Growing Persistent Trees into the 21st Century

Marc Kramis
supervised by Prof. Dr. Marcel Waldvogel

University of Konstanz
Department of Computer and Information Science
Box V 519, 78457 Konstanz, Germany
marc.kramis@uni-konstanz.de

ABSTRACT

The days of mechanical disks are numbered. Being a handy fellow for sequential access through many years, poor average random access times notoriously cause disks to struggle when it comes to handling large sets of XML data. Ripping out the mechanics and its inherent seek delays is an absolute *must* to allow for efficient and effective operations on fine-grained XML trees. In this paper, we describe TreeTank, a system which takes advantage of the zero-delay seek time of flash-based storage, both addressing the strengths and weaknesses of flash, yet still performing rather well on traditional disks. The switch to flash keenly motivates to shift from the "current state" paradigm towards remembering the evolutionary steps leading to this state. Not only does this simplify many applications, it also offers a huge potential when it comes to accessing web-based resources in a temporal fashion. Being tuned for zero-delay flash-based storage, TreeTank will be able to provide more features faster and with less memory requirements than traditional approaches.

1. INTRODUCTION

Despite its reputation as being bloated, slow, and inefficient, XML established itself as a first-class citizen throughout the modern computer world. As it expands and is adopted for a growing number of document formats, people *do* actually value features such as self-descriptiveness and data-before-schema as well as XML's rich toolset and universal interchangeability including long-term archival. This justifies the immersion of XML as a native data type into many programming languages and databases. However, dealing with large disk-based sets of XML data starting as low as several hundred kilobytes, can unhesitatingly be described as tedious. Opening and saving an OpenDocument file for a tiny modification can easily be in the spell of seconds. A daily download of an XML dump of Wikipedia for performing offline modifications is hardly feasible – not at least because the XML dump itself takes longer than a single day. To put it bluntly, as the English would for XML, this is 'Typical!'.

1.1 Problem Statement

It is currently not possible to efficiently and effectively modify large disk-based sets of XML data. The lack of modification efficiency and effectiveness is deeply rooted in two restrictions imposed by traditional persistent storage. First, the average random access time is so excessive that data needs to be extensively clustered and stored sequentially. This leads to an inefficient mismatch between the fine-grained logical and the coarse-grained physical data model. Second, the capacity is so scarce that applications try to be conservative in their storage needs, making only the most essential of their data persistent – regrettably excluding modification history and past states. This jeopardises the effectiveness of the user's workflow due to an unnaturally skewed focus on the current coarse-grained state instead of the fine-grained modification history. However, the switch to flash-based storage does not only improve the situation, it also brings along its own problems: flash-based storage will eventually wear out if the blocks are overwritten too many times and the block erasure procedure consumes a significant amount of time.

1.2 Contribution

Our contribution is twofold and consists of a background analysis as well as a tangible system named TreeTank.

1. **Analysis**. We uncover the deficiencies of traditional storage and show how flash technology alleviates them. Average random access times are significantly shrinking with the advent of each new storage technology and due to its evolution over time. Simultaneously, capacities are increasing steadily. Consequently, we find a clear tendency from coarse-grained storage units such as flat files or binary large objects towards fine-grained record-, tree-based, or semi-structured storage which does not only store the current state but also the evolutionary steps leading to this state.

2. **TreeTank**. Our system overcomes the traditional limitations by consistently tuning data structures for flash-based storage while still working with magnetic disks and lowering the memory requirements. TreeTank provides a scalable, lightweight, transactional, secure, and persistent framework for efficiently and effectively modifying fine-grained data structures such as XML.

The rest of this paper is organised as follows: Section 2 contains the background analysis. Section 3 describes TreeTank, our main contribution. Section 4 concludes our work and gives an outlook on the remaining work.

2. ANALYSIS

2.1 Hardware Impact

Table 1 lists one state-of-the-art product for each major persistent storage technology in the order of appearance. This includes magnetic tape [24], magnetic disk [22], and flash [8]. Volatile memory [1] is added for the purpose of comparison. The columns have the following meaning: type of storage, capacity, price per capacity[1], sustained sequential read throughput, average random access time, and mixed IO operations with a queue depth of one and a size of 8k.

Type	Cap. [GB]	Price [$/GB]	Seq. Read [MB/s]	Rand. Access [s]	Ops. [IOPS]
Tape	500	0.3	120	6.2E1	1.6E-2
Disk	73	6	96	2.9E-3	1.8E2
Flash	80	30	700	5.0E-5	8.8E4
Memory	2	35	7800	6.4E-8	1.0E6

Table 1: Comparison of persistent storage with volatile memory.

The parameters indicate that each new technology brought persistent storage closer to volatile memory. In stark contrast to strong similarities of capacity, price per capacity, and sequential throughput within one order of magnitude, average random access time and input-output operations per second show a wide discrepancy by two to four orders of magnitude. In addition, each technology itself saw continuous enhancements. E.g., IBM introduced magnetic disks in 1956 with the 350 Disk Storage Unit being a part of the IBM 304 RAMAC (Random Access Memory Accounting). Disk capacity was about 4.8MB and memory capacity about 2.3kB [13]. Hence, the capacity of both disk and memory rose over four orders of magnitude within the better half of the last century. This trend is yet unbroken and close to six orders of magnitude when the focus is not performance but capacity.

Interestingly, the number of input-output operations per second was further improved by flash by truly parallelising the access in analogy to the central processing units that do no longer only ameliorate the performance by making a single core faster but also by adding more cores.

2.2 Degree of Granularity

The most stringent limitation of mechanical disks is their ropy average random access time. Given both a fixed amount of data and time, average random access time determines the number of IO operations per second as well as the size of the moved data. The higher the IO operations per second, the more data objects of smaller size can be shuffled around. In other words, average random access time has an immediate effect on the granularity at which data objects can be handled efficiently. From a conceptual perspective, tapes work best at file-level granularity. Disks can deal with record-level granularity. Flash pushes granularity to the field level. Memory eventually is the candidate of choice when it comes to byte-level data processing. Figure 1 gives a conceptual illustration of the relationship between degree of granularity, average random access time, and object size.

The only way to work at a finer granularity than available with a given storage technology, is to switch to sequential processing or to temporarily store all or a part of the data with a technology that allows a finer granularity. Talking about XML, which is a fine-grained unranked ordered node tree, it immediately becomes clear why there must be a penalty with traditional disk-based storage. XML must be stored sequentially. For random node-level access, it must first be parsed into memory. Once all XML nodes are residing in-memory, they can be randomly accessed and modified. If a modification took place, all nodes must be sequentially serialised back to disk. As such, the mismatch between XML's fine and the disk's coarse granularity consequently leads to a loss of efficiency when it comes to random access or modifications.

Things change considerably when taking flash into account. With the finer granularity, each XML node is directly accessible by its key or position in the XML tree. The requirement to physically cluster related nodes can be dropped. Sequentially accessing physically dispersed nodes on flash-based storage will be in the same order of magnitude as accessing physically clustered nodes on a disk. As a side effect, memory can be used much more efficiently to just cache the frequently used nodes instead of caching all nodes.

The evolution of persistent data structures backs our observation. In the early days, merge sort was the prevalent method to keep data organised on tapes. Now, while merge sort is still a valuable topic to teach and now and then appears in practice, b+trees or even hash storage dominate the field. The practical implication of this development is impressive. Tape-based systems frequently run merge sort to avoid data fragmentation due to insertions or deletions. Disk-based systems intermittently de-fragment their file system trees for the same purpose. In stark contrast, flash-based systems are indifferent as the performance does not degrade with scattered data. Generally speaking, shorter average random access time leads to less management overhead due to data fragmentation.

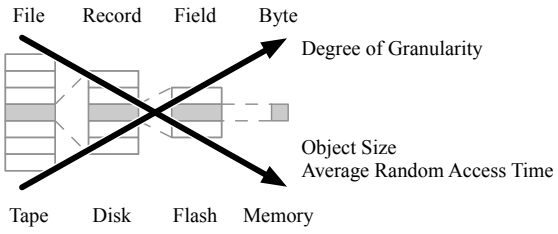

Figure 1: Degree of granularity in relation to average random access time and object size.

2.3 Evolution of State

Each modification evolves an existing state into a new

[1]Prices based on Internet research as of the time of writing.

one. Both the modification and the state are bound to a given point in time. The current state is an aggregate of all past modifications taking place during a given time period. Often, the modification is small compared to the new state it creates. Again, we find a mismatch between fine-grained modification and coarse-grained state. Figure 2 illustrates the relationship between the state and its evolution.

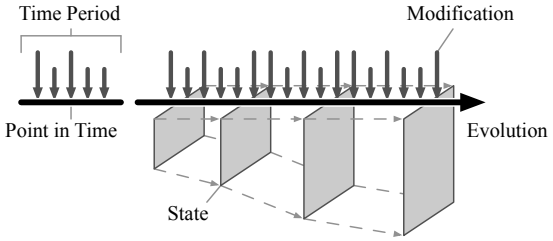

Figure 2: Evolution of state through a sequence of modifications.

Given the constraint to cluster data due to poor random access performance, the system can either try to cluster modified data in-place by overwriting old data or out-of-place by writing it to a free place. Given the constraint to overwrite clustered data due to the limited capacity, out-of-place traditionally is only a choice when the old place is marked as free. Looking back to our OpenDocument example from the Introduction, it becomes clear why a user cannot effectively modify even a medium-sized XML file. Before the modification, the XML is sequentially read into memory. Now the user makes a small modification by, say, adding a new XML node representing a section title in the middle of the XML node tree. Instead of making this tiny modification persistent, the whole XML must be sequentially written back to disk overwriting the old XML file. Both the old file and the modification are lost. What remains is the current state.

The fact that the system only knows the latest, i.e., current, state is widely accepted as the user sees the result of her modification. However, big efforts are required as soon as the user wants to do backup, undo, or redo operations. As for the backup, this can currently only be achieved manually by the user or with a separate application that laboriously determines the difference between the last backup and the current state. Once the difference is detected, an incremental delta is backed up. As for undo or redo operations, they currently either only span a single session between the opening and closing of the document or they have to be stored as a modification history together with the actual data. It would be much easier if the system inherently knew about the modification history and would be able to persistently reproduce the state after each modification.

Flash is well suited to model fine-grained modifications. The system gains the freedom to decide whether it should make the modification or the state resulting from this modification persistent. The former allows to quickly answer questions about the modification history, the latter to swiftly reconstruct the state at a given point in time. E.g., the insertion of the section title can be achieved by simply storing the remark about the inserted XML node or by storing the whole sub-tree as it looked like after the insertion. The amount of data written is negligible for the modification-only when compared to the whole-state variant. As a side-effect to the modification-only variant, the backup application could ask for the last modification and just backup the freshly inserted XML node. Furthermore, the user could decide right after the next start of her application whether she would like to undo the new section title.

2.4 Related Work

In 2.2 and 2.3, we analysed the limitations of traditional persistent storage and assumed a traditional file system as the intermediary between XML and storage. In this subsection, we analyse advanced systems which employ sophisticated data structures to push the limits set by magnetic disks. We investigate related work in the area of file, revisioning, database, and XML systems. All of these systems largely depend on persistent storage. We look at its impact on how each system organises data.

2.4.1 File Systems

We perceive six major developments in the area of file systems. First, transactional object store. Second, copy-on-write. Third, end-to-end integrity. Fourth, file system event messaging. Fifth, full text index. Sixth, event-based backup.

While the transactional object store[2] of Sun's ZFS [19] currently works at file-level granularity, it is readily available for storing finer-grained objects. ZFS is one of the first widespread file systems to integrate transactional behaviour on the basis of a single write transaction combined with concurrent reads. The copy-on-write in ZFS is a mix of a log-structured [21] and a traditional file system. The former only appends data and the latter only overwrites data. ZFS write transactions append data. But in case the user does not mark it as a long-term snapshot, the freshly appended data eventually gets overwritten to save capacity. The end-to-end integrity is an important feature as it can deal with many failure scenarios uncovered with simple error detection codes on various underlying layers. It even allows for cryptographic-strength integrity checks and is a tribute to the ever growing capacity as each hash or message authentication code [6] allocates up to several dozen bytes.

Apple introduced file system event messaging, full text index, and event-based backup with FSEvents [2], Spotlight [23], and Time Machine [3] respectively. To make the best out of these new technologies, Apple pragmatically splits large files into many small files. E.g., a single file which used to store multiple mails or events is split into multiple files, each storing a single mail or event. Finer semantic data granularity leads to a higher precision when it comes to communicate file modifications to applications such as the full text index Spotlight – another tribute to shrinking average random access times. Notably, the modification events are made persistent and aggregated gradually not to waste too much capacity. A notorious user of the novel event

[2]Also known as Data Management Unit (DMU).

messaging framework is Time Machine. Instead of intermittently searching and calculating the deltas between the last backup and the current state to perform an incremental backup, Time Machine asynchronously consumes fine-grained modification events and only backups the affected files. Note that still whole files are backed up but since the files get smaller, the granularity is finer as when compared to other systems.

2.4.2 Revisioning Systems

Revlog [15] is an important contribution in the area of revisioning systems. Revlog stores deltas of files which change during a revision. Each delta is derived by a diff algorithm comparing the last file revision with the current one. Intermittently, a full snapshot is stored to accelerate the retrieval of a file in a given revision. Without snapshots, Revlog would have to reconstruct the original file by sequentially applying all intermediary deltas up to the requested revision, starting from the current one. As such, Revlog allows to efficiently keep a revision history of all deltas. Both the state of a file as well as the modification evolving this state are derivable with reasonable effort. Still, the diff algorithm is time-consuming and not aware of the modification semantics.

2.4.3 Database Systems

With database systems, we find an interesting approach with Berkeley DB Java Edition which stores a traditional b+tree in a log-structured fashion [20]. B+trees play an important role in database systems. They allow to keep the data sorted and to quickly retrieve it both sequentially and randomly. However, the data must still be clustered not to over-stretch the capabilities of traditional storage and the tree must be kept balanced after modifications. Like ZFS, Berkeley DB clusters writes and appends them sequentially. Berkeley DB does not store the modifications and is just able to reconstruct the last successfully committed state. Furthermore, it has to reconstruct the b+tree in-memory to provide reasonable random access to the physically scattered data.

Recent work tries to tackle the comparably long write time of flash when the write occurs for the second time due to the block erasure procedure. FlashDB [16] tries to self-tune its b+tree by analysing the workload and switching between a disk, i.e., random, and a log-structured mode. In-page logging (IPL) [14] tries to take advantage of both the traditional in-place and the log-structured approach. IPL tries to minimise the number of block erasures by reserving a small space in each page for future modifications. Only if the reserved space is consumed, the page must be written to a new location – potentially involving a block erasure.

2.4.4 XML Systems

XML systems are also known as (native) XML (database) systems. Interestingly, the average user still stores most XML as flat files instead of one of these optimised alternatives. We distinguish four approaches by how XML systems locate a node in the fine-grained unranked ordered XML tree. First, fixed-size key. Second, variable-size key. Third, positional key. Fourth, index-based key. Table 2 lists the four types and compares key stability, global order, whether it is extractive, and random write performance.

Type	**Key Stab.**	**Glob. Ord.**	**Ext.**	**Rand. Write**
Fixed	yes	no	yes	++
Var.	yes	yes	yes	+
Pos.	no	yes	yes	−
Index	no	yes	no	−−

Table 2: Comparison of the main approaches on how to store XML with fine-grained data structures.

Persistent DOM [12] is an example for fixed-size keys. Each node is located by a unique immutable key of fixed size. This key does not necessarily reflect global order and must be stored. Modifications are efficient because they involve at most the modified, parent, left sibling, and right sibling node. However, updates do not scale when a node contains a large number of child references. Reads on the other hand are only efficient if the global order of two nodes must not be established.

ORDPATH [18] is an example for variable-size keys. Each node is located by a unique immutable key of variable size. This key maintains global order and must be stored. The key is derived during node insertion depending on the location of the node in the tree. Hotspots seeing frequent node insertions within the same sub-trees lead to long keys and thus restrict write scalability. Sequential and random reads can be done efficiently unless long keys are involved.

XPathAccelerator [11] is an example for positional keys. Each node is located by its unique mutable position in the tree. The position respects global order and is not directly stored. Writes are efficient with a positional b+tree [9]. However, the usual implementations have a limited update capability which is only achieved by leaving gaps in the positional numbering. Reads are scalable as long as only forward axes are involved. BaseX [10] is a special implementation that shrinks the size of each node by further dropping support for the preceding axis. Tightly packing XML nodes allows to store more of them in-memory and to notably accelerate processing.

Virtual Token Descriptor (VTD) [25] is an example for index-based keys. VTD is the only non-extractive system as it does not extract strings but directly references them at their position in the original XML file. Each reference is of fixed size and equivalent to an index key. Write scalability is provided, as long as VTD can sequentially process XML files. Random insertions or deletions still require to re-serialise both the XML file and the index. As such it is a close relative of the positional key approach. Reads are efficient for sequential access. Random access must be supported with the help of in-memory location caches linking parents and their respective first child.

2.4.5 Summary

Our analysis of related work shows a trend towards finer granularity from the semantic and the storage perspective. In addition, more and more systems try to

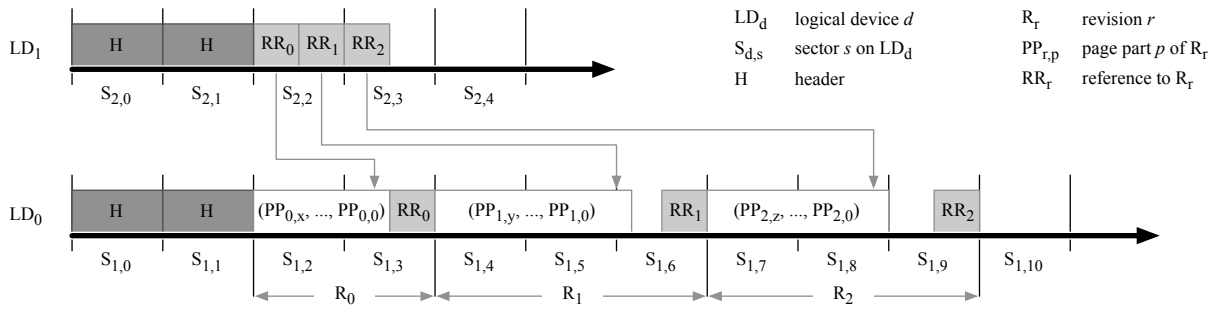

Figure 3: Layout of two logical devices storing an instance of TreeTank consisting of three revisions.

introduce some notion of evolution and past state complementing the common current-state-only philosophy. Consequently, all systems challenge the limits given by average random access time as well as capacity and closely follow the technological development towards faster random access and growing capacity. Nevertheless, most systems still assume mechanical disks as their underlying storage which leaves room for further improvements when designing for flash as shown by FlashDB and IPL.

3. TREETANK

From a logical perspective, TreeTank stores all revisions of an unranked ordered tree of pages. Each page stores a set of page references pointing to other pages as well as a set of nodes containing the application-specific data. From a physical perspective, TreeTank stores the per-page and per-revision modifications as page deltas. Note that a delta is not the result of an expensive diff calculation but just the plain modification event. Intermittently, a full page snapshot is stored for each page to fast-track its in-memory reconstruction. Consequently, TreeTank can quickly derive the state of each node in each revision as well as the modification history of each node between two revisions.

TreeTank was designed with security in mind. This involves the security primitives authentication, confidentiality, integrity, non-repudiation, access control, and availability. According to Schneier [7], the user is only left with one option, i.e., whether security is turned on or off. If activated, a small set of secure, fast, and time-proven algorithms is used: CTR-AES-256 [17, 5] for encryption, SHA-256 [4] for key salting and stretching, and HMAC-SHA-256 [6, 4] for authentication. The following paragraphs assume deactivated security due to space constraints.

Each instance of TreeTank is bound to a session. The session allows a single write and multiple concurrent read transactions at any time. The write transaction is bound to the latest successfully committed revision and allows to modify it in-memory. A new revision is created and all modifications are serialised sequentially when the write transaction commits. Each read transaction is bound to a committed revision and allows to read the page tree in this revision.

The logical device layout is depicted in Figure 3. TreeTank stores all data and metadata on the primary logical device LD_0. The secondary logical device LD_1 just contains replicated metadata for safety and performance reasons. Both logical devices may grow to the right by appending more sectors $S_{d,s}$. To prevent wear-out of the flash device, data is only appended. To provide optimal write performance, data is only written sequentially. The header H contains the configuration data and is replicated four times. The revision reference RR_r pointing to revision R_r is replicated twice. The page snapshots and deltas are stored once.

Binary search is used twice with TreeTank. First, it finds the last successfully committed revision. Second, it finds the closest revision number for a given point in time. In both cases, binary search works on the array of revision references stored on LD_1. TreeTank guarantees that at least one revision reference, i.e., RR_0, exists. A revision reference is valid if the first eight bytes are not zero. To find the last successfully committed revision, binary search looks for the right-most valid revision reference. With each chosen median, the binary search continues to the right, if the revision reference is valid, else, it continues to the left. To find the closest revision number for a given point in time, binary search asserts the validity of each chosen revision reference and then compares the provided point in time with the stored one. The search finishes, if either an exact match was found or the smallest possible time difference.

In a nutshell, b+trees always cluster data within each page of the tree. TreeTank only clusters data during snapshots and usually just stores deltas. This trades random access time for space and availability of the full modification history. E.g., a rough approximation (calculations are based on Table 1) shows that a magnetic-disk-based b+tree with five levels requires 14.5$[ms]$ to find a data item. A flash-based TreeTank with five levels and ten deltas per level on average requires 2.5$[ms]$ to find the same data item. TreeTank can tune the snapshot frequency to adapt itself to the available storage and workload. Furthermore, it does not depend on in-memory caches to speed-up its operation.

4. CONCLUSION

We analysed why it is currently not possible to efficiently and effectively modify large disk-based sets of XML data. We identified traditionally poor average random access times of mechanical disks as a major problem. Flash-based storage smashes this technological hurdle twofold. First, it prepares the ground to

align the degree of granularity of logical and physical data models to enhance efficiency. Second, it allows to store fine-grained modifications instead of coarse-grained state to improve the effectiveness of the user's workflow. An overview of state-of-the-art file, revisioning, database, and XML systems shows the trend towards finer-grained data structures to better model user requirements. However, the trend is bound by technological progress of mechanical disks and does not yet consider flash as its underlying storage.

We suggest TreeTank as a system to take full advantage of flash-based storage while not dropping support for erstwhile mechanical disks. TreeTank enables node-level access faster than traditional systems and without their extensive memory requirements. Additionally, TreeTank endows the user with the freedom to query both the node state for a given revision as well as the node modification history between two revisions. TreeTank provides a scalable, lightweight, transactional, secure, and persistent framework facile to implement, dependable to run, and modest to maintain.

Future work includes a detailed specification of TreeTank, as well as extensive benchmarks measuring a variety of XML documents and workloads on both mechanical and flash-based storage. As TreeTank mainly is a log-structured system, we will investigate, how to prune old revisions to reclaim space which is not yet abundant on flash. In addition, we will look at the distribution of TreeTank and its adoption for other applications such as full-text indices. After all, it is close to the author's heart to investigate how TreeTank can be made an integral part of a flash-based and thus a more energy-efficient and sustainable IT system.

5. REFERENCES

[1] A-DATA. Vitesta Extreme Edition. `http://techgage.com/article/a-data_2gb_pc2-8000_vitesta_extreme_edition/`, 2008.

[2] ArsTechnica. FSEvents. `http://arstechnica.com/reviews/os/mac-os-x-10-5.ars/7#fsevents`, 2007.

[3] ArsTechnica. Time Machine. `http://arstechnica.com/reviews/os/mac-os-x-10-5.ars/14#time-machine`, 2007.

[4] Federal Information Processing Standards Publication 180-2. *Secure Hash Standard*. National Institute of Standards and Technology, August 2002.

[5] Federal Information Processing Standards Publication 197. *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, November 2001.

[6] Federal Information Processing Standards Publication 198. *The Keyed-Hash Message Authentication Code (HMAC)*. National Institute of Standards and Technology, March 2002.

[7] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley Publishing, 2003.

[8] Fusion-io. ioDrive. `http://www.fusionio.com/iodrivedata.pdf`, 2008.

[9] C. Grün, A. Holupirek, M. Kramis, M. H. Scholl, and M. Waldvogel. Pushing XPath Accelerator to its Limits. In *Proceedings of EXPDB 2006*, Chicago, IL, USA, 2006.

[10] C. Grün, A. Holupirek, and M. H. Scholl. Visually Exploring and Querying XML with BaseX. In *12. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW 2007)*, Aachen, Germany, March 2007. (Demo).

[11] T. Grust. Accelerating XPath Location Steps. In *SIGMOD Conference*, 2002.

[12] G. Huck, I. Macherius, and P. Fankhauser. PDOM: Lightweight Persistency Support for the Document Object Model. German National Research Center for Information Technology, Integrated Publication and Information Systems Institute, 1999.

[13] IBM. 305 RAMAC. `http://en.wikipedia.org/wiki/IBM_305`, 2008.

[14] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 55–66, New York, NY, USA, 2007. ACM.

[15] M. Mackall. Towards a better SCM: Revlog and Mercurial. *Ottawa Linux Symposium*, July 2006.

[16] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 410–419, New York, NY, USA, 2007. ACM.

[17] NIST Special Publication 800-38A. *Recommendation for Block Cipher Modes of Operation*. National Institute of Standards and Technology, 2001.

[18] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 903–908, New York, NY, USA, 2004. ACM Press.

[19] OpenSolaris. ZFS Documentation. `http://www.opensolaris.org/os/community/zfs/docs/`, 2004.

[20] Oracle. Oracle Berkeley DB Java Edition. `http://www.oracle.com/database/berkeley-db/je/index.html`, 2008.

[21] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[22] Seagate. Savvio 15K. `http://www.seagate.com/docs/pdf/datasheet/disc/ds_savvio_15k.pdf`, 2008.

[23] A. Singh. *Mac OS X Internals*. Addison-Wesley Professional, 2006.

[24] Sun. StorageTek T10000 Tape Drive. `http://www.sun.com/storagetek/tape_storage/tape_drives/t10000/specs.xml`, 2008.

[25] XimpleWare. VTD-XML. `http://vtd-xml.sourceforge.net/`, 2004.