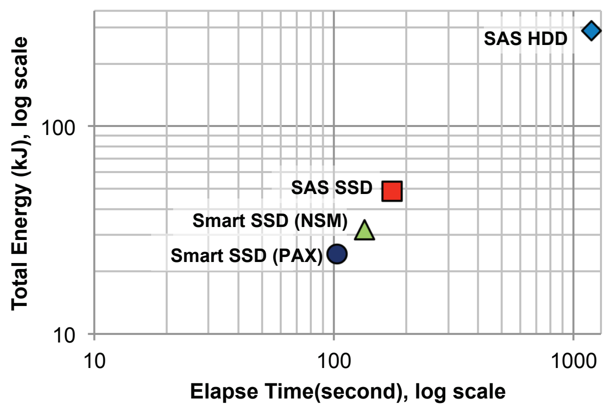

Figure 7: Elapsed time and entire system energy consumption for the TPC-H query 6 on the LINEITEM table (100SF).

Similar to the previous results, the Smart SSD shows significant performance and energy savings over the HDD and the SSD cases. As seen in Figure 6 (a), the Smart SSD improves performance for the highly selective queries by up to **2.7X** over the (regular) SSD case when 0.1% of the tuples satisfy the selection predicate. In addition, as shown in Figure 6 (b), using the Smart SSD (with PAX) is **2.9X** more energy efficient than the regular SSD case when the selectivity is 0.1%. Furthermore, as can be observed from Figure 6 (c), the Smart SSD with PAX is **1.9X** more efficient in the I/O subsystem over the (regular) SSD case, at 0.1% selectivity.

The one big difference between the simple selection query results shown in Figure 5 and Table 2, and the aggregate query results shown in Figure 6 and Table 3, is that with the aggregate query, the Smart SSD has better performance than the HDD and the SSD cases even at 100% selectivity. The reason for this behavior is that the output of the aggregation query is far smaller than the output of the selection query. Thus, the selection query has a much higher I/O cost associated with transferring data from the Smart SSD to the host, which diminishes the benefits of the Smart SSD.

With the Synthetic4 and Synthetic16 tables, similar to the selection query results, the Smart SSD is usually slower than the regular SSD for the aggregate query at 0.1% selectivity, and in the worst case about 2.5X slower with the NSM format. As above, performance is higher with the PAX format in the Smart SSD, and in the worst case the Smart SSD is 20% slower than the regular SSD. The reasons for this are also similar to the case of the Synthetic4 table shown in Figure 4 (See Section 4.2.1).

4.2.3 TPC-H Query 6

For this experiment, we used the LINEITEM table and Query 6 from the TPC-H benchmark [30], using the default SHIPDATE, DISCOUNT, and QUANTITY values for the predicates in the query. This query is:

```
SELECT SUM (EXTENDEDPRICE*DISCOUNT)
FROM LINEITEM
WHERE SHIPDATE >= 1994-01-01 AND
          SHIPDATE < 1995-01-01 AND
          DISCOUNT > 0.05 AND
          DISCOUNT < 0.07 AND
          QUANTITY < 24
```

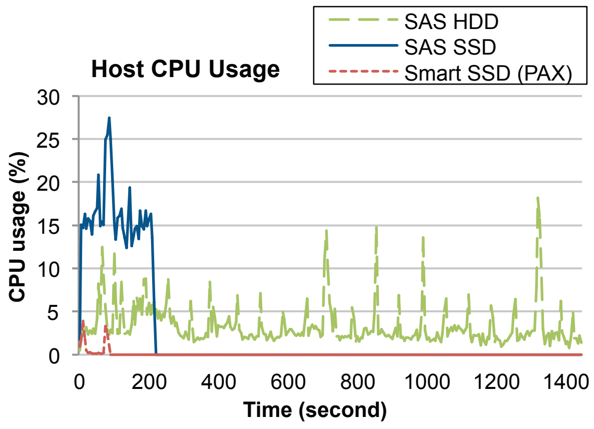

Figure 8: Host CPU usage for the SAS HDD, the SAS SSD, and the Smart SSD for a selection query with average query on Synthetic64 table at 0.1% selectivity factor.

Figure 7 shows the results with the HDD, the SSD, and the Smart SSD (with the NSM and the PAX layouts). The Smart SSD with the PAX layout improves overall query response time by **11.5X** and **1.7X** over the HDD and the SSD cases respectively. Also, it provides **12.0X** and **2.0X** energy efficiency gains for the entire system over the HDD and the SSD respectively. The LINEITEM table contains 51 tuples in a data page, which is more than the Synthetic64 case (29 tuples/page), but less than the Synthetic16 table case (109 tuples/page). With the Synthetic16 table, the Smart SSD with the PAX layout provides about **12.5X** and **1.8X** performance improvements over the HDD and the SSD respectively, for the aggregate query at 0.1% selectivity factor. The selectivity factor of the TPC-H benchmark Query 6 is 0.6%. As explained in Section 4.2.1, the number of tuples in a data page has a big impact on the performance improvement that is achieved using the Smart SSD. So, the LINEITEM table should have provided better performance improvement than the Synthetic16 table. However, the higher selectivity of the TPC-H benchmark Query 6 (0.6% vs. 0.1%), and its more complex predicates (five predicate vs. one predicate) saturates the CPU and the memory resources in the Smart SSD. As a result, the performance improvement of TPC-H Query6 with LINEITEM table is similar to that of the aggregate query described in Section 4.2.2 with a 0.1% selectivity factor for the Synthetic16 table.

4.3 Discussion

The energy gains are likely to be much bigger with more balanced host machines than our test-bed machine. Recall from the discussion in Section 4.2.2 that with the aggregate query, we observed 18.7X and 2.9X energy gains for the entire system, over the HDD and the SSD, respectively. If we only consider the energy consumption over the base idle energy (235W), then these gains become 25.1X and 11.6X over the HDD and the SSD, respectively. Figure 8 shows the host CPU usage for the HDD, the SSD, and the Smart SSD. The SAS SSD uses about 20% of the host CPU during the query execution time whereas the Smart SSD rarely uses the host CPU. In our experimental setup, the power consumption of the host CPU is about 65W whereas the power consumption of the general ARM core is less than 5W. This power consumption difference results in the Smart SSD's huge energy consumption gain (11.6X) over the SAS SSD at the entire system level.

A crucial observation that we made is that the processing capabilities inside the Smart SSD quickly became a performance bottleneck, in particular when the selection predicate matches many input tuples or when there is a large amount of processing to be done per page of data (e.g., the Synthetic4 table). For example, as seen in Figure 5 (a), when all the tuples match the selection predicate (i.e., the 100% point on the x-axis), compared to the regular SSD the query runs 43% slower on the Smart SSD. In this case, the low-performance embedded processor without L1/L2 caches and the high latency cost for accessing the DRAM memory quickly became bottlenecks. Also, as discussed in Section 4.2.1, the Smart SSD achieves greater benefits when the query requires fewer computations per data page.

The development environment that is required to run code inside the Smart SSD needs further development. A large part of the tool that we used in this study was developed hand-in-hand with Samsung for this project. To maximize the performance that we could achieve with the Smart SSD, we had to carefully plan the layout of the data structures used by the code running inside the Smart SSD to avoid having crucial data structures spill out of the SRAM. Similarly, we used a hardware-debugging tool called Trace32, a JTAG in-circuit (ICD) debugger [31], which is far more primitive than the regular debugging tools (e.g., Visual Studio) available to database systems developers.

On the DBMS side, the implication of using a Smart SSD for query processing has other ripple effects. One key area is around caching in the buffer pool. If there is a copy of the data in the buffer pool that is more current than the data in the SSD, pushing the query processing to the SDD may not be feasible. Similarly, queries with any updates can't be processed in the SSD without appropriate coordination with the DBMS transaction manager. If the database is immutable then some of these problems become easier to handle.

In addition, there are other implications for the internals of existing DBMSs, including query optimization. If all or part of the data is already cached in the buffer pool then pushing the processing to the Smart SSD may not be beneficial (from both the performance and the energy consumption perspectives). In addition, even when processing the query the *usual* way is less efficient than processing all or part of the query inside the Smart SSD, we may still want to process the query in the host machine as that brings data into the buffer pool that can be used for subsequent queries.

Finally, using a Smart SSD can change the way in which we build database servers/appliances. For example, if the issues outlined above are fixed, and Smart SSDs in the near future have both significantly more processing power and are easier to program, then one could build appliances that have far fewer compute and memory resources in the host server than what typical servers/appliances have today. Thus, pushing the bulk of the processing to Smart SSDs could produce a data processing system that has higher performance and potential a lower energy consumption profile than traditional servers/appliances.

At the extreme end of this spectrum, the host machine could simply be the coordinator that stages computation across an array of Smart SSDs, making the system look like a parallel DBMS with the master node being the host server, and the worker nodes in the parallel system being the Smart SSDs. The Smart SSDs could basically run lightweight isolated SQL engines internally that are globally coordinated by the host node. Of course, the challenges associated with using the Smart SSDs (e.g. buffer pool caching and transactions as outlined above) must be addressed before we can approach this end of the design spectrum.

5. RELATED WORK

Since Jim Gray's 2006 prediction [13] that "tape is dead, disk is tape, and flash is disk", various DBMS internal components have been revisited for flash SSDs to improve the DBMS performance (e.g., for query processing [9, 32], index structures [2, 21, 33], and page layout [20]). In particular, a promising and well-established way of using the SSDs in a DBMS is to extend the main-memory buffer pool [5, 7, 10, 11, 19]. With an SSD buffer pool extension, pages that are evicted from the main-memory buffer pool are selectively cached in the SSDs to be served for subsequent accesses on the pages. The industry has released commercial storage appliances including Oracle Exadata [1], Teradata Virtual Storage System [29], and IBM XIV Storage System [14] that use similar ideas. As revealed in [11], however, the SSD buffer pool extensions are mainly beneficial for OLTP workloads, and not data warehousing workload, which is the focus of this paper.

A nice overview of techniques that use flash memory for DBMSs is described in [18].

Over a decade ago, the concept of in-storage processing, which involves combining on-disk computational power with memory to execute all or part of application functions directly in the device, was propose in the Active Disks [27, 28] and the Intelligent Disks [16] projects. The studies proposed to exploit the excess computational power of the embedded processors in disks for useful data processing (offloaded from the host) to mainly reduce the data traffic between the host and the device. For example, *Riedel et al.* demonstrated performance gains for data computational tasks (e.g., filtering, image processing [28], and primitive database operations such as scan, aggregation [27]) in this environment. Since then, however, the computational power of disk controllers has not been improved significantly [6], and therefore none of the approaches have been commercially successful.

Similar efforts of moving computation closer to the data have been realized with the help of special-purpose or commodity hardware to improve the performance of database processing. *Mueller et al*. [24, 25] proposes an FPGA-based approach, in which an FPGA is located between the disk and the host. In this approach, the data from the disk is pre-processed before it is fed to the host processors, and as a result, some of the computational work can be offloaded from the host. A commercial product based on this idea can be found in [12]. Another approach that uses additional commodity processors in storage servers is Oracle Exadata [1]. By pushing down some database operations from database servers to storage servers, the amount of data traffic can be significantly reduced. Our work follows in this same direction, but directly uses processing that can be directly built as part of the SSD manufacturing process.

Recently, several studies have explored the feasibility of in-storage processing on flash SSDs [8, 17]. These studies propose using a dedicated hardware logic (that is placed inside a flash controller) to accelerate the scan operation. A commercial SoC designer was used to demonstrate performance and energy gains by simulating the hardware logic. In [6], an analytical model was presented to examine the energy-performance trade-offs when data analysis tasks are carried out on the SSD-resident processors in a High Performance Computing (HPC) context. Lessons from these studies can be used to guide the future development of additional processing inside the Smart SSD for database related data processing.

6. CONCLUSIONS AND FUTURE WORK

The results in this paper show that Smart SSDs have the potential to play an important role when building high-performance database systems/appliances. Our end-to-end results using SQL Server and a Samsung Smart SSD demonstrated significant performance benefits (> 2.7X in some cases) and a significant reduction in energy consumption for the entire server (> 3.0X reduction in some cases) over a regular SSD. While we acknowledge that these results are preliminary (we only tested a limited class of queries and on only one server configuration), we also feel that there are potential new opportunities for crossing across the traditional hardware and software boundaries with Smart SSDs.

A significant amount of work remains. On the SSD vendor side, the existing tools for development and debugging must be improved if Smart SSDs are to have a bigger impact. We also found that the hardware inside our Smart SSD device is limited, and that the CPU quickly became a bottleneck as the Smart SSD that we used was not designed to run general purpose programs. The next step must be to add in more hardware (CPU, SRAM and DRAM) so that the DBMS code can run more effectively inside the SSD. These enhancements are absolutely crucial to achieve the 10X or more benefit that Smart SSDs have the potential of providing (see Figure 1). The hardware vendors must, however, figure out how much hardware they can add to fit both within their manufacturing budget (Smart SSDs still need to ride the "commodity" wave) and the associated power budget for each device. On the software side, the DBMS vendors need to carefully weigh the pros-and-cons associated with using smart SSDs. Significant software development and testing time will be needed to fully exploit the functionality offered by Smart SSDs. There are many interesting research and development issues that need to be further explored, including extending the query optimizer to push operations to the Smart SSD, designing algorithms for various operators that work inside the Smart SSD, considering the impact of concurrent queries, examining the impact of running operations inside the Smart SSD on buffer pool management, considering the impact of various storage layout, etc. To make these longer-term investments, DBMS vendors will likely need the hardware vendors to remove the existing roadblocks.

Overall, the computing hardware landscape is changing rapidly and Smart SSDs present an interesting additional new axis for thinking about how to build future high-performance database servers/appliances. Our results indicate that there is a significant potential benefit for database hardware and software vendors to come together to explore this opportunity.

7. ACKNOWLEDGEMENTS

We would like to thank Paul Dyke and Mike Zwilling for their insightful comments on previous drafts of this paper. This research was supported in part by a grant from the Microsoft Jim Gray Systems Lab, a gift donation from Samsung, and by the National Science Foundation under grant IIS-0963993.

8. REFERENCES

[1] A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server. *White Paper*, Oracle Corp, 2012.

[2] D. Agrawal, D. Ganesan, R. K. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *PVLDB*, 2009.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, 2001.

[4] ARM Developer Suite. http://infocenter.arm.com/help/topic/com.arm.doc.dui0068b/DUI0060.pdf

[5] B. Bhattacharjee, C. Lang, G. A. Mihaila, K. A. Ross, and M. Banikazemi. Enhancing Recovery Using an SSD Buffer Pool Extension. In *DaMoN*, 2011.

[6] S. Boboila, Y. Kim, S. S. Vazhkudai, P. Desnoyers, and G. M. Shipman. Active Flash: Out-of-core Data Analytics on Flash Storage. In *MSST*, 2012.

[7] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD Bufferpool Extensions for Database Systems. In *VLDB*, 2010.

[8] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. Ganger. Active Disk Meets Flash: A Case for Intelligent SSDs. *CMU Technical Report*, 2011.

[9] J. Do and J. M. Patel. Join Processing for Flash SSDs: Remembering Past Lessons. In *DaMoN*, 2009.

[10] J. Do, D. Zhang, J. M. Patel, and D. J. DeWitt. Fast Peak-to-Peak Behavior with SSD Buffer Pool. In *ICDE*, 2013.

[11] J. Do, D. Zhang, J. M. Patel, and D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS Buffer Pool Using SSDs. In *SIGMOD*, 2011.

[12] P. Francisco. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. *IBM Redbook*, 2011.

[13] J. Gray. Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King, 2006. http://research.microsoft.com/en-us/um/people/gray/talks/ Flash_is_Good.ppt

[14] IBM XIV Storage System. http://www.ibm.com/systems/storage/disk/xiv/index.html

[15] Iometer. http://www.iometer.org

[16] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A Case for Intelligent Disks (IDISKs). In *SIGMOD Record*, vol. 27, 1998.

[17] S. Kim, H. Oh, C. Park, S. Cho, and S-W. Lee. Fast, Energy Efficient Scan inside Flash Memory SSDs. In *ADMS*, 2011

[18] I. Koltsidas and S. Viglas. Data Management over Flash Memory. *SIGMOD Tutorial*, 2011.

[19] I. Koltsidas and S. Viglas. Designing a Flash-Aware Two-Level Cache. In *ADBIS*, 2011.

[20] S.-W. Lee and B. Moon. Design of Flash-Based DBMS: An In-Page Logging Approach. In *SIGMOD*, 2007.

[21] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree Indexing on Solid State Drives. *PVLDB*, 2010.

[22] LSI, SAS 9211-4i HBA. http://www.lsi.com/channel/products/storagecomponents/Pages/LSISAS9211-4i.aspx

[23] Microsoft SQL Server 2012. http://www.microsoft.com/sqlserver

[24] R. Mueller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *PVLDB*, 2009.

[25] R. Mueller and J. Teubner. FPGA: What's in it for a Database? In *SIGMOD*, 2009.

[26] Power and Temperature Measurement Setup Guide. http://spec.org/power/docs/SPEC-Power_Measurement_Setup_Guide.pdf

[27] E. Riedel, C. Faloutsos, and D. F. Nagle. Active Disk Architecture for Databases. *CMU Technical Report*, 2000.

[28] E. Riedel, G. A. Gibson, and C. Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *VLDB*, 1998.

[29] Teradata. Virtual Storage. http://www.teradata.com/t/brochures/Teradata-Virtual-Storage-eb5944

[30] TPC Benchmark H (TPC-H). http://www.tpc.org/tpch

[31] Trace32, Lauterbach Development Tools. http://www.lauterbach.com

[32] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query Processing Techniques for Solid State Drives. In *SIGMOD*, 2009.

[33] C.-H. Wu, T.-W. Kuo, and L.-P. Chang. An Efficient B-tree Layer Implementation for Flash-Memory Storage Systems. *ACM TECS*, 2007.