The Promise of Solid State Disks

Increasing efficiency and reducing cost of DBMS processing

Karsten Schmidt
kschmidt@cs.uni-kl.de

Yi Ou
ou@cs.uni-kl.de

Theo Härder
haerder@cs.uni-kl.de

Databases and Information Systems Group
Department of Computer Science
University of Kaiserslautern
D-67653 Kaiserslautern, Germany

ABSTRACT

Most database systems (DBMSs) today are operating on servers equipped with magnetic disks. In our contribution, we want to motivate the use of two emerging and striking technologies, namely native XML DBMSs (XDBMSs for short) and solid state disks. In this context, the traditional read/write model optimized by a sophisticated cache hierarchy and the IO cost model for databases needs adjustment. Such new devices together with optimized storage mapping of XML documents provide a number of challenging characteristics. We discuss howtheir optimization potential ca be exploited to enhance transactional DB performance under reduced energy consumption to match increasing application demands and, at the same time, to guarantee economic energy use in data centers.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*mass storage*; H.2.4 [**Database Management**]: Systems; H.3.2 [**Information Storage and Retrieval**]: Information Storage

General Terms

Performance, Design, Economics

Keywords

SSD, DBMS Storage, XML Mapping, Energy Efficiency, TCO

1. INTRODUCTION

"Green Computing" was coined just a year ago and is now omnipresent in the IT world. Even for DBMSs, energy and resource consumption is a rapidly emerging problem to be solved, before environmental-friendly data centers can be established. New and improved hardware technologies addressing the energy consumption challenge arise everywhere on the market. Besides processor developments, chip miniaturizations, and cooling improvements, disks have experienced an increased attention, too. But, information system design is still focusing on the fairly static characteristics of magnetic disks when optimizing or employing a DBMS. The fast development of new storage devices such as solid state disks (SSD) or NAND-based[1] flash disks leads to new challenging research opportunities in the DBMS world [11].

SSD devices compared to magnetic disks (disks for short) have a favorable IO model, which results from their underlying physics and primarily differs in power consumption and space capacity. The advantages of this IO model should also be addressed at the DB level to enhance operational goals such as transactional throughput, power consumption, reliability, or scale factors (i.e., number of users, amount of data) [6].

Web systems, data logging or tracking, data warehousing, and data exchange are very popular uses and often confronted with vast amounts of new data. Because of its evolution capability, XML is the emerging standard data format for such tasks in a variety of "dynamic" application domains. Most of these kinds of data stores are dealing with large and fast growing volumes [9] of data as well as highly selective access (queried by SQL/XML, XPath, or XQuery) or reporting functions which aggregate large amounts of data. Due to XML's flexibility, native storage techniques for an XDBMS became a hot research area in recent years. Beginning with shredding solutions (i.e., mapping XML data to relational structures), more XML-aware approaches such as hybrid, tree-based, and native storage mappings successfully emerged [12, 3]. Especially native XDBMSs [8] tackle the performance issue when processing XML data and new developments should also care about the demands for energy savings and alternative hardware usage to reduce the TCO (Total Cost of Ownership).

Virtualization, often used in large data centers, can reduce hardware costs, but also minimize operational costs (e.g., energy supply or administration overhead). For DBMSs, such a virtualization (e.g., of the storage) is mostly unwanted, because it cannot guarantee predictable response times which are crucial for query optimization and the validity of the underlying cost model. However, DB algorithms can be adjusted to a variety of resources to control the load of IO, CPU, and memory, which, in turn, may provide a strongly improved energy balance. As long as energy costs can amortize potentially higher purchase costs for modern and "green" hardware, their usage is advisable.

Evaluating alternative hardware in terms of performance and costs in a native XDBMS environment requires a suitable benchmark modeling the afore mentioned use cases. The most promising benchmark proposal is TPoX [10] which meets most of our desires for manipulating XML databases.

C3S2E-09 2009, May 19-21, Montreal [QC, CANADA]

[1] Availability and prices disqualify NOR-based devices.

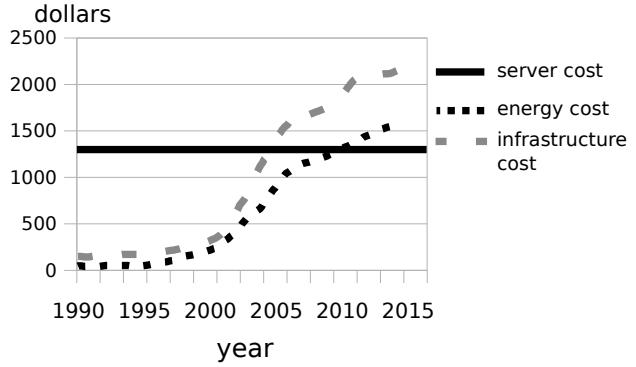


Figure 1: Amortized sever cost [2].

2. TCO OF STORAGE SYSTEMS

Nowadays data processing systems are confronted with two requirements, (1) to provide enough capacity to store the steadily increasing amounts of data, and (2) to support concurrent read and write access by huge bandwidths. For the TCO (total cost of ownership), hardware prices and operational costs are as important as the price/performance and price/capacity ratios.

The TCO of a storage system can be considered independent of applications using this storage by taking the average read and average write loads into account. Thus, the TCO for a time period t can be calculated by $TCO(t) = cost_{initial} + cost_{runtime} * t_{runtime}$. The most complex function is $cost_{runtime}$ because runtime costs include not only the energy consumption but also cooling costs, for instance; however we omit these additional costs. Furthermore, the initial costs $cost_{initial}$ of a device setup (e.g., number of devices, RAID), are driven by data volume, availability, and throughput requirements specified by the applications.

The decision, which storage configuration is advisable based on their TCO, needs to compare different cost estimations for them. However, such a comparision is heavily dependent on the expected runtime. For instance, the higher $cost_{initial}$ of system A may be amortized by lower $cost_{runtime}$ compared to system B after a certain runtime. Such a simple TCO-based comparison only holds if both systems offer similar performance characteristics. Otherwise, the TCO needs to be normalized for read and write access to a comparable unit, such as IO per second (IOPS).

In recent reports [4, 2], comparison of hardware costs and operational costs in data centers has revealed that the rising energy prices very quickly lead to a situation where the operational costs are dominating (see Figure 1). Our observations and the work by [5] have shown that energy costs for the storage equipment are usually between 8 % and 25 % of a database server system. The use of disk arrays (i.e., RAID) significantly increases this share [13]. Thus, the trend in Figure 1 can be scaled down to the storage part showing similar characteristics.

Even more important for a database system is the IOPS measurement to adequately compare storage systems. By combining the TCO and IOPS figures, a comparison of totally different storage configurations may be possible, too.

3. SSD VS DISK

To allow for a comparison of SSDs and disks, a basic understanding of their internals is necessary. Usual magnetic disks have several rotating platters divided into areas called sectors. Each of these sectors covers 512 B of data—the basic unit of read or write—and is mechanically accessed by one of the read/write heads. The

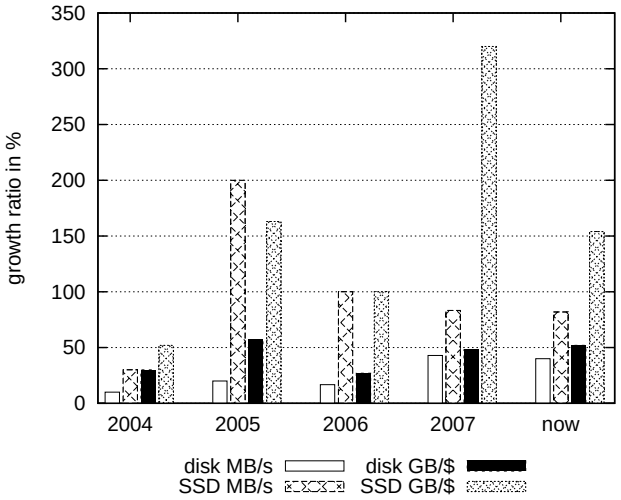


Figure 2: Growth of critical device parameters (source: APC Magazine).

major drawback is the *latency* time that is required to (re-)position the head's arm. Therefore, the ("mechanical") time to seek for the right *track* and the right *sector* is a major time fraction of random IO as opposed to sequential IO which is considerably faster because positioning happens only once.

In contrast, SSD is an electronic storage where no mechanical movement is required at all. Although a tiny overhead is needed to address or locate the requested block, its impact is much less compared to the disk's latency. The basic unit of read is a single segment which is typically 2 KB and can be up to a physical block of $\leq$256 KB. Because of the *NAND* logic, an entire block needs to be written instead of a subset of segments contained and often implies an expensive erase operation of the block in advance. And if data is already stored in the target block, it needs to be merged with the new data, which is achieved by merging the segments and moving the entire block to a new and empty block (prepared by an erasure operation). A built-in mechanism, the so-called wear leveling, ensures durability of the device [1]. Hence, SSDs offer a zero-latency design, but embody a more expensive write model, which leads to a different IO model that has to be addressed by algorithms and cost parameters.

Figure 2 exhibits the potential of SSD development to be expected in the next years when comparing various annual growth ratios. The displayed ratios show that SSD's price decline and throughput increase have a faster pace than those of disks. In the near future, SSDs will presumably balance out the price and capacity penalty by their speed advantage, beyond it, they may outperform disks in all respects.

A more detailed insight is given in Figure 3 where we compared two equally fast consumer drives—an SSD and a disk—showing their typically different IO characteristics and illustrating major differences occurring for random and sequential access patterns. Note that random reads (RR) and sequential reads (SR) are depicted in the left chart, whereas random writes (RW) and sequential writes (SW) are depicted in the right chart, respectively. The disk drive achieves nearly the same poor throughput ratios for read and write operations, whereas sequential access in both cases shows excellent performance. In contrast, the SSD's sequential access characteristics are comparable, but random writes are clearly problematic compared to random reads, which scale linearly with the block size.

Another meaningful performance measurement is IOPS. In Ta-

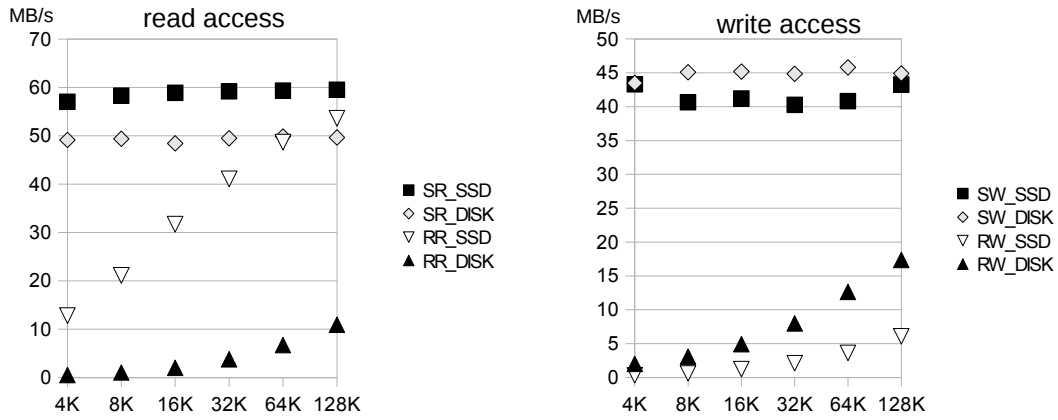


Figure 3: Read-write comparison for SSDs and disks.

ble 1, the specifications for consumer-level and high-end drives are outlined. Due to the disbalance between read/write and sequential/random access, the table shows combinations and their typical IOPS ratios. The SSD's random read superiority is clearly confirmed, whereas the random write penalty, at least for the available consumer devices, are eye-catching again. Comparing sustained IO ratios, difference between read and write is marginal; however, the trend for SSDs shows a significant increase for both values. Thus, having sequential access, both kind of drives are fairly comparable, but having random access they identify a huge disbalance.

4. CACHE HIERARCHY

In order to evaluate throughput ratios at the DBMS level, we need to investigate the entire cache hierarchy that influences the data flow from the device level to the application level. But where should the SSD be integrated regarding its characteristics? Either as an intermediate caching layer between disks and RAM or as a substitute for disks. Our focus is to show the SSD's adequacy to replace a disk, thus Figure 4 depicts the layered overall architecture for data caches and makes the distinction between OS- and device-controlled caches when necessary. Disks benefit from device caches when data is sequentially read or sequentially/randomly written, whereas raw-device IO of an SSD is neither limited nor boosted, because it is not equipped with (and would not benefit from) a device cache. At the interface of the OS kernel to the storage devices, a *kernel cache* is allocated that allows for buffering entire blocks using an LRU-based replacement policy. Furthermore, this cache is automatically resized depending on available main memory. Cache filling is optimized for sequential accesses by synchronous and asynchronous prefetching. In the layer above, a virtual file system (VFS) maintains several caches for *inodes*, *block mappings*, and *directories*, as well as a *page cache*. As these caches are completely controlled by the kernel, a flush to empty them is the only functionality provided for the application. Typically, page cache and kernel cache hold actual, but not explicitly and currently requested data in RAM and often they keep the same data twice. In contrast, the *inode cache* and *directory cache* speed-up file resolution, block addressing, and navigation.

At the OS level, room for file caching may be reserved in such a way that all RAM currently not needed by the OS and the applications is used for caching block reads and writes of the applications. Moreover, the file system may use this cache to prefetch blocks in a way not visible and controllable by the applications. This kind of prefetching is often heuristically performed depending on the locality or sequentiality of the current block accesses to disk. For example, Linux offers the following options:

- The first file access is conservative when deciding on prefetching. Only when the *first block* of a file is read, sequential access is anticipated and some minimal prefetching takes place.
- Synchronous prefetching by the file system can be enabled to lower seek costs for larger reads.
- Read access pattern recognition may dynamically trigger synchronous and/or asynchronous prefetching.

To enable greater flexibility, these kinds of disk access optimization are further separated into raw-block caching and file-structure caching (metadata such as inodes, indirect block references, etc.). Moreover, some access options (direct, sync) are provided at the file system interface. However, not all OS versions (in our case Linux) observe such application desires, e.g., even if the raw-disk option is enforced, the OS does not respond to it. As a result, great uncertainty may exist for all kinds of disk IO timings in real application environments.

[2]announced, not available yet

Table 1: IOPS, MB/s, and energy consumption figures.

Drive	IOPS random		MB/s sequential		Watt
	read	write	read	write	
consumer SSD	3,000	50	70	50	0.06 – 2.0
high-end SSD[2]	35,000	5,000	250	170	0.9 – 3.5
consumer disk	125	100	100	100	6.0 – 12.5
high-end disk	≤ 200	≤ 200	160	150	10.0 – 18

5. SSD USE IN DB APPLICATIONS

Important SSD characteristics are still visible at the application level: the poor random-write and excellent random-read performance. Moreover, energy consumption is differing from that of a disk-based system. For these reasons, we identified the following optimization options:

- Design compact and fine-grained storage structures to minimize IO for allocating, fetching, and rewriting DB objects.
- Use processing concepts which maximize memory-resident operations and minimize random reads/writes to storage—in particular, strictly avoid random writes to SSD.

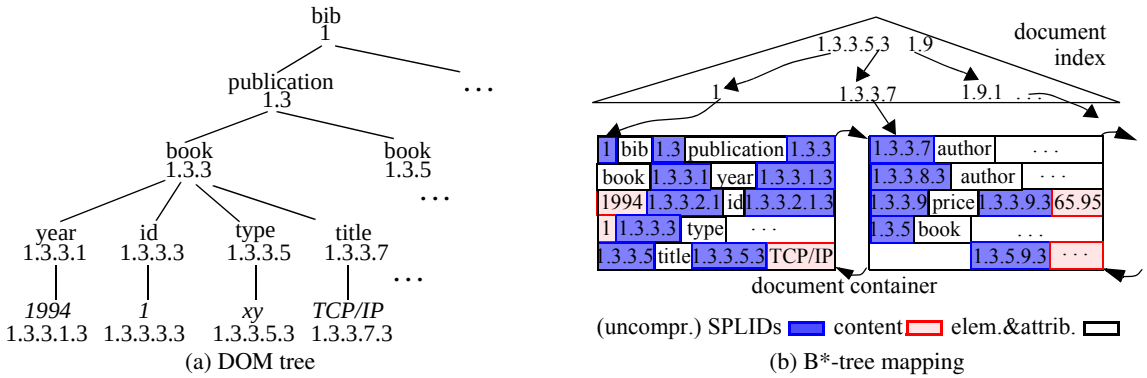


Figure 5: Completely stored XML document.

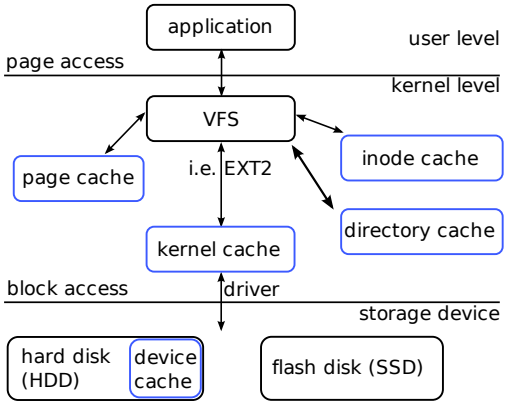


Figure 4: Caching hierarchy.

We discuss these principles and their potential energy savings for native XML DBMSs. In the following, we outline differing storage mappings for XML documents, which may help to reveal critical aspects of DBMS-related IO performance. In general, saving IO is the major key to performance improvements and, in turn, energy efficiency in DBMSs. Therefore, both goals imply the use of compact storage formats for DB objects. This is particularly true for storing, modifying, or querying XML documents, because they may contain substantial redundancy in the structure part, i.e., the inner nodes of the document tree (see Figure 5a). Because write propagation of modified DB objects causes a large share of DB IO and dependent log IO, optimization of storage mapping reduces log IO at the same time. Hence, XML documents have to be encoded into suitable physical representations, flexible enough for dynamic modifications, and kept as trees on storage to later enable fine-grained document processing.

5.1 Mapping Concepts

Natively storing XML documents requires identification of the resulting tree nodes, e.g., the assignment of node labels, the type of which is particularly important for processing flexibility and effectivity.

Range-based labeling schemes are quite expressive. When two labels are compared, all axis relationships can be directly decided. However, they fail to provide the ancestor IDs when the ID of the context node is known. Moreover, dynamic insertion of subtrees would cause a relabeling of the document (or a fraction of it) triggering a bulk IO operation. In contrast, *prefix-based labeling*, derived from the concept of *DeweyIDs*, remains stable also in dynamic documents and supports all node operations and axis relationships mentioned without requiring document access—only by checking a given label with that of the context node [7].

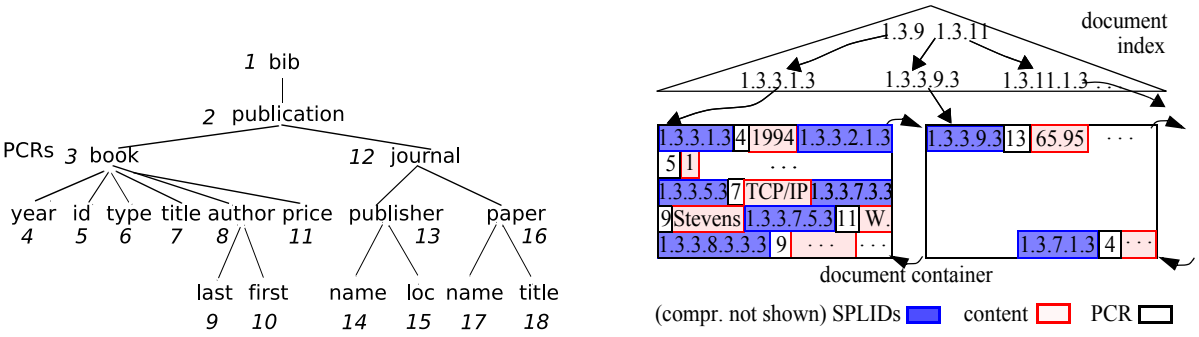
B*-trees—made up by the document index and the document container—and DeweyIDs are the most valuable features of physical XML representation. B*-trees enable logarithmic access time under arbitrary scalability and their split mechanism takes care of storage management and dynamic reorganization. As illustrated in Figure 5b, we provide an implementation based on B*-trees which cares about structural balancing and which maintains the XML nodes stored in variable-length format (DeweyID+element/attribute (dark&white boxes) or DeweyID+value (dark&grey boxes)) in document order.

Structure virtualization aims at getting rid of the structure part in a lossless way and helps to drastically save storage space and, in turn, document IO. As a consequence, log space and log IO may be greatly reduced, too. This virtualization is enabled by the combined use of DeweyIDs as node labels and a small memory-resident data structure, called path synopsis (illustrated in Figure 6a for a cut-out of the well-known *dblp* document [9]) representing only the path classes. As a consequence, storage footprint is saved for the entire structure part and the document or selected paths of it are only reconstructed on demand. For this reason, PCRs (path class references) are added to the nodes of the path synopsis to identify path classes. When, e.g., DeweyID=1.3.3.7.3 together with PCR=7 is delivered as a reference (e.g., from an index) for value *TCP/IP*, the entire path instance together with the individual labels of the ancestor nodes can be rebuilt: *bib/publication/book/title*.

5.2 Storage Mappings

DeweyIDs offer two options for the storage mapping of XML documents. The so-called full-storage (*fs*) mode is sketched in Figure 5b. It encodes the dot-separated parts of a DeweyID by space-efficient Huffman codes (not shown) and adds it to each tree node. Although each node is stored as a variable-length B*-tree entry, this mapping results in maximum storage footprint and IO for an XML document.

Due to the document order, the DeweyID labeling lends itself to effective prefix compression reducing the DeweyIDs' avg. size to $\sim$20–30 % [7]. Hence, by applying this optimization step separately to all DeweyIDs in a container page (shown in Figure 6b), we obtain the prefix-compressed (*pc*) storage mapping, which saves space and IO, but implies more computational effort, because the reconstruction of the DeweyIDs is needed (starting with the first DeweyID in a container page).



(a) Path synopsis

(b) B*-tree and container layout

Figure 6: Elementless storage structures for XML documents.

Because of the typically huge repetition of element and attribute names in an XML document, a specific storage mapping is very space-effective when the structure part is virtualized and only its content is physically stored, i.e., the values in the leaves together with their DeweyIDs. Each root-to-leaf path in an XML document in Figure 5a would be reduced to a single leaf value with node label and PCR (for path reconstruction). For example, the left-most path of Figure 5a would be stored as (1.3.3.1.3,*4*,1994). Because all elements and attributes disappear in the storage representation, we call this mode *elementless* (*el*).

All three storage mappings[3] of XML documents—*fs*, *pc*, and *el*—were compared in empirical performance measurements. We applied the TPoX tools [10] to generate the various documents in the external text format, called the *plain* format, from which the differing storage mappings were created. To assess their storage effectivity, we cannot refer to absolute numbers, because the sizes of our sample documents vary from 10 MB to 1 GB. Hence, we have normalized the space figures w.r.t. to *plain* (100 %). Hence, the storage footprints of *fs*, *pc*, and *el* obtained quite some reduction and reached ∼95 %, ∼70 %, and ∼65 %, respectively; these figures directly characterize the IO saving when storing or reconstructing the entire document.

The availability of these storage mappings offers the required flexibility to evaluate in detail various storage configurations (device type/storage mapping) and workload scenarios aiming at the analysis of energy efficiency and throughput.

6. EXPERIMENTAL RESULTS

All findings are gained by using the TPoX benchmark suite [10] and our native XDBMS where we stored the TPoX documents in so-called document collections. The DB page size and the XDBMS buffer size were 4 KB respectively 16 MB in all experiments.

To define typical workloads, we assembled two sets of TPoX workloads containing the following queries: *report an account summary*, *return an order*, *resolve a customer*, *search for a specific security*, *return a security price value*, *place an order*, *update a price value*, and *create a new account*. The first five queries are used in a read-only workload. The remaining four queries include updates or inserts and, hence, we assembled a mixed workload with them and the read-only queries. To reflect the indeterminism of real environments, all queries are supplied with random parameters. By changing the weights of the queries, the write load in the benchmarks can be scaled from 0 % to ∼50 %[4]. After locating its (random) target (identified by a DeweyID) in a document via an element index—the only additional index used for these benchmarks—each query traverses the document index and locally navigates in one or two container pages (see Figure 5b and 6b) thereby accessing a number of records. Hence, the set of queries—each executed as a transaction—causes fine-grained and randomly distributed IOs on the XML database. The element index is necessary to avoid plain but costly document scans, which are the fallback access solution but not the preferred access path because their scaling properties are really poor. Furthermore, the randomly selected document nodes reflect selective and, thereby, realistic access behavior.

All our benchmarks and experiments were performed on an AMD Athlon X2 4450e (2.3 GHz, 1MB L2 cache) processor using 1 GB of main memory (RAM) and a separate disk for the operating system. As hard disk, we use a WD800AAJS (Western Digital) having a capacity of 80 GB and an 8MB cache, NCQ, and 7,200 rpm. The SSD device is a DuraDrive AT series (SuperTalent) having a capacity of 32 GB. The operating system is a minimal Ubuntu 8.04 installation using kernel version 2.6.24 and Java 1.6.0_06 for our native XDBMS.

6.1 Benchmark Measurements

The results—measured in *tps* (transactions per second)—of the first set of experiments shown in Figure 7 and Figure 8 compare the throughput rates for the different storage configurations operating on SSD and disk. The read-only scenario (Figure 7) exhibits that disks have a much higher impact on growing database sizes than SSDs, because the tps rates drastically decrease from ∼100 tps for small databases down to ∼40 tps for midsize databases and ∼30 tps for large databases, respectively. Important reasons are the longer search paths, because the heights of the document indexes grow from 2 for 10 MB to 3 or 4 for 1000 MB, and longer seeks, because the storage footprints of the documents cover larger disk areas. On the other hand, the tps results are influenced by caching effects, as discussed in Section 4, and even dominated, especially for small databases. In contrast, the SSD tps rates stay at a high level (∼90 tps), although they are not supported by a device cache. Obviously, the "zero" seek time is beneficial and even the longer search paths in larger databases do not hamper read performance. Note, the *el* storage mapping on SSD produces the highest tps rates regardless of the database size, whereas the decompression overhead of the *pc* storage mapping is not always compensated by the storage savings and, in turn, reduces IOs compared to the *fs* mapping.

[3]Here, we do not consider content compression. It would further decrease IO overhead, but increase CPU time for additional compression/decompression tasks.

[4]Having more writes than reads is conceivable, however, most DB applications are reader-dominated.

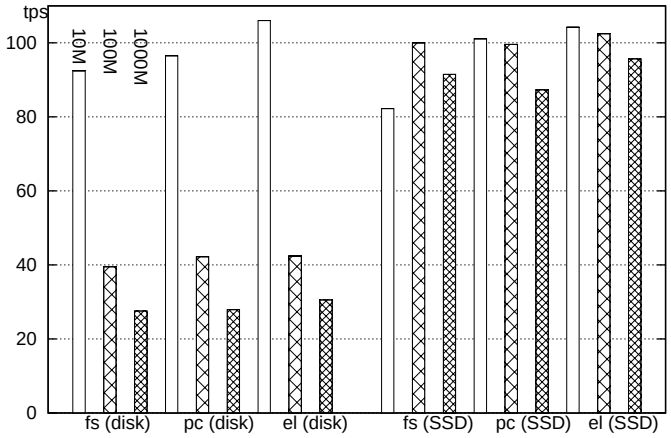


Figure 7: Read-only TPoX benchmark results.

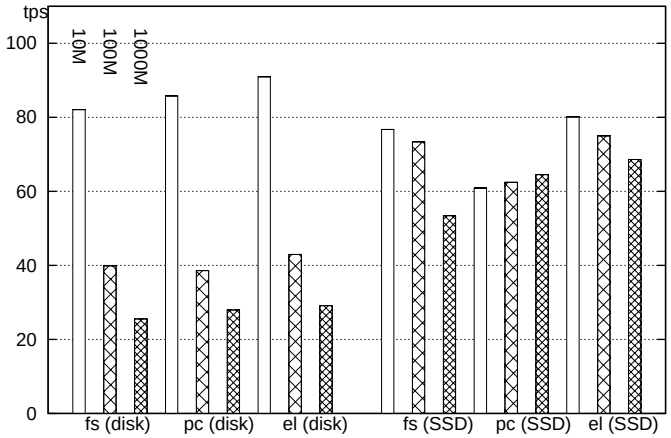


Figure 8: Mixed TPoX benchmark results.

The mixed workload scenario in Figure 8 reveals that tps rates are more sensitive to the share of writes when SSD storage is used. That means, the huge read advantage of SSD is continuously decreasing with an increasing frequency of writes. Because reads and writes are equally fast on disks, the tps characteristics are comparable to those of the read-only benchmarks. In contrast, the tps rates of all SSD scenarios are sensibly decreased because of the higher write penalty.

The overall throughput decrease for growing database sizes has several reasons. In our Linux environment, the file system overhead (e.g., indirect inode references) causes additional IO, which also degrades the cache hit ratios. In addition, disk access is hampered by larger seek times (i.e., arm movements on the platter have to cross larger distances) which further worsens IO times. On the other hand, hidden prefetching caused by the cache hierarchy, as outlined in Section 4, tries to improve IO performance. As a consequence, the overall IO behavior is often unpredictable.

The second set of results in Figure 9 depicts the resource usage during the execution of the mixed workload. It shows that the share of write operations has significant impact on processing time and, as a consequence, that the energy consumption using an SSD device is substantially influenced by the unbalanced read/write model. For each configuration, the processing time can be divided into pure

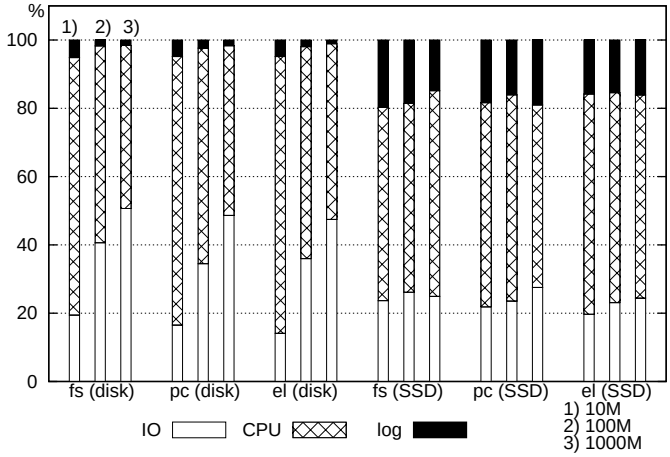


Figure 9: Analysis of elapsed time fractions.

IO time (i.e., for the data), CPU time, and log IO (i.e., sequential writes of log entries). Note, a direct comparison of absolute values across two orders of magnitude is not meaningful. However, the relative comparison of the shares for IO, log writes, and CPU usage is very expressive. One finding is that the share of reader-dominated IO costs is nearly independent of the database size when using an SSD, whereas disk usage implies IO impact steadily growing with the database size. Another interesting fact concerns log IO which exclusively consists of write operations; here SSD cases are penalized again. Nevertheless, the CPU costs are dominating in all scenarios, which diminishes the differences caused by the device type and reveals that XML processing is complex and expensive.

6.2 Energy Consumption and TCO

Guided by the performance superiority of SSDs, we want to assess the prospects of using energy-efficient devices for database systems. For simplicity, we use reasonable default values for energy consumption when necessary (see Table 1). By analyzing workload runtimes and time distributions, a device-oriented energy balance can be calculated. Note that time spend for CPU processing (see Figure 9) overlaps with asynchronous IO and concurrent processing on multiple CPU cores.

The total amount of Watts (cost) consumed by a *device* for a given workload *wl* is calculated as follows:

$$cost_{device}(wl) = (cost_{load}(device) * (time_{IO} + time_{log}) + cost_{idle}(device) * time_{idle}) / time_{wl}$$

whereby

$$time_{idle} = time_{wl} - (time_{log} + time_{IO})$$

to avoid inconsistencies due to CPU time overlappings. The costs for *load* and *idle* are the max and min value in column *Watt* derived from Table 1. That leads to the following device-related Watt (cost) consumptions:

Disk: During read-only workloads, the disk consumes between 6.85 and 9.66 Watt depending on the storage mapping and database size (see Figure 10). These figures increase only a little bit for the mixed workload up to the range between 7.5 and 9.7 Watt. The power consumption slightly varies for the different storage mappings with a uniform database size by 5 %. On the other hand, keeping the same storage mapping, the variation amounts to 28 % between the smallest and largest database size.

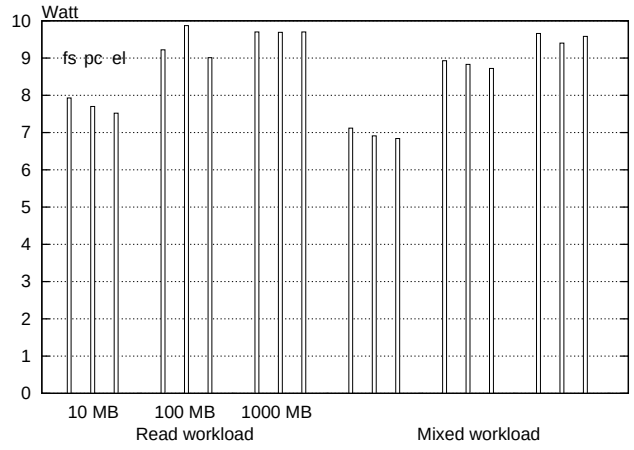


Figure 10: Disk benchmark power consumption.

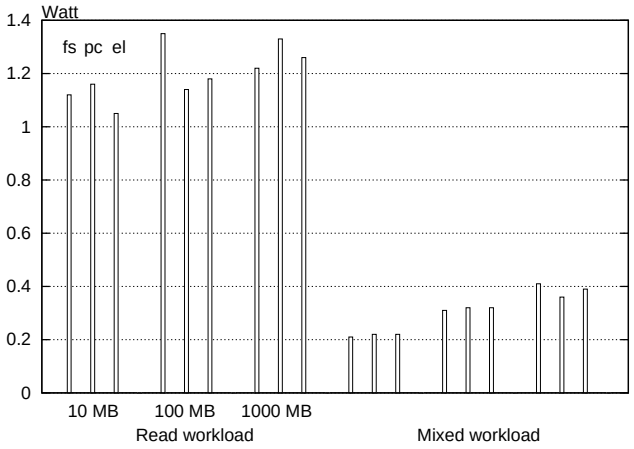


Figure 11: SSD benchmark power cnosumption.

SSD: For the read-only and mixed workloads, the SSD consumes between 0.21 and 0.41 Watt and between 1.05 and 1.35 Watt, respectively (see Figure 11). Here, the variation of power consumption among different storage mappings ranges from 2 to 12 %. However, the difference in power consumption amounts up to 50 % for the database size range considered.

Eventually, the overall power consumption is significantly higher for the disk-based benchmarks, however, the deviation for different configurations is higher for the SSD-based figures which leads to more difficult TCO estimations. Because the impact of write costs is noticeable using SSDs, the space savings due to structure compression (*el*) are significant. However for some cases, the compression overhead penalizes SSD usage because of the SSD's tiny IO footprint, where space savings have lower impact compared to the SSD's idle time during CPU processing.

Extrapolating these findings for the announced high-end SSD disk can be done by comparing IOPS/Watt measurements. Affordable and available SSD disks provide ≤ 2000 read operations per Watt and ≤ 100 write operations per Watt. These numbers increase by ~ 5 times for reads and ~ 15 times for writes using high-end devices. Thus, the TCO may earlier reach the "break-even" point depending on the device's load and runtime. However, taking energy prices into account is a volatile calculation and not considered in this study.

7. CONCLUSIONS

This work revealed that SSD usage in XDBMSs may make magnetic disks become obsolete. However, as long as the steadily falling GB/$ prices dominate TCO, SSD usage is reserved for high IOPS demands where IOPS/$ or MB/$ are of minor importance. Adjusted storage structures and consideration of cache hierarchy influence help to fine-tune storage parameters which is necessary to fully exploit the physical characteristics of SSDs. However, other approaches to reduce power consumption are switching on-demand large magnetic disks on and off—solely to store vast amounts of rarely used data [13]. Furthermore, a hybrid approach combining SSD and disk to distribute data structures such as indexes or by "aging", by data size, or access probability seems to be possible.

In the future, multiuser environments have to be explored. In this context, the on-going SSD development or new non-volatile storage concepts may play an important role.

8. REFERENCES

[1] A. Ban. Wear leveling of static areas in flash memory. US patent, (6732221); assigned to M-Systems, 2004.

[2] C. L. Belady. In the data center, power and cooling costs more than the IT equipment it supports. electronics cooling. vol. 13, no. 1; http://electronics-cooling.com/articles/2007/feb/a3/, 2007.

[3] S. Chaudhuri and K. Shim. Storage and retrieval of xml data using relational databases. In *Proceedings of VLDB '01*, page 730, 2001.

[4] EPA. Epa report on server and data center energy efficiency, http://energystar.gov/index.cfm?c=prod_development.server_efficiency_study, 2007.

[5] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of ISCA '07*, pages 13–23, 2007.

[6] J. Gray and B. Fitzgerald. Flash disk opportunity for server-applications, http://research.microsoft.com/~gray/papers/flashdiskpublic.doc, 2007.

[7] T. Härder, C. Mathis, and K. Schmidt. Comparison of complete and elementless native storage of XML documents. In *Proceedings of IDEAS '07*, pages 102–113, 2007.

[8] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native xml database. *The VLDB Journal*, 11(4):274–291, 2002.

[9] G. Miklau. XML Data Repository, www.cs.washington.edu/research/xmldatasets, 2002.

[10] M. Nicola, I. Kogan, and B. Schiefer. An XML transaction processing benchmark. In *SIGMOD Conference*, pages 937–948, 2007.

[11] M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe. Fast scans and joins using flash drives. In *Proceedings of DaMoN '08*, pages 17–24, 2008.

[12] F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative XML storage strategies. *SIGMOD Rec.*, 31(1):5–10, 2002.

[13] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: helping disk arrays sleep through the winter. In *Proceedings of SOSP '05*, pages 177–190, 2005.