Bulletin of the Technical Committee on

Data Engineering

June 2013 Vol. 36 No. 2

IEEE Computer Society

Letters

Special Issue on Main-Memory Database Systems

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

Bulletin Matters

Readers will, perhaps, notice that the issue available on the web has subtlely changed in format. The individual papers accessed via the Bulletin web sites are now correctly page numbered, and the letters have the letter writers names included. This is an example of teaching an old dog new tricks. The teacher is Sudarshan, and I am the old dog. It seems that the dvipdfm program that I have been using to generate the entire issue can, properly parameterized, also produce individual papers quite nicely. Who knew? Well, Sudarhan knew, and now, thanks to him, I do also.

TCDE and ICDE Activities

Once again, the organizing committee for ICDE have done a fine job in hosting a top-notch conference, this time in Brisbane, Australia. The technical program was first-rate and the social calendar was a real plus.

An interesting new event at the conference was a TCDE Reception, initiated by Kyu-Young Whang, the TCDE Chair, and organized by the local ICDE organizers. This is a new step in attracting new members to the TCDE and to providing them with information and "enticements" to join.

Our new Working Group on Cloud Data Management held a workshop in conjunction with ICDE. This issue of the Bulletin contains a report on this workshop by the program chairs, Ashraf Aboulnaga and Carlo Curino.

The Current Issue

"New hardware, new problems!" That is one way to view the new hardware scene of multi-core processors, memory hierarchies, flash storage, and increasingly "remote" disk storage. Another way to view it is "New Hardware, new opportunities!" There is enormous compute power in modern hardware. And coupled with enormous main memory, query latency can be dramatically reduced. These open the door to order-of-magnitude throughput improvements in database engines.

But order of magnitude gains never come easily. The new hardware landscape requires new database system architectures, and new techniques, for potential gains to be turned into real gains. No longer can we simply count instructions to determine the performance of our algorithms. Even counting memory references will not do the job. Careful attention must be paid to threading models, to reference patterns, to potential blocking behavior, to how to provide durability, particularly in the presence of system crashes.

This brings me to the current issue, which is on main memory database systems. Paul Larson, the issue editor, is a colleague of mine at Microsoft Research in Redmond. He has been intimately involved with the SQL Server Hekaton main memory DBMS effort. So he knows this technical area thoroughly. He also knows about competitive systems and research efforts in this space. The result is that Paul has assembled an issue that brings together both industrial and research contributions that are truly revolutionizing DBMS performance, especially in the OLTP space, where it is frequently possible for the entire database to fit into main memory.

I am very partial to issues such as the one Paul has produced here with the help of authors. Industrial folks and research folks have much to learn from each other. And the current issue is a great example of this. Many of the successful approaches to great performance gains for DBMSs are described in these pages. I want to thank Paul for doing a fine job with the issue and to encourage you all to read and enjoy the papers contained in the issue.

David Lomet
Microsoft Corporation

Second International Workshop on Data Management in the Cloud

1 Introduction

The Second International Workshop on Data Management in the Cloud took place on April 8, 2013 in Brisbane, Australia, on the day before ICDE. The DMC workshop aims to bring researchers and practitioners in cloud computing and data management systems together to discuss the research issues at the intersection of these two areas, and also to draw more attention from the larger data management and systems research communities to this new and highly promising field. The DMC Workshops are sponsored by the IEEE TCDE Workgroup on Cloud Computing.

The DMC 2013 program began with a keynote presentation by Amr El Abbadi, Professor at the University of California, Santa Barbara. This was followed by six technical papers presented in two sessions. The workshop concluded with a panel discussion on research challenges in data management for the cloud.

2 Keynote

Amr El Abbadi is a Professor of Computer Science at the University of California, Santa Barbara. He is an established researcher in the areas of databases and distributed systems. He is an ACM Fellow and an AAAS Fellow, and has been Program Chair for multiple database and distributed systems conferences, including the ACM Symposium on Cloud Computing (SoCC) 2011. Amr's keynote talk was titled "MIND THE GAP: Managing Multi-Data Center Data." It focused on transaction management issues that arise when the data is distributed across a "wide gap", that is, in multiple data centers.

Amr pointed out that storing data in a geographically distributed, multi-data center setting is required by many applications for performance and fault tolerance. In such an environment, large round trip times for network messages increase the cost of coordinating transactions among replicas of a data item. One can address this problem by weakening the transactional guarantees offered to applications, but that makes application development quite difficult for programmers. Instead, it would be highly desirable to give programmers full ACID transactions in a multi-data center setting. Amr provided an overview of the problems that need to be solved to achieve this goal, and presented work from his group that addresses several of these problems. Amr concluded by pointing out the relationship between the topic of consensus from distributed systems and the topic of commitment from database transactions, and noted that the solutions to many of the problems in multi-data center data management may be found at the confluence of these two areas.

3 Paper Sessions

The six papers presented at the workshop dealt with a wide range of research issues related to data management in the cloud. The first three papers, presented in the morning session, dealt mostly with issues related to cloud data management infrastructure. The three papers in the afternoon session dealt mostly with issues related to cloud data management applications.

Rao et al. [1] presented a protocol and system for replicating data on-demand between different clusters running analytics systems, in particular Hadoop. The protocol is motivated by the observation that data on different clusters is shared by different applications, and replicating this data on-demand saves storage cost and bandwidth as compared to bulk copying the data.

Jin et al. [2] presented a mechanism for implementing materialized views in the Cassandra distributed storage system. The mechanism enables fast access based on secondary keys in Cassandra, and keeps the materialized views updated in the face of insertions and deletions. The paper compares different alternatives for secondary key access and explores the tradeoffs in these alternatives.

Duggan et al. [3] presented a model for predicting database performance under changing hardware configurations. Unlike previous work in this area, the proposed model can be trained on one hardware configuration and be used to predict performance on other hardware configurations. The model relies on workload fingerprinting to characterize workloads and collaborative filtering models to predict performance.

Baralis et al. [4] presented an approach for supporting frequent itemset data mining on multi-core servers. The approach relies on a disk-based data structure called the VLDBMine data structure. The paper presents techniques to create this data structure in parallel and to exploit it for data mining.

Li et al. [5] proposed that data centers can be used to reduce the variance in electrical load on a power grid. The basic idea is that data centers can migrate jobs between them to adjust their energy consumption in response to fluctuations in the load on the power grid. The paper presents a dynamic pricing approach that can be used to provide incentives for data center operators to perform this migration.

Finally, Künsemöller et al. [6] presented a study of business models for caching in internet service providers (ISPs). Content providers on the internet can use content delivery networks (CDNs) to cache their data in order to improve performance. It should be possible for ISPs to provide similar caching, and the paper uses a game-theoretic approach to investigate models under which it would be beneficial to the ISP and/or the content provider to adopt a particular style of caching.

4 Closing Panel

The workshop concluded with a panel entitled "Data Management in the Cloud - Where are We? And Where to Next?" The panel consisted of five distinguished panelists: Hakan Hacıgümüş from NEC Labs America, Sriram Rao from Microsoft, Boon Thau Loo from the University of Pennsylvania, David Maier from Portland State University, and Markus Weimer from Microsoft.

Hacıgümüş presented an overview of database management issues in the cloud, noting that there is a strong desire to support SQL on the cloud. This requires advances in several areas, such as service level agreements (SLAs) and workload prediction.

Rao noted that for analytical workloads, a significant fraction of MapReduce jobs in production workloads is written in high-level languages such as Pig Latin or Hive, so the focus of research in this area should expand beyond supporting individual MapReduce jobs to supporting Pig Latin and Hive on the cloud. He also noted the importance of investigating multi-tenancy for MapReduce.

Loo suggested that using the cloud for mission critical applications gives rise to many interesting research problems. He also pointed out opportunities for leveraging Openflow software defined networks in cloud data management. In addition, he suggested that formal methods can hold promise for solving some cloud data management problems.

Maier presented work on serving scientific data from a cloud environment. He pointed out research problems that are unique to this application area, especially for scientific domains that produce massive data sets. Issues such as latency and monetary cost become important in this context.

Weimer pointed out limitations in the support for machine learning provided by current cloud data management technologies. He argued that there is a need to better support the end-to-end machine learning workflow, from data capture through to learning and all the way to deployment. He also noted that better integration of machine learning concepts such as error tolerance for certain computations can result in higher performance and better models.

The panel concluded with a discussion session in which the audience presented their questions and views on the workshop topic.

References

[1] Sriram Rao, Benjamin Reed, and Adam Silberstein. HotROD: Managing grid storage with on-demand replication. In *Proc. IEEE Int. Conf. on Data Engineering, Workshop on Data Management in the Cloud (DMC '13)*, pages 243–249, 2013.

[2] Changjiu Jin, Rui Liu, and Kenneth Salem. Materialized views for eventually consistent record stores. In *Proc. IEEE Int. Conf. on Data Engineering, Workshop on Data Management in the Cloud (DMC '13)*, pages 250–257, 2013.

[3] Jennie Duggan, Yun Chi, Hakan Hacıgümüş, Shenghuo Zhu, and Ugur Çetintemel. Packing light: Portable workload performance prediction for the cloud. In *Proc. IEEE Int. Conf. on Data Engineering, Workshop on Data Management in the Cloud (DMC '13)*, pages 258–265, 2013.

[4] Elena Baralis, Tania Cerquitelli, Silvia Chiusano, and Alberto Grand. P-Mine: Parallel itemset mining on large datasets. In *Proc. IEEE Int. Conf. on Data Engineering, Workshop on Data Management in the Cloud (DMC '13)*, pages 266–271, 2013.

[5] Yang Li, David Chiu, Changbin Liu, Linh T.X. Phan, Tanveer Gill, Sanchit Aggarwal, Zhuoyao Zhang, Boon Thau Loo, David Maier, and Bart McManus. Towards dynamic pricing-based collaborative optimizations for green data centers. In *Proc. IEEE Int. Conf. on Data Engineering, Workshop on Data Management in the Cloud (DMC '13)*, pages 272–278, 2013.

[6] Jörn Künsemöller, Nan Zhang, and João Soares. ISP business models in caching. In *Proc. IEEE Int. Conf. on Data Engineering, Workshop on Data Management in the Cloud (DMC '13)*, pages 279–285, 2013.

Ashraf Aboulnaga and Carlo Curino
Program Committee Chairs
University of Waterloo and Microsoft

Letter from the Special Issue Editor

Most the major commercial database systems were designed in an era of small memories and uniprocessor machines. The only realistic design option was to store data on disk and bring disk pages into memory as needed for processing. This model served us well for many years but the world has changed. Today, a server with 32 processor cores and 1TB of memory costs as little as $50K. The majority of OLTP databases fit entirely in 1TB and even the largest OLTP databases can keep the active working set in memory. To take full advantage of this class of machines it is not sufficient to merely increase the size of the buffer pool. The architecture of the database system needs to be reconsidered from the ground up.

This issue aims to provide an overview of the current state of the art for database systems that have been designed to exploit large main memories and multi-core processors and are targeted for OLTP workloads. The issue covers seven different database systems, five commercial systems and two research prototypes. The commercial systems, roughly in order of launch date, are Oracle TimesTen, IBM solidDB, VoltDB, SAP HANA, and Microsoft's Hekaton. The two research systems are HyPer from the Technical University of Munich, and Calvin from Yale University. The issue also includes a paper on Bw-tree, a novel latch-free version of B-trees.

The design of any database system must solve a number of fundamental issues. What index structures to use? How to support high levels of concurrency? How to ensure transaction isolation and durability? What to do about long-running transactions? How to ensure high availability? It is fascinating to see the variety of solutions that the designers of the systems described here have come up.

The technique chosen for concurrency control, for example, has a major impact in system performance. Concurrency control adds overhead but also affects the level of concurrency that the system can support. VoltDB and HyPer partition the database and execute transactions serially within a partition. However, the system also needs to handle cross-partition transactions and the two systems use very different mechanisms for this. TimesTen, solidDB, Calvin, and Hekaton do not rely on partitioning but use different concurrency control algorithms. TimesTen uses a locking-based approach but uses lightweight latches instead of traditional locks. solidDB uses a similar approach but with a twist where readers avoid latching. Calvin has several alterative locking approaches which can be switched in and out for each other using Calvin's modular architecture, one of which (termed VLL) requires only lightweight semaphores in conjunction with a deterministic execution engine. Hekaton uses multiversioning to avoid interference between readers and writers and an optimistic concurrency control scheme

The system designers also opted for different index structures. For example, several systems support range indexes but implemented differently. TimesTen range indexes use T-trees, Hekaton uses Bw-trees while solidDB and HyPer both use tries but of very different design.

The papers in this issue provides a snapshot of the current state of the art and cover many innovate solutions. I would like to extend a heartfelt thanks to the authors for their efforts.

Happy reading! I trust that you will find the papers as interesting as I did.

Per-Ake Larson
Microsoft Research
Redmond, WA

Oracle TimesTen: An In-Memory Database for Enterprise Applications

Tirthankar Lahiri
tirthankar.lahiri@oracle.com

Marie-Anne Neimat
neimat@outlook.com

Steve Folkman
steve.folkman@oracle.com

Abstract

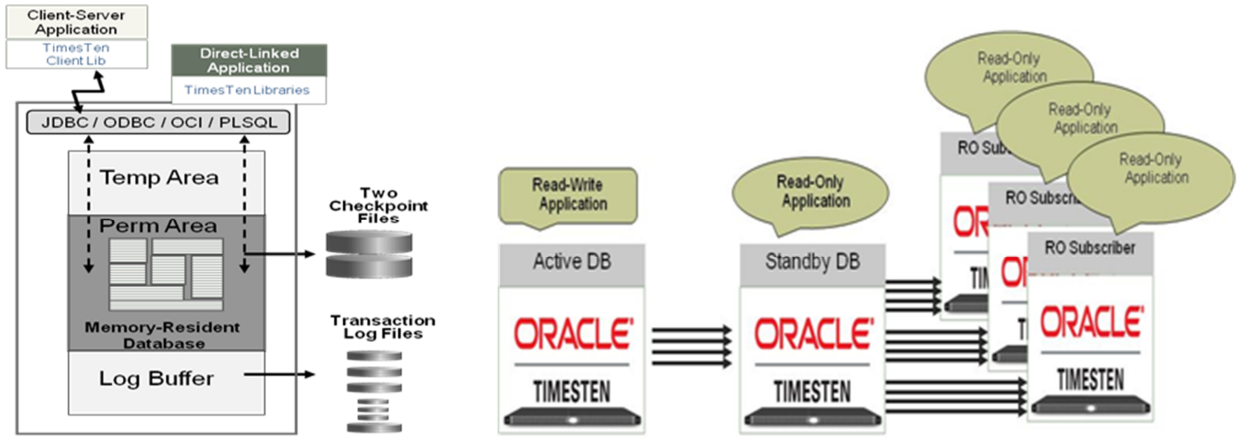
In-memory databases can provide a significant performance advantage over disk-oriented databases since they avoid disk IO, and since their storage managers are built and optimized for complete memory residency. In this paper we describe the functionality of the Oracle TimesTen In-Memory Database – a full-featured memory optimized relational database. With SQL-92 and ACID compliance, TimesTen is used by thousands of customers in a variety of applications – both for high-performance OLTP applications as well as for Business Analytics applications. While TimesTen can be used as a standalone database, it can also be used as a high-performance transactional cache that seamlessly caches data from an underlying Oracle RDBMS. This configuration is suitable for Oracle RDBMS applications that require real-time management of some of their data and scale-out on private or public clouds.

1 Introduction

With increasing DRAM densities, systems with hundreds of GigaBytes, or even TeraBytes of memory are becoming common. A large fraction of user data, and in many cases, all user data, can be stored entirely in DRAM avoiding the need for costly disk IO for query processing. As a result of this trend, there are numerous in-memory databases that are commercially available now.

H-store [18] is an in-memory database optimized for highly concurrent OLTP workloads. However H-store requires applications designers to have complete understanding of their data access patterns in order to avoid conflicts, and all database accesses must be performed in terms of one or more pre-defined stored procedures. H-store is therefore not well-suited for ad-hoc applications, nor for applications with complex transactions. MonetDB [4] is a database that is highly optimized for complex query workloads, employing a shared-nothing architecture and columnar processing. MonetDB however is not well-suited for write-intensive OLTP workloads and as such cannot be used as a general-purpose in-memory solution. In [5] the author describes a vision for a general purpose in-memory database for SAP applications based on columnar storage and compression. However, columnar storage is notoriously expensive to maintain in an update-intensive environment, and in our experience does not lend itself well to OLTP workloads characterized by ad-hoc queries and concurrent DML – often just single-row updates. MySQL cluster [1] is a widely used scale-out in-memory database for high throughput OLTP environments with stringent HA requirements, however it is not designed for analytics workloads.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering



(a) Basic structure

(b) TimesTen replication

Figure 1: TimesTen architecture and replication

Oracle TimesTen is a an enterprise-class in-memory database with a rich feature set, with the intention of being applicable to a wide variety of workloads and data-sets. TimesTen has full-featured SQL support and supports standard APIs, full ACID properties, and high-availability mechanisms. TimesTen can also be used to complement an Oracle RDBMS by providing a high-performance transactional cache for performance-critical data. This cache can be scaled-out to a grid across a networked cluster of nodes. It is because of this gestalt of features, that TimesTen is the most widely deployed in-memory database in existence today. It is used in many different application areas such as Telecommunications, Financial Services, Ecommerce, Fraud Detection, etc., all of which have stringent response time and throughput requirements. In response to the growing need for high-performance real-time analytics, Oracle recently announced the Exalytics In-Memory machine in which TimesTen is the in-memory relational repository for interactive BI workloads.

In this paper, we first describe the basic architecture of the Oracle TimesTen In-Memory Database. Second, we describe how TimesTen replication can be used to provide a highly-available configuration. Third, we describe the *In-Memory Database Cache* configuration and how it can be used to scale-out TimesTen to a grid of caches. We conclude with performance results.

2 Basic Architecture

The TimesTen database is a memory-resident and completely persistent database. As shown in Figure 1(a), TimesTen has a shared memory segment storing the database. The memory segment is divided into three areas of configurable sizes: a permanent portion storing database objects, a temporary portion for run-time allocations, and a buffer for the transaction log. The on-disk structures comprise 2 checkpoint files for capturing the state of the in-memory permanent partition, and transaction log files for logging changes to the permanent partition of the database.

2.1 APIs and connectivity

TimesTen provides standard APIs for database operations. These include ODBC, JDBC, as well as the Oracle Call Interface (OCI). Applications connect to TimesTen either via a conventional client-server protocol, or an optimized *direct-mode* protocol in which the application directly links the TimesTen library. In direct mode, all database API invocations are treated simply as function calls into the TimesTen shared library allowing for in-process execution of database code.

2.2 Basic storage manager design

Some basic design principles guide the design of the storage manager. Concurrency control mechanisms are both light-weight (e.g. using latches instead of locks whenever possible) and fine-grained to allow for maximal scaling on multi-core architectures. Another over-arching design principle is the use of memory-based addressing rather than logical addressing. For instance, indexes in TimesTen contain pointers to the tuples in the base table. The metadata describing the layout of a table contains pointers to the pages comprising the table. Therefore both index scans and tablescans can operate via pointer traversal. This design approach is repeated over and over again in the storage manager, with the result that TimesTen is significantly faster than a disk-oriented database, even one that is completely cached – since there is no overhead from having to translate logical rowids to physical memory addresses of buffers in a buffer cache.

Multiple types of indexes are supported by TimesTen. TimesTen supports Hash Indexes for speeding up lookup queries, Bitmap Indexes for accelerating star joins with complex predicates, as well as Range Indexes for accelerating range scans. Since the data is always in memory, indexes do not need to store key values, with significant space savings. Indexes are used far more aggressively by the TimesTen optimizer than by a disk-based optimizer: They are used for scans, joins, group by computations, etc. For instance, the optimizer often prefers index scans over full table scans since the CPU cost of index scans is usually lower, unless the column being scanned is compressed. Extensive use of indexes is an advantage in-memory databases have over disk-based databases as the use of indexes in disk-based databases will result in many disk seeks if the index traversal does not match the clustering of records on disk.

The TimesTen storage manager supports columnar compression using dictionary-based encoding. With this feature, the column values are replaced with dictionary table identifiers. Each column has a separate dictionary of unique values, and the compression domain spans the whole column (or a group of columns). Columnar compression has been observed to offer a 5-10x compression benefit (See Section 5.3) and helps to accommodate large datasets as is typical of BI and OLAP workloads. For example, the present generation Exalytics Machine has a DRAM capacity of 1TB. This allows for a conservative limit of 500GB for the TimesTen permanent memory segment (taking into account the needs of the OS, and the TimesTen temporary memory area) but with compression allows an effective capacity of 2.5-5 TB.

2.3 Transactional durability

TimesTen has been designed from the ground up to have full ACID properties. TimesTen employs checkpointing with write-ahead logging for durability. Logging is frequently a scaling bottleneck on large multi-core architectures, and for this purpose – again in keeping with the fine-grained concurrency control design motif, TimesTen has a multi-threaded logging mechanism so that generation of log is not serialized. In this mechanism, the TimesTen log buffer is divided into multiple partitions which can be populated in parallel by concurrent processes. Sequential ordering of the log is restored when the log is read from disk. For the highest performance, TimesTen offers an optional *delayed-durability* mode: an application commits by writing a commit record into the log buffer without waiting for the log to be forced to disk. The log is periodically flushed to disk every 100 mSec as a background activity. This mode of operation allows the highest possible throughput with the possibility of a small (bounded) amount of data loss, unless the system is configured with 2-safe replication as described in Section 3.

The database has two checkpoint files for the permanent data, and a variable number of log files. Once a database checkpoint is completed, the checkpointing operation switches to the other file. With this approach, one of the two checkpoint files is always a completed checkpoint of the in-memory contents and can be used for backup and roll-forward recovery. TimesTen supports fuzzy checkpointing and the user can configure the rate of checkpoint disk writes.

TimesTen offers read-committed transactional isolation by default (serializable isolation as an application

choice). With read-committed isolation, application bottlenecks are minimized since TimesTen has a lockless multi-versioning mechanism for read-write concurrency and row-level locking for maximal write-write concurrency. Row-level multi-versioning allows readers to avoid getting any locks altogether, while row-level locking provides the highest possible scalability for update-intensive workloads.

3 High Availability

The vast majority of TimesTen deployments use replication for high-availability. TimesTen replication is log-based, relying on shipping log records for committed transactions from a transmitter database to a receiver database, on which the changes in the log are then applied. Replication can be configured with multiple levels of resiliency. For the highest resiliency, TimesTen provides *2-safe* replication. With 2-safe replication, a transaction is committed locally only after the commit has been successfully acknowledged by the receiver. 2-safe replication is often used with non-durable commit. This combination allows applications to achieve commit durability in two memories, without requiring any disk IO.

Replication can be at the level of individual tables or at the level of the whole database. Replication can be configured to be multi-master and bi-directional. The preferred configuration for replication is to have whole database replication from an *Active* Database to a *Standby* Database. Standby Databases can in turn propagate changes to a number of *Read-Only Subscriber* Databases, as depicted in Figure 1(b). In this type of replication scheme, the Active Database can host applications that both read and modify the database. The Standby and the Read-Only Subscriber Databases can host read-only applications. If a failure of the Active occurs, the Standby can be promoted to become an Active Database. If a failure of the Standby occurs, the Active will then replicate directly to the Read Only Subscribers.

In keeping with the design principle of end to end parallelism, replication can be parallelized across multiple *replication tracks*. Each track consists of a replication transmitter on the sending side, and a replication receiver on the receiving side. Parallel replication preserves commit ordering, but allows transactions without any data dependencies to be applied in parallel by the receiver. Together with log parallelism, replication parallelism provides end to end parallelism between the master and the receiver, and the highest possible throughput.

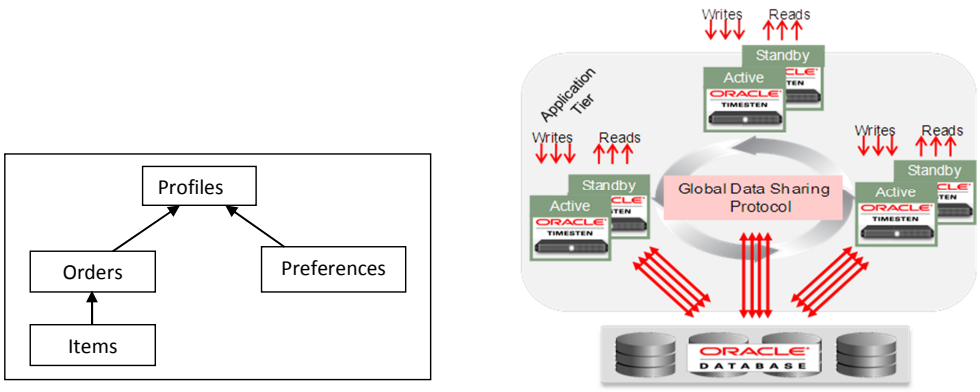
4 IMDB Cache Grid

TimesTen as we have shown is a full-featured database with durability, persistence and high availability and can be used as the only database of record for an application. However, in practice most enterprises still store their business critical data in full-featured disk-based database such as the Oracle RDBMS. The storage capacities and functionality far outweigh those of in-memory databases. For this purpose the IMDB cache configuration allows TimesTen to be deployed as a persistent transactional cache [2] for data in an Oracle database. This is a cache that the application must be aware of, since it is a full featured database embedded in the application tier.

Deploying TimesTen as an application-tier cache against a disk-based database can greatly accelerate an application even when the database working set is fully cached in memory within the buffer cache. This is for two reasons:

1. Application Proximity: The TimesTen database can be collocated in the application-tier, resulting in lower communication costs to the database for access to cached data. For the best response time, TimesTen can be directly linked to the application providing in-process execution of operations on cached data.

2. In-Memory Optimizations: As stated in Section 2, the TimesTen storage manager is built assuming full memory-residency; it requires far fewer instructions for database operations than disk-based databases.



(a) Example cache group

(b) IMDB cache grid

Figure 2: TimesTen cache group and cache grid

4.1 Cache groups

TimesTen allows a collection of tables to be declared as a *Cache Group* against corresponding tables on an Oracle database. A cache group is a syntactic declaration and comprises one *Root table*, and optional *Child tables* related by foreign key constraints. For instance, for an ecommerce application, one approach to creating a cache group would be to have a root table for all user profiles, and child tables for open orders placed by each user, and the list of preferences for each user. This cache group is depicted in Figure 2(a).

Cache groups can be loaded from an Oracle database in one of two ways:

Pre-Loaded: In this mode, the cache group is explicitly loaded before the workload is run. For instance in the above example, all users could be preloaded from the Oracle RDBMS into the cache group.

Dynamically Loaded: In this mode, data is brought into the cache when referenced. For instance, when a user logs in, it could be at that point that the user profile record for the user and all the associated Orders and Preference records are brought into the cache group so that all subsequent references by that user will benefit from in-memory processing (with a penalty on the first reference only). The data to be referenced must be identified by an equality predicate on the primary key of the root table (in this case, by the user profile Id). The root table row and related child table rows are referred to as a *cache instance*.

For dynamically loaded cache groups the user can set one of two different aging policies to remove data to ensure that the database does not run out of space. TimesTen supports time-based aging (age on the value of a timestamp column) or LRU aging, based on recency of usage. When the aging mechanism removes rows from a cache group, it removes them in units of cache instances. Cache instances form the unit of caching since they are atomic units for cache replacement.

Cache groups can be of two basic types in order to handle a variety of caching scenarios. There are two corresponding data synchronization mechanisms for these cache group types.

Read-Only Cache Groups: For data that is infrequently updated, but widely read, a read-only cache group can be created on TimesTen to offload the backend Oracle database. Very hot reference data, as online catalogs, airline gate arrival/departure information, etc. is a candidate for this type of caching. The Oracle side tables corresponding to Read-Only cache groups are updated on Oracle. The updates are periodically refreshed into TimesTen using an automatic refresh mechanism.

Updatable Cache Groups: For frequently updated data, an updatable cache group with write-through synchronization is appropriate. Account balance information for an online ecommerce application, the location of subscribers in a cellular network, streaming sensor data, etc. are all candidates for write-through caching. TimesTen provides a number of alternative mechanisms for propagating writes to Oracle, but the most commonly used and highest performing mechanism is referred to as *Asynchronous Write-through*, where the changes

are replicated to Oracle using a log-based transport mechanism. This mechanism is also capable of applying changes to Oracle in parallel, in keeping with the parallel-everywhere design theme of the system.

It is possible to deploy multiple TimesTen cache databases against a single Oracle database. This architecture is referred to as the *In-Memory Database Cache Grid* (depicted in Figure 4). Cache groups can once again be classified into two categories of data visibility on a grid.

Local Cache Groups: The contents of a local cache group are not shared and are visible only to the grid member they are defined on. This type of cache group is useful when the data can be statically partitioned across grid members; for instance, different ranges of user profile Ids may be cached on different grid members.

Global Cache Groups: In many cases, an application cannot be statically partitioned and Global Cache Groups allow applications to transparently share cached contents across a grid of independent TimesTen databases. With this type of cache group, cache instances are migrated across the grid on reference. Only consistent (committed) changes are propagated across the grid. In our example, a user profile and the related records can be loaded into one grid member on initial login by the user. When the user disconnects and later logs in onto a different grid member, the records for the user profile and related child table entries are migrated from the first grid member to the second grid member. Thus, the contents of the global cache group are accessible from any location, via data shipping.

The cache grid also provides data sharing via function shipping or global query. A global query is a query executed in parallel across multiple grid members. For example, if a global query needs to compute a COUNT(*) on the user profiles table, it would ship the count operation to all the grid members, and then collate the results by summing the received counts. This mechanism can be generalized to much more complex queries involving joins, aggregations, and groupings. Global queries are useful for running reporting operations on a grid, for example, what is the average value of the current outstanding orders for all users on the grid.

5 Experimental Results

5.1 Response time and throughput

The standard TimesTen throughput performance benchmark (TPTBM) models a telecommunication workload. The combination of in-memory execution and direct-linked access allow TimesTen to provide extremely low response times, see Table 1. This experiment was run on a standard commodity Intel Xeon 5670- system with 2 processors and 12 cores, running Oracle Enterprise Linux 5.6.

Scale up on multi-core systems is also evidenced by the throughput data, in which we measure peak throughput for a number of different workload mixes. These measurements were taken on a SPARC T5-8 system with 128 cores. On this system TimesTen achieves nearly 60 million TPTBM reads per second, and over 1 million TPTBM writes per second (this number is limited by the available IO bandwidth for logging and checkpointing). On a mobile call processing benchmark derived from a customer workload, TimesTen achieves 367K TPS on a SPARC T5-2 system with 32 cores.

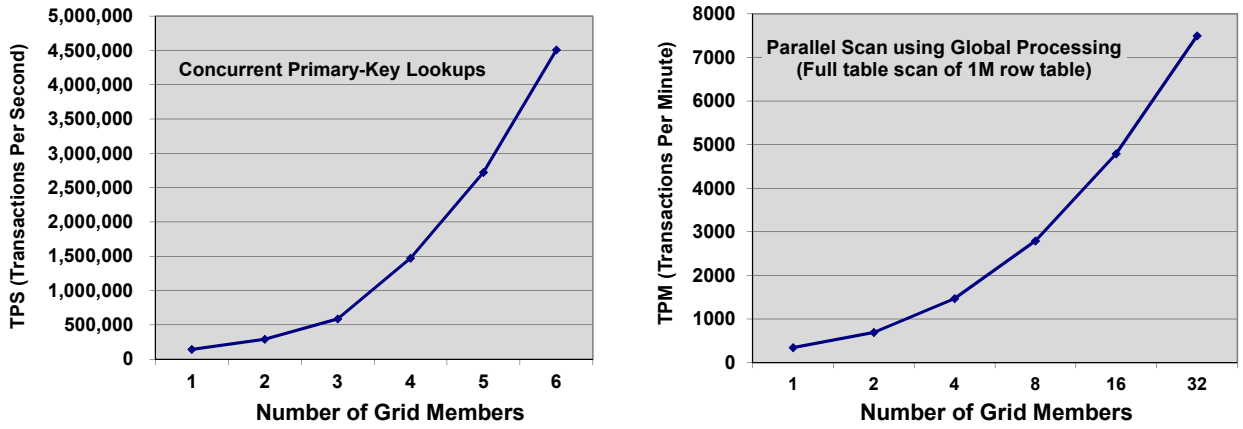
(a) Measured response time

Operation	**Response Time (microseconds)**
TPTBM Select	1.78
TPTBM Update	7.0

(b) Measured throughput

Benchmark	**Throughput (ops per second)**	**System (SPARC)**
TPTBM 100% read	59.9 million	T5-8, 128 cores
TPTBM 100% update	1.015 million	T5-8, 128 cores
Mobile call processing	367,000 txns	T5-2, 16 cores

Table 1: TimesTen response time and throughput



(a) Lookup on primary key

(b) Table scan

Figure 3: Concurrent lookup scaling on Cache Grid

5.2 Scaling across a grid

In this section, we show how TimesTen scales out when extended to the IMDB cache configuration. In Figure 3 (left graph), we show performance results from running concurrent primary key-based lookup queries across an IMDB grid simulated on a on older-generation Ultra SPARC T2 system using 2 CPU threads per grid member. As we can see that the grid scales very effectively as we increase the number of grid members. Here the key values are uniformly randomly distributed so that an increased percentage of the queries reference remote data as the number of nodes is increased.

Second, also in in Figure 3 (right graph), we show the performance of a global query that issues a full table scan on a million row table. In this test there are 6 application connections to one TimesTen grid member. The table being scanned is equi-partitioned across the grid databases. The scan is processed via global query execution in which the query is shipped to all grid members and the results compiled on the originating node. As shown below, the performance of global scans scales fairly linearly with the number of participating grid members.

5.3 Columnar compression

Next, we show the benefits of columnar compression on a database comprising aggregates computed from a CRM Data Warehouse star schema. This aggregated database has 29 tables. In Table 2, we show the distribution of compression factors achieved on individual tables in the schema. As we can see, 14 out of the 29 tables compress by more than a factor of 4x. The space savings from compression on this database is substantial: The total size of the tables in the database reduces from 19.5GB to around 3.16GB, or roughly a 6x reduction in size.

Compression factor	Number of tables
<2x	8
2x-4x	7
4x-6x	10
6x-8x	4

Table 2: Distribution of compression factors by table for CRM data warehouse aggregated star schema

6 Conclusions and Future Work

As we stated earlier, it is necessary for an in-memory database to provide a wide range of capabilities to be applicable to Enterprise workloads. We have shown that Oracle TimesTen is a full-featured relational in-memory database with ACID properties and transactional semantics. TimesTen has a storage manager optimized for in-

memory access and for high concurrency. TimesTen has full-featured SQL support with extensions for analytics applications, stored procedures via the PL/SQL language, and access via standard database APIs. TimesTen provides full database as well as table-level multi-master replication for high availability, and can be deployed as a grid of cache databases to cache selective data from an underlying Oracle database. We have also shown performance results showing that TimesTen provides very low response times as well as very good scaling characteristics, both on a single system as well as across multiple systems on a cache grid. It is because of this comprehensive list of capabilities that Oracle TimesTen is the most widely deployed commercially available in-memory database product today, with a wide range of usages ranging from high performance in-memory OLTP workloads to high-speed in-memory Analytics.

Future work for TimesTen includes extending the capabilities of the database to large scale-out clusters, instantaneous restart using persistent RAM technologies such as Phase Change Memory, continued optimization for high performance network technologies such as Infiniband, and optimistic, lockless algorithms for maximal scaling on very large SMP systems.

7 Acknowledgments

Oracle TimesTen is a complex and sophisticated team-product, and the authors are merely scribes for the functionality in the product. Many thanks are due to the Oracle TimesTen Engineering team: Development, QA, Documentation, and Product Management, for their tireless contributions to the product.

We owe our thanks to the Oracle RDBMS development teams particularly in the areas of NLS support, OCI/Pro*C support and PL/SQL support, in helping Oracle TimesTen to adopt these enterprise critical features.

We are indebted to our partner teams within Oracle such as the Oracle CGBU Billing and Revenue Management team for contributing valuable ideas to the IMDB cache grid architecture, and to the Oracle BI/EE team for their many contributions to the analytics support in TimesTen.

Last, but most importantly, we owe our thanks to the thousands of customers who have adopted TimesTen and provided the most valuable feedback possible for our product based on real-world usage-based requirements.

References

[1] *MySQL Cluster*. http://www.mysql.com/cluster.

[2] O. America. *Using Oracle In-Memory Database Cache to Accelerate the Oracle Database. An Oracle Technical White Paper.* http://www.oracle.com/technetwork/database/focus-areas/performance/wp-imdb-cache-130299.pdf, June 2009.

[3] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[4] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB J.*, 9(3):231–246, 2000.

[5] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD Conference*, pages 1–2, 2009.

IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability

Jan Lindström, Vilho Raatikka, Jarmo Ruuth, Petri Soini, Katriina Vakkila
IBM Helsinki Lab, Oy IBM Finland Ab
jplindst@gmail.com, raatikka@iki.fi, Jarmo.Ruuth@fi.ibm.com, Petri.Soini@fi.ibm.com,
vakkila@fi.ibm.com

Abstract

A relational in-memory database, IBM solidDB is used worldwide for its ability to deliver extreme speed and availability. As the name implies, an in-memory database resides entirely in main memory rather than on disk, making data access an order of several magnitudes faster than with conventional, disk-based databases. Part of that leap is due to the fact that RAM simply provides faster data access than hard disk drives. But solidDB also has data structures and access methods specifically designed for storing, searching, and processing data in main memory. As a result, it outperforms ordinary disk-based databases even when the latter have data fully cached in memory. Some databases deliver low latency but cannot handle large numbers of transactions or concurrent sessions. IBM solidDB provides throughput measured in the range of hundreds of thousands to million of transactions per second while consistently achieving response times (or latency) measured in microseconds. This article explores the structural differences between in-memory and disk-based databases, and how solidDB works to deliver extreme speed.

1 Introduction

IBM solidDB [1, 18] is a relational database server that combines the high performance of in-memory tables with the nearly unlimited capacity of disk-based tables. Pure in-memory databases are fast but strictly limited by the size of memory. Pure disk-based databases allow nearly unlimited amounts of storage but their performance is dominated by disk access. Even if the server has enough memory to store the entire database in memory, database servers designed for disk-based tables can be slow because data structures that are optimal for disk-based tables [13] are far from optimal for in-memory tables.

The solidDB solution is to provide a single hybrid database server [5] that contains two optimized engines:

- The disk-based engine (DBE) is optimized for disk-based access.
- The main-memory engine (MME) is optimized for in-memory access.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Both engines coexist inside the same server process and a single SQL statement may access data from both engines. The key components of solidDB MME technology (Vtrie and index concurrency control) are discussed in more detail in section 2.

To take the full advantage of the in-memory technology, you can also link your client application directly to the database server routines for higher performance and tighter control over the server. The shared memory access (SMA) feature of solidDB uses direct function calls to the solidDB server code. The in-memory database is located in a shared memory segment available to all applications connected to the server via SMA. Multiple concurrent applications running in separate processes can utilize this connection option to reduce response times significantly. This means that the applications ODBC or JDBC requests are processed almost fully in the application process space, with no need for a context switch among processes.

In addition to a fully functional relational database server, solidDB provides synchronization features between multiple solidDB instances or between solidDB and other enterprise data servers. The solidDB server can also be configured for high availability. Synchronization between solidDB instances uses statement-based replication and supports multi-master topologies.

The same protocol extends to synchronization between solidDB and mobile devices running the IBM Mobile Database. For synchronization between solidDB and other data servers, log-based replication is used. The solidDB database can cache data from a back-end database, for example, to improve application performance using local in-memory processing or to save resources on a potentially very expensive back-end database server. Caching functionality supports back-end databases from multiple vendors including IBM, Oracle and Microsoft.

The Hot-Standby functionality in solidDB combines with the in-memory technology to provide durability with high performance. This is discussed in more detail is section 3.

2 Makings of solidDB in-memory technology

Database operations on solidDB in-memory tables can be extremely fast because the storage and index structures are optimized for in-memory operation. The in-memory engine can reduce the number of instructions needed to access the data. For example, the solidDB in-memory engine does not implement page-oriented indexes or data structures that would introduce inherent overhead for in-page processing.

In-memory databases typically forgo the use of large-block indexes, sometimes called bushy trees, in favor of slimmer structures (tall trees) where the number of index levels is increased and the index node size is kept to a minimum to avoid costly in-node processing. IBM solidDB uses an index called trie [4] that was originally created for text searching but is well suited for in-memory indexing. The Vtrie and index concurrency control designs are discussed in more detail in sections 2.1 and 2.3.

The solidDB in-memory technology is further enhanced by checkpoint execution. A checkpoint is a persistent image of the whole database, allowing the database to be recovered after a system crash or other cases of down time. IBM solidDB creates a snapshot-consistent checkpoint that is alone sufficient to recover the database to a consistent state that existed at some point in the past.

2.1 Vtrie indexes

The basic index structure in the in-memory engine is a Vtrie (variable length trie) that is optimized variation of path- and level-compressed trie. Vtrie is built on top of leaf node level very much similar to those used in B+-trees. Key values are encoded in such a way that comparison of corresponding key parts can use bytewise comparison and end of keypart evaluates smaller than any byte value. Each Vtrie node can point to maximum 257 child nodes, one for each byte value plus “end of keypart”. Vtrie index stores the lowest key value of each leaf node, which guarantees the Vtrie index will not occupy excessive space even if the key value distribution would cause low branching factor. The worst case branching factor for Vtrie is 2 which occurs when key values

are strings consisting of only two possible byte values. The B+-tree -like leaf level guarantees the Vtrie still has far fewer nodes than there are stored key values in the index.

There are two types of key values stored in the index: routing and row keys. Routing keys are stored in Vtrie nodes, while row keys are stored at the leaf level, which consists of an ordered list of leaf nodes. Every leaf node is pointed to by one routing key stored in the Vtrie. The value of the routing key is higher than the high key of the node's left sibling and at most the same as the low key of the leaf node it points to.

Leaf node size varies depending on the keys stored to it but the default size is a few cache lines. Unlike B-, and binary trees Vtrie does not execute any comparisons during tree traversal. Each part of a key is applied as an array index to a pointer array of a child node. Contrary to a value comparison, array lookup is a fast operation if the array is cached in processor caches. When individual array index is sparsely populated, it is compressed to avoid unnecessary cache misses. Finally, on the leaf level, row key lookup is performed by scanning prefix-compressed keys in cache-aligned leaf node.

2.2 Main-memory checkpointing

A checkpoint is a persistent image of the whole database, allowing the database to be recovered after a system crash or other case of down time. IBM solidDB executes a snapshot-consistent checkpoint [10] that is alone sufficient to recover the database to a consistent state that existed at some point in the past. Other database products do not normally allow this; the transaction log files must be used to reconstruct a consistent state. However, solidDB allows transaction logging to be turned off, if desired. The solidDB solution is memory-friendly because of its ability to allocate row images and row shadow images (different versions of the same row) without using inefficient block structures. Only those images that correspond to a consistent snapshot are written to the checkpoint file, and the row shadows allow the currently executing transactions to run unrestricted during checkpointing.

2.3 Main-memory index concurrency control

The in-memory engine's indexing scheme [12] allows any number of readers to simultaneously traverse the index structure without locking conflicts. A read request on an index uses an optimistic scheme that relies on version numbers in index nodes to protect the integrity of the search path. The read-only traverse algorithm is presented on Figure 1.

```
retry:
  parent = root; current = root; parent_version = parent.version;
  traverse_ongoing = TRUE
  while traverse_ongoing loop
    current_version = current.version;
    if parent.version_mismatch(parent_version) then
      goto retry;
    child = current.child_by_key(key)
    if current.version_mismatch(current_version) then
      goto retry;
    if child.terminal() then
      result = child; traverse_ongoing = FALSE;
    else
      parent_version = current_version; parent = current;
    end if
  end loop
  return result;
```

Figure 1: Index concurrency control algorithm.

A write operation on an index uses a two-level locking scheme to protect the operation against other write requests.

To support the lock-free read-only traversal scheme, when a writer changes any index node it first increments the node version number (to an odd value). After the consistency of the path being modified is achieved again the (odd) version numbers are incremented again (to even). The method version_mismatch() above returns TRUE if either of these apply:

1. node.version <> version argument OR
2. version is an odd value.

To help a read request to complete the search even if the search path is being heavily updated, a hot-spot protection scheme is also implemented. A search that needs to run a retry several times due to a version conflict falls back to locking mode that performs the traversal as if it were a write request. In practice this is very seldom needed.

The above methods still needs some refinement: When a ”child” pointer is dereferenced, two issues arise.

1. Can we safely even read the version number dereferencing the pointer?
2. Even if the pointer value is still a legal memory reference, how do we know that the version number field mismatches in case the memory location has been freed and then reallocated for some other purpose?

Both those issues would certainly be problems if freeing of an unused tree node would be a direct call to the allocator's free()-function. To resolve this issue, solidDB uses a memory manager for versioned memory objects [11] that guarantees the version number field is neither overwritten by an arbitrary value nor decremented. The engine has a ”purge level” which is a deallocation counter value before the oldest index traversal started. When an index traversal ends, it may increase the purge level. When the purge level increases, freed nodes can be released to underlying memory manager. This ensures that the version-checking traversal remains safe even when index nodes are freed. Recycling of memory to new versioned memory objects, i.e. index tree nodes, is still possible, because unpurged freed nodes can be used as new versioned memory objects.

3 High Availability Through the Use of Hot-Standby Replication

IBM solidDB provides high availability through automated error detection and fast fail over functionality in two-node Hot-Standby cluster (HSB, for short). HSB can be efficiently combined with in-memory engine because replicating REDO logs provides failure resiliency without having to wait a single disk access before commit. Full durability is achieved by means of asynchronous logging. Prior to active HSB replication the nodes, namely Primary (master) and Secondary (standby, slave) establish a connection which guarantees that database in nodes are in sync. Connection is maintained during normal operation and monitored by heartbeat mechanism. When connected, Primary accepts write transactions, generates row-level redo log records of changes and sends them to Secondary. Secondary secures the consistency of standby database by first acquiring long-term locks for rows before re-executing log operations. As a consequence, conflicting updates [16] are serialized due the use of locks, but non-conflicting ones benefit from parallelized, multi-threaded execution in multi-core environments.

In an HSB database, transaction logs are sent to the Secondary server by way of a replication protocol. In order to preserve the database consistency in the presence of fail over, the replication protocol is built very much on the same principles as physical log writing: the transaction order is preserved, and commit records denote committed transactions. If a fail over happens, the standby server performs a similar database recovery as if a transaction log was used: the uncommitted transactions are removed and the committed ones are queued for execution.

Synchronous (2Safe) replication protocols provide varying balance between safety and latency [16]. However, they all ensure that there is no single point of failure which would lose transactions. There are three 2Safe variables available. 2Safe Received commits as soon as Secondary acknowledges that is has received transaction log. 2Safe Visible and 2Safe Committed both commit when Secondary has executed and committed the transaction, but when 2Safe Visible is used, Secondary sends acknowledgment to the Primary prior to physical log writing. Unlike 2Safe protocols, asynchronous replication protocol (1Safe) prefers performance over safety by committing immediately without waiting for Secondary's response.

When solidDB server is used in Hot-Standby mode, it is state conscious: inside solidDB there is a non-deterministic finite-state automaton, which at any moment, unambiguously defines the server's capabilities. For example, whether the server can execute write transactions, solidDB provides an administrative API for querying and modifying HSB states.

solidDB also provides a High-Availability Controller (HAC, for short) software to automatically detect errors in HSB and recover from them. Every HSB node includes a solidDB HSB server, and a HAC instance. In addition to sub-second failure detection and fail over, HAC handles all single failures and several failure sequences without downtime. Network errors are detected by using an External Reference Entity (ERE) [15]. External Reference Entity which can be any device sharing the network with HSB nodes, which responds to ping command. When HSB connection gets broken, HAC first attempts to reach the ERE to determine the correct failure type. If ERE responds, HAC concludes that the other server has either failed or become isolated. Only after that the local HSB server can continue (if it was Master) or start (otherwise) to execute write transactions.

4 Performance and Conclusions

In solidDB, the main focus is on short response times that naturally results from the fact that the data is already in memory. Additional techniques are available to avoid any unnecessary latency in the local database usage, such as linking applications with the database server code by way of special drivers. By using those approaches, one can shorten the query response time to a fraction (one tenth or even less) of that available in a traditional database system. The improved response time also fuels high throughput. Nevertheless, techniques of improving the throughput of concurrent operations are applied too. The outcome is a unique combination of short response times and high throughput.

The advantages of solidDB in-memory database over a disk-based database are illustrated in Figure 2 where the response time in milliseconds of single primary key fetch and single row update based on the primary key is shown.

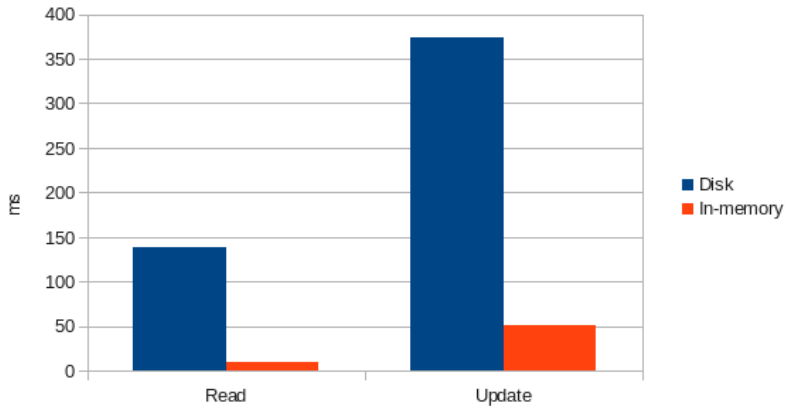


Figure 2: solidDB single operation response time.

Figure 3 shows the results of an experiment involving a database benchmark called Telecom Application Transaction Processing (TATP) [1] that was run on a middle-range system [2] platform. The TATP benchmark simulates a typical Home Location Register (HLR) database used by a mobile carrier. The HLR is an application mobile network operators use to store all relevant information about valid subscribers, including the mobile phone number, the services to which they have subscribed, access privileges, and the current location of the subscriber's handset. Every call to and from a mobile phone involves look ups against the HLRs of both parties, making it a perfect example of a demanding, high-throughput environment where the workloads are pertinent to all applications requiring extreme speed: telecommunications, financial services, gaming, event processing and alerting, reservation systems, and so on. The benchmark generates a flooding load on a database server. This means that the load is generated up to the maximum throughput point that the server can sustain. The load is composed of pre-defined transactions run against a specified target database.

The benchmark uses four tables and a set of seven transactions that may be combined in different mixes. The most typical mix is a combination of 80% or read transactions and 20% of modification transactions. The TATP benchmark has been used in industry [6] and research [7, 3, 8, 9, 14]. Experiment used database containing 1 million subscribers. The results of a TATP benchmark show the overall throughput of the system, measured as the Mean Qualified Throughput (MQTh) of the target database system, in transactions per second, over the seven transaction types. This experiment used shared memory access for clients and figure shows the scalability of the solidDB when number of concurrent clients are increased.

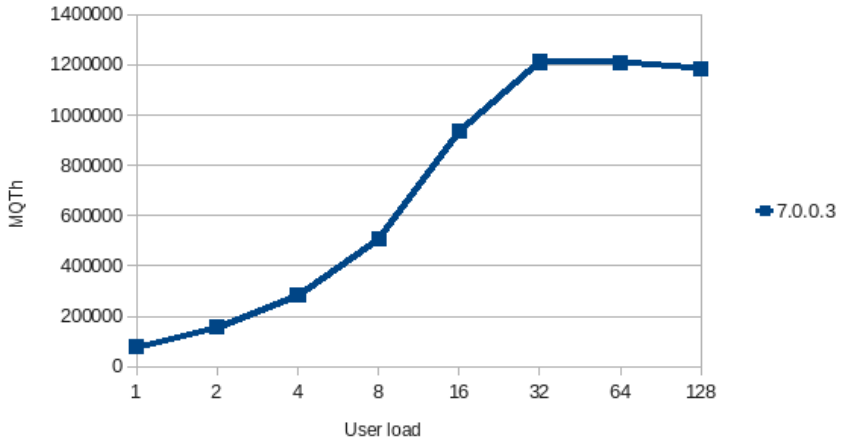


Figure 3: solidDB user load scalability.

In addition to telecom solutions, solidDB has shown its strength on various other business areas where predictable low-latency and high-throughput transactional data processing is a must, such as media delivery [2] and electronic trading platform.

References

[1] Chuck Ballard, Dan Behman, Asko Huumonen, Kyösti Laiho, Jan Lindström, Marko Milek, Michael Roche, John Seery, Katriina Vakkila, Jamie Watters and Antoni Wolski, *IBM solidDB: Delivering Data with Extreme Speed*, IBM RedBook, ISBN 07384355457, 2011.

[1] http:tatpbenchmark.sourceforge.net

[2] Hardware configuration: Sandy Bridge EP Processor: Intel Xeon E5-2680 (2.7 GHz, 2 socket, 8 cores16 threads per socket), Memory: 32GB memory (8 x 4GB DDR3 1333GHz), Storage: 4x 32GB Intel X25-E SSDs, OS: RHEL 6.1 (64-bit)

[2] *Fabrix Systems - Clustered Video Storage, Processing and Delivery Platform*, http://www-304.ibm.com/partnerworld/gsd/solutiondetails.do?solution=44455&expand=true&lc=en, IBM, 2013.

[3] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan , and Yun Wang: *A Novel Design of Database Logging System using Storage Class Memory*. ICDE 2011.

[4] Edward Fredkin: *Trie Memory*, Communications of the ACM 3 (9): 1960.

[5] Joan Guisado-Gámez, Antoni Wolski, Calisto Zuzarte, Josep-Lluís Larriba-Pey, and Victor Muntés-Mulero, *Hybrid In-memory and On-disk Tables for Speeding-up Table Accesses*, DEXA 2010.

[6] *Intel and IBM Collaborate to Double In-Memory Database Performance*, Intel 2009 http://communities.intel.com/docs/DOC-2985,

[7] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki: *Data-Oriented Transaction Execution*. PVLDB, 3(1), 2010.

[8] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki: *Scalability of write-ahead logging on multicore and multisocket hardware.* VLDB Journal 21(2), 2011.

[9] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling: *High-Performance Concurrency Control Mechanisms for Main-Memory Databases*. VLDB 2012.

[10] Antti-Pekka Liedes and Antoni Wolski, *SIREN: A Memory-Conserving, Snapshot-Consistent Checkpoint Algorithm for in-Memory Databases*, ICDE 2006.

[11] Antti-Pekka Liedes and Petri Soini, *Memory allocator for optimistic data access*, US Patent number 12/121,133, 2008.

[12] Antti-Pekka Liedes, *Bottom-up Optimistic Latching Method For Index Trees*, US Patent 20120221531, 2012.

[13] Kerttu Pollari-Malmi, Jarmo Ruuth and Eljas Soisalon-Soininen, *Concurrency Control for B-Trees with Differential Indices*, IDEAS 2000.

[14] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan: *No More Backstabbing... A Faithful Scheduling Policy for Multithreaded Programs*. PACT 2011.

[15] Vilho Raatikka and Antoni Wolski, *External Reference Entity Method For Link Failure Detection in Highly Available Systems*, International Workshop On Dependable Services and Systems (IWoDSS), 2010.

[16] Antoni Wolski and Vilho Raatikka, *Performance Measurement and Tuning of Hot-Standby Databases*, ISAS 2006, Lecture Notes in Computer Science, Volume 4328, 2006.

[17] Antoni Wolski and Kyösti Laiho, *Rolling Upgrades for Continuous Services*, ISAS 2004.

[18] Antoni Wolski, Sally Hartnell: *solidDB and Secrets of the Speed*, IBM Data Management (1), 2010.

The VoltDB Main Memory DBMS

Michael Stonebraker
stonebraker@csail.mit.edu
VoltDB, Inc.

Ariel Weisberg
aweisberg@voltdb.com
VoltDB, Inc.

Abstract

This paper describes the VoltDB DBMS as it exists in mid 2013. The focus is on design decisions, especially in concurrency control and high availability, as well as on the application areas where the system has found acceptance. Run-time performance numbers are also included.

1 Introduction

The purpose of VoltDB is to go radically faster than traditional relational DBMSs, such as MySQL, DB2 and SQL Server on a certain class of applications. These applications include traditional transaction processing (TP), as has existed for years. In addition, more recent TP applications include a variety of non-traditional use cases, such as maintaining the state of internet games and real-time ad placement on web pages. Also included are high velocity use cases whereby an incoming firehose of messages (either from humans or "the internet of things") must be processed and some sort of state maintained. High velocity applications include maintaining the overall risk in an electronic trading application and a variety of telco applications. Lastly, this collection includes use cases where some sort of state must be reliably maintained over repeated update, such as maintaining the directory of a distributed file system. In all cases, a database must be reliably maintained when updated by a firehose of transactions, interspersed with queries. For ease of exposition, we call the above class of applications, **ModernTP**, to distinguish it from the various subclasses which it encompasses.

The motivation for the design of VoltDB (and its research prototype predecessor, H-Store [4]) comes from the plummeting cost of main memory. Today, one can buy 1 Tbyte of main memory for perhaps $30,000, and deploy it readily as 128 Gbytes on each of 8 nodes in a computer network. Over time, 10 - 100 Tbyte databases will be candidates for main memory deployment. Hence, most (but not all) ModernTP applications are a candidate for main memory deployment, either now or in the future.

Toward the end of this decade it is possible that a new memory technology, for example Phase Change Memory (PCM) or Memristors, will become viable. Such a technology may well be more attractive than DRAM in main memory database applications, thereby further enlarging the reach of byte-addressible DBMSs.

If your data fits in main memory, then [3] presents a sobering assessment of the performance of traditional disk-based DBMSs on your application. Specifically, on TPC-C around 10% of the time is spent on useful work (i.e. actually executing transactions). The remaining 90% is consumed by four sources of overhead, namely buffer pool overhead, multi-threading overhead, record-level locking and an Aries-style [6] write-ahead log. The details of these measurements appear in [3] and their relative sizes vary with the exact transaction mix being

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

observed. For the purposes of this paper, we can just say each source of overhead is larger than the useful work in the transaction.

The conclusion to be drawn is that all four sources of overhead must be eliminated if one wishes to perform an order of magnitude better than traditional systems. A quick calculation shows that eliminating three of the four sources will result in only marginally better performance. This motivation drives the architecture of VoltDB.

In Sections 3-6 we discuss our solution for eliminating each source of overhead. Then in Section 7, we indicate the current engineering focus, followed in Section 8 by a discussion of early application areas where VoltDB has found acceptance, along with some performance numbers on VoltDB speed. We begin in Section 2 with some assumptions about ModernTP applications that have guided our design.

2 Assumptions

2.1 Workload

First, high availability (HA) is a requirement in all but the lowest end applications. If a node fails, then the system should seamlessly fail-over to a backup replica(s) of the data and keep going. As such single-node computer systems have no possibility of supporting HA, and VoltDB assumes a multi-node cluster architecture. In this environment, it makes sense to horizontally partition every table into ”shards” and deploy the shards to various nodes in the cluster. VoltDB supports hash (and range in the future) partitioning for shards and assumes every shard is replicated K +1 times, supporting the concept of K-safety.

Second, VoltDB strives to provide ACID properties both for concurrent transactions and for replicas. This is in contrast to many DBMSs, which provide some lesser guarantee. Moreover, so-called ”eventual consistency” provides no guarantee at all, unless the transaction workload is commutative. Even Google, long the champion of squishy guarantees, is coming around to an ACID point of view [8]. Hence, VoltDB is a pure ACID SQL system, deployed across a replicated, sharded database.

We assume that applications are a mix of transactions, containing both small/medium reads and small writes. Large reads are best performed against a downstream data warehouse, which is optimized for this purpose. In addition, the writes we see have the following composition:

(Vast majority) **Single-node transactions.** In this case, there is a database design that allocates the shards of each table to nodes in such a way that most transactions are local to a single node. Looking up a banking account balance or a purchase order is an example of a single-node transaction.

(Lesser number) **One-shot transactions.** In this case, the transaction consists of a collection of SQL statements, each of which can be executed in parallel at various nodes in a computer network. For example, moving $ X from account Y to account Z (without checking for an overdrawn condition) is a one-shot that will be executed at two nodes, one holding the shard of Y and the other the shard containing Z.

(Even less) **General transactions.** These are transactions that consist of multiple phases where each phase must be completed before the next phase begins. Moreover, one or more of the phases touches multiple nodes.

Lastly, VoltDB transactions are assumed to be deterministic. Hence, they cannot have code that, for example, branches on the current reading of a system clock. In addition, they cannot block, for example by trying to communicate with a human user or a remote site, external to the VoltDB system.

In our world, a transaction is a stored procedure that is registered with the system. Transactions are analyzed at compile time, categorized as above, and stored inside the VoltDB runtime system. A user executes a transaction by identifying the stored procedure to be called and supplying run-time parameters. Ad-hoc queries and updates are accepted and compiled on-the-fly into temporary stored procedures that are called in the same way.

2.2 LAN and WAN Behavior

Most VoltDB applications are deployed on single clusters connected by LAN technology. In such worlds, LAN network partitions are exceeding rare, and are dominated by node failures, application errors (errant logic corrupts data in the database) and DBA errors (for example, accidentally deleting data) as noted in [10]. The latter two will force a VoltDB cluster to stop. Hence, network partitions are not "the high pole in the tent" and providing HA in this case will not significantly improve overall availability. Therefore, VoltDB chooses consistency over availability during network partitions, consistent with our position of providing ACID wherever possible.

We do have users who are running replicas across a WAN between data centers. None of these users are willing to pay the cost (in transaction latency) of ensuring replica consistency across a WAN by performing synchronous replica update. Hence, VoltDB performs synchronous replication across LAN replicas and asynchronous replication across a WAN. As a result, it is possible for a WAN replica to lag its primary replica and to lose transactions on a WAN failure. Our users are happy with this design choice, and sometimes put resources into replicated networks to lower the probability of a WAN failure. Lastly, we see many cases of transient WAN failures. Hence, our users do not want to see automatic WAN failover as a result of transient errors. Instead, they want DBA-initiated WAN failover. A contrary opinion concerning synchronous WAN replication and automatic cutover is presented in [8].

3 Multi-Threaded Operation

VoltDB statically divides shared memory into "chunks" and statically allocates chunks to individual cores. These CPUs are not multi-threaded. Hence, there are no latches or critical sections to contend with in the VoltDB code. In this paper, a node refers to this pairing of non-shared main memory and a CPU. To improve load balance, we are in the process of supporting the movement of chunk boundaries. Notice that VoltDB is not sensitive to the race conditions prevalent in multi-threaded systems. In our opinion these are the cause of many concurrency control and recovery errors. As such, VoltDB has been relatively easy to debug as it consists of non-parallel code.

4 Main Memory Operation

VoltDB stores all data in main memory format. Of course, performance degrades if real memory is exhausted, and virtual memory starts to swap. It is crucial to provision VoltDB clusters with an appropriate amount of physical memory. All data is stored in main memory format and any pointers exist in virtual memory. There is no buffer pool and no buffer pool code.

5 Concurrency Control

VoltDB executes all operations in a deterministic order. Previous work in this area has typically used time as the ordering mechanism (timestamp order); however, VoltDB does not use a clock-based scheme. A single-node transaction is examined in the user-space VoltDB client library, where parameters are substituted to form a runnable transaction. The client library is aware of VoltDB sharding, so the transaction can be sent to a **node controller** at the correct node. This node controller serializes all single-node transactions and they are executed from beginning to end in this order without any blocking. Any application that consists entirely of single-node transactions will obtain maximum parallelism and, in theory, run all CPUs at 100% utilization, as long as the sharding achieves good load balance.

Other kinds of transactions are sent to a special **global controller**, which is responsible for deciding a serial order. The various SQL commands in such a transaction must be sent to the correct sites, where they are inserted into the single-node transaction stream at any convenient place. Hence, they are arbitrarily interleaved into the single-node transaction mix. A moment's reflection should convince the reader that the result is serializable. Various optimizations are implemented and others are planned for the classes of more general transactions.

It should be noted that our deterministic ordering scheme is one example of a class of deterministic concurrency control schemes. Other candidates are presented in [8, 9, 11].

6 Crash Recovery and High Availability (HA)

There are two aspects we need to discuss in this section. First, we discuss replication, fail-over and fail-back. Following that we consider global cluster failures, such as power failures.

6.1 Replication

High availability is achieved through replication. VoltDB implements a simple and elegant scheme. When a transaction is executed, it is first sent reliably to all replicas, which process the transaction in parallel. In other words, it is an active-active system. Since concurrency control is deterministic, each transaction will either succeed or fail at all nodes, and no additional synchronization is needed. In contrast, other popular concurrency control schemes, such as MVCC [1], do not have this deterministic property and must have some protocol to ensure that one gets the same outcome at each replica. Alternately, they would have to use an active-passive scheme that moves the log over the network, which we expect will offer much worse performance.

Nodes exchange "I am alive" messages. When a node is determined by its peers to be dead, it is disconnected from the VoltDB cluster and the remainder of the cluster continues operation. Up to K such node failures can be accommodated in this fashion. On the K + 1st error, the cluster stops.

When a node returns to operation, it executes the protocol in Section 6.2 to bring its state nearly current with the rest of the cluster. Then, the system catalogs are updated to reflect the newly added node and the tail of the transaction log (discussed in the next subsection) is executed at the newly operational node. Then, the cluster resumes normal operation. Users report that this failover-failback occurs without anybody being aware that it is happening.

6.2 Global Errors

We have a number of VoltDB customers running on off-premises shared servers (the cloud). Such servers usually have an uninterruptable power supplies (UPS). On-premises servers often do not have a UPS system. In such circumstances, a power failure or other cluster-wide error can cause all nodes in the cluster to stop, and the contents of main memory will be lost. Obviously, this is unacceptable behavior.

To deal with this error condition, VoltDB takes asynchronous, transaction-consistent checkpoints of the state of main memory. Specifically, when checkpoint-begin occurs, a local executor goes into a copy-on-write mode. Main memory is passed by the checkpoint code, and the results of all transactions before a specific **anchor** one are written to disk. Indexes are not checkpointed, but instead are rebuilt during recovery. A user can control the frequency of checkpoints and only the last one need be retained for subsequent recovery. The net result of a checkpoint is a disk image of main memory together with the anchor transaction.

In addition, VoltDB keeps a transaction log. For each transaction, we record the transaction identifier and its parameters. A group commit is used to reliably write this log to disk, before results are returned to the user.

When a global error occurs, the latest checkpoint is restored at each local node. A protocol is used to trade the tail of the log among nodes, so the set of committing transactions can be determined. Then, the transactions

since the anchor transaction are re-executed as conventional user transactions. The net result is a transaction consistent database, which is then free to accept new user transactions.

Our implementation of checkpointing and the transaction log degrades the run time performance of a VoltDB cluster by around 5%. In contrast, some of us implemented a main memory data logging scheme which pushed the effects of each transaction to disk. This technique is described in [5] and is a simplified and streamlined variation of Aries. In testing the run-time performance of the two recovery schemes it was found that the overall throughput of transaction logging was 3 times the throughput of the data logging scheme. Of course, recovery time was substantially longer with transaction logging. Since global errors are rare, it seems appropriate to optimize for run time performance. Also, a cluster is up and running much more often than it is recovering from a node failure. Also, single node failures can be recovered in the background, so again recovery time is less critical than performance at run time.

7 Current Development Focus

7.1 SQL

VoltDB engineering is focused on enlarging the subset of SQL that is supported. After all, the SQL specification is some 1500 pages long, and a startup obviously has to proceed in stages. Hence, every VoltDB release adds SQL features, based on customer requests.

7.2 On-Line Reprovisioning

Our customers, especially the ones offering some sort of web service, often experience dramatic load swings as well as substantial increases in load over time (the so-called hockey stick of success). Obviously, they wish to begin operation on K server nodes, and then add or subtract processing nodes, re-sharding the data as needed, all without incurring down time. On-line reprovisioning should be completely operational by the time this paper appears.

7.3 Integration with Downstream Repositories

As noted in Section 2, VoltDB is not architected to solve resource-intensive long-running data warehouse-style queries. Moreover, the storage requirements for long-term historical data repositories can be extreme. Hence, VoltDB is often ”upstream” from some sort of large storage system. Our customers request high performance parallel export to the major warehouse vendors, and we are well-along at satisfying this need.

8 Application Areas and Benchmarks

8.1 Early Application Areas

VoltDB found early acceptance in the web community, where it is used to maintain the state of internet games, leaderboards, and other applications which need high speed update of a shared state. In addition, it is widely used to execute real-time ad placement on web pages.

It has also found early adopters in aspects of real-time electronic trading, for example to keep track of the global position of the enterprise across multiple electronic trading desks. In addition, it is also used in real-time compliance applications.

Lastly, VoltDB has become widely used in embedded products in a variety of application areas, whereby an application needs to maintain a high velocity state of some sort. In all VoltDB has about 200 production users,

and is now finding application in a variety of markets, that are not regarded as early adopters. Hence, VoltDB should be considered as "having crossed the chasm" [7].

8.2 VoltDB Performance

We have been benchmarked on a variety of applications and include numbers in this section for a representative Voter benchmark, which obeys the application assumptions of Section 2. Voter uses the schema in Table 1, and has three transactions. The first one simulates a telephone vote tallying system, whereby each customer gets a collection of X votes, which he can allocate to contestants one-by-one until they are exhausted. The second transaction produces a heat map for a contestant for all states once a second, while the third produces a leaderboard once a second.

```
Tables:
    Contestants(c_number, c_name) -- replicated at all nodes
    Area_code(area_code, state)   -- replicated at all nodes
    Votes(phone_number, state, c_number) -- partitioned on phone_number
Materialized views:
    Votes_by_phone_number(phone_number, count)
    Votes_by_contestant_number_state(c_number, state, count)
```

Table 1: The Voter Schema

Referring to Table 1, the Vote transaction does a SQL lookup in the Contestants table to verify that the consumer is voting for a legitimate contestant. Then, it does a SQL lookup to the Votes_by phone_number materialized view to ensure that the customer has sufficient votes left to do his proposed operation. Third, the procedure looks up the state for the given area code from the Area_code table. Then, it does an insert into the Votes table. Finally, the transaction causes updates to the materialized views whereby appropriate counts are incremented. In all there are three selects, one insert and two updates. Since Votes is partitioned on phone number, each materialized view is partitioned the same way and the two read-only tables are replicated at all nodes, the transaction has 6 SQL commands and is single-noded. Finally, once per second the heatmap and leaderboard must be constructed. The tally of votes by contestant is a one-shot which aggregates the Votes_by_contestant_number_state materialized view. Similarly the heatmap is a one-shot constructed from the same materialized view. Performance on the Voter benchmark is measured by the number of votes per second that can be processed for the indicated schema, performing the one-shots every second. Details about Voter can be obtained from files in the directory: https://github.com/VoltDB/voltdb/blob/master/examples/voter/.

A five node dual quad-core SGI system (total of 40 cores) performs 640,000 Voter transactions/second. This is about one SQL command per core every 10 microseconds. Adding nodes to the configuration resulted in linear scalability. In all a 30 node system executed 3.4 million transactions/sec, and linear scalability was observed with a slope of 0.88. Details concerning these results can be found at www.sgi.com/pdfs/4238.pdf.

9 Summary

VoltDB is a main memory DBMS designed for ModernTP applications running on a cluster of nodes. It uses deterministic scheduling and an active-active replication strategy. As an example of a "NewSQL" DBMS, we expect it (and other products) will gradually take over the ModernTP market from legacy disk-oriented vendors because of dramatically superior performance.

References

[1] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI'12: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[3] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD Conference*, pages 981–992, 2008.

[4] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[5] N. Malviya. et. al. Recovery algorithms for in-memory OLTP databases. *(submitted for publication)*.

[6] C. Mohan, D. J. Haderle, B. G. L. 0001, H. Pirahesh, P. M. Schwarz, and P. M. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, pages 94–162, 1992.

[7] G. Moore. *Crossing the Chasm: Marketing and Selling Disruptive Products to Mainstream Customers*. Harper-Collins, 2002.

[8] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[9] A. Pavlo, E. P. C. Jones, and S. B. Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *PVLDB*, 5(2):85–96, 2011.

[10] M. Stonebraker. In search of database consistency. *Commun. ACM*, 53(10):8–9, Oct. 2010.

[11] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD Conference*, pages 1–12, 2012.

High-Performance Transaction Processing in SAP HANA

Juchang Lee[1], Michael Muehle[1], Norman May[1], Franz Faerber[1], Vishal Sikka[1],
Hasso Plattner[2], Jens Krueger[2], Martin Grund[3]
[1]SAP AG
[2]Hasso Plattner Insitute, Potsdam, Germany,
[3]eXascale Infolab, University of Fribourg, Switzerland

Abstract

Modern enterprise applications are currently undergoing a complete paradigm shift away from traditional transactional processing to combined analytical and transactional processing. This challenge of combining two opposing query types in a single database management system results in additional requirements for transaction management as well. In this paper, we discuss our approach to achieve high throughput for transactional query processing while allowing concurrent analytical queries. We present our approach to distributed snapshot isolation and optimized two-phase commit protocols.

1 Introduction

An efficient and holistic data management infrastructure is one of the key requirements for making the right decisions at an operational, tactical, and strategic level and is core to support all kinds of enterprise applications[12]. In contrast to traditional architectures of database systems, the SAP HANA database takes a different approach to provide support for a wide range of data management tasks. The system is organized in a main-memory centric fashion to reflect the shift within the memory hierarchy[2] and to consistently provide high performance without prohibitively slow disk interactions. Completely transparent for the application, data is organized along its life cycle either in column or row format, providing the best performance for different workload characteristics[11, 1]. Transactional workloads with a high update rate and point queries can be routed against a row store; analytical workloads with range scans over large datasets are supported by column oriented data structures. However, the DBA is free to choose an appropriate layout and to apply automated partitioning strategies as presented in [2]. In addition to a high scan performance over columns, the column-oriented representation offers an extremely high potential for compression making it possible to store even large datasets within the main memory of the database server. However, the main challenge is to support the new kind of mixed workload that requires a rethinking of the storage layer for modern database systems. In such a mixed workload for enterprise-scale applications the aggregated size of all tables will no longer fit on a single node, emphasizing the need for optimized distributed query processing. As a consequence, we have to enhance the traditional transactional query processing with workload specific information to achieve the best possible throughput. The overall goal of SAP HANA as a distributed in-memory mixed workload database is to exploit transactional locality as often as possible. In this paper, we present insights into the distributed transaction and session management in SAP

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

HANA that allow to combine transactional and analytical query processing inside a single DBMS. Therefore we will discuss our approach to distributed snapshot isolation and our optimizations to our in-memory optimized two-phase commit protocol.

2 Distributed Transaction Management

In contrast to other scale-out solutions like Hadoop, SAP HANA follows the traditional semantics of databases by providing full ACID support. In order to deliver on the promise of supporting both OLTP and OLAP-style query processing within a single platform, the SAP HANA database relaxes neither any consistency constraints nor any degree of atomicity or durability. To achieve this the SAP HANA database applies reduced locking with MVCC and logging schemes for distributed scenarios with some very specific optimizations like optimizing the two-phase commit protocol or providing sophisticated mechanisms for session management. Based on the deployment scheme for the given use case the framework provided by SAP HANA allows to minimize the cross-node cost for transaction processing in the first place. This allows to use for example one big server for transaction processing of the hot tables and use a multitude of smaller servers for analytical processing on the rest of the data partitions. In general, SAP HANA relies on Multi-Version-Concurrency-Control (MVCC) as the underlying concurrency control mechanism, the system provides distributed snapshot isolation and distributed locking to synchronize multiple writers. Therefore, the system implements a distributed locking scheme with a global deadlock detection mechanism avoiding a centralized lock server as a potential single point of failure. Based on this general description we will now explain our optimizations for handling distributed snapshot isolation and the optimized two-phase commit protocol for distributed transactions.

2.1 General Concurrency Control Mechanism

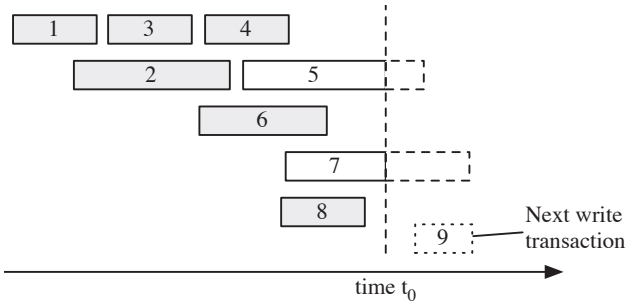
Isolation of concurrent transactions is enforced by a central transaction manager maintaining information about all write transactions and the *consistent view manager* deciding on visibility of records per table. A so-called *transaction token* is generated by the transaction manager for each transaction and encodes what transactions are open and committed at the point in time the transaction starts. The transaction token contains all information needed to construct the consistent view for a transaction or a statement. It is passed as an additional context information to all operations and engines that are involved in the execution of a statement.

For token generation, the transaction manager keeps track of the following information for all write transaction: (i) unique transaction IDs (TID), (ii) the state of each transaction, i.e., *open*, *aborted*, or *committed*, and (iii) once the transaction is committed, a commit ID (CID) (see Figure 1). While CIDs define the global commit order, TIDs identify a single transaction globally without any order. However, there are reasons to avoid storing the TID and CID per modified record. In the compressed column store, for example, the write sets of the transactions are directly stored in the modifiable delta buffer of the table. To achieve fine granular transaction control and high performance it becomes necessary to translate the information associated with the global CID into an representation that only uses TIDs, to avoid revisiting all modified records during the CID association at the end of the commit. The data consolidation is achieved using the transaction token. It contains the global reference to the last committed transaction at this point in time and the necessary information to translate this state into a set of predicates based on TIDs. The transaction token contains the following fields: `maxCID`, `minWriteTID`, `closedTIDs`, and the `TID`. Now, we will explain how the state information of the `maxCID` is translated into a set of `TID` filter predicates. The `minWriteTID` attribute of the token identifies *all* records that are visible to *all* transactions that are currently running hereby exploiting the assumption that the number of concurrently running transactions is limited and the order of the TIDs is similar to the order of the CIDs. This means that for all transactions T_j all records R with $TID_R < minWriteTID$ are visible. Since there can be committed transactions between `minWriteTID` and `TID` the attribute `closedTIDs`

identifies all subsequently closed transactions. Thus, the for the transaction T_j all records R are visible where $TID_R < minWriteTID \vee TID_R \in closedTIDs \vee TID_R = TID_{T_j}$. This expression identifies all TIDs that match a certain transactional state.

Rows inserted into the differential store by an open transaction are not immediately visible to any concurrent transaction. New and invalidated rows are announced to the consistent view manager as soon as the transaction commits. The consistent view manager keeps track of all added and invalidated rows to determine the visibility of records for a transaction. Using the TID expression from above and these two lists, all visible records are identified. Furthermore it is important to mention that for the column-store tables in HANA the write sets are not stored separately but directly in the differential buffer. If transactions are aborted this may result in a little overhead, but only until the merge process[8] recompresses the complete table and discards all invalid records.

To generalize, for every transaction T_j, the consistent view manager maintains two lists: one list with the rows added by T_j and a second list of row IDs invalidated by T_j. For a transaction T_k, we apply the previously mentioned predicate expression on this list of added and invalidated records. For a compact representation of change information, the consistent view manager consolidates the added row list and invalidated row lists of all transactions with a TID smaller than the *maxWriteTID*. This allows a fast consolidation of the first part of the predicate. For all other TIDs individual change information is kept requiring to scan the list of added or invalidated records.



TID	Status	CID
1	aborted	-
2	commited	2
3	commited	1
4	commited	3
5	open	-
6	commited	5
7	open	-
8	commited	4

Transaction Token	
nextWriteTID	9
Closed TIDs	6,8
maxCID	5
maxWriteTID	4

Figure 1: Data Consolidation with Transaction Token

2.2 Distributed Snapshot Isolation

For distributed snapshot isolation reducing the overhead of synchronizing transaction ID or commit timestamp across multiple servers belonging to a same transaction domain has been a challenging problem [10, 4, 5]. Therefore, the primary goal of all our optimizations is to increase the performance of short-running local-only transactions. The motivation for this case is that typically only long-running analytical queries will require data access to multiple nodes, where short-running queries will be mostly single-node queries. This observation is close in spirit to fixed partitioning approaches recently presented in [6, 7]. To avoid costly short-running multi-node queries the DBA is supported with partitioning tools for automatic partitioning. For long-running analytical queries spanning across multiple nodes, the incurred overhead incurred for communicating this transactional information will be comparable small in comparison to the overall execution time of the query.

In isolation mode Read-Committed the boundary of the lifetime of the MVCC snapshot is the query itself. Thus, the isolation requirements can be determined at compile time to identify which partitions of the tables are accessed. The entire transaction context that is needed for a query to execute is cached on a local node. For the *Repeatable Read* or *Serializable* isolation level, the transaction token can be reused for the queries belonging to the same transaction, which means that its communication cost with the coordinator node is less important. Whenever a transaction (in a transaction-level snapshot isolation mode) or a statement (in a statement-level

snapshot isolation mode) starts, it copies the current transaction token into its context (called snapshot token) for visibility detection.

In a distributed environment, the transaction protocol defines that for every started transaction a worker node should access the transaction coordinator to retrieve its snapshot transaction token. This could cause (1) a throughput bottleneck at the transaction coordinator and (2) additional network delay to the worker-side local transactions. To avoid these situations we use three techniques: First, enabling local read-only transactions to run without accessing the global coordinator, second, enabling local read or write transactions to run without accessing the global coordinator, and third, using *Write-TID-Buffering* to enable multi-node write transactions to run with only a single access to the global coordinator.

Optimization for worker-node local read transactions or statements In general, every update transaction accesses the transaction coordinator to access and update the global transaction token. However, read-only statements can reuse the cached local transaction token ensuring a consistent view on the data, as it contains an upper limit for visible records. This cached local transaction token is refreshed in two cases: (i) by the transaction token of an update transaction when it commits on the node, or (ii) by the transaction token of a *global statement* when it comes in to (or started at) the node. If the statement did not need to access any other node, it can just finish with the cached transaction token, i.e., without any access to the transaction coordinator. If it detects that the statement should also be executed in another node, however, it is promoted to a *global statement* type, after which the current statement is retried with the global transaction token obtained from the coordinator. Single-node read-only statements/ transactions do not need to access the transaction coordinator at all, since they reuse the locally-cached transaction token. This is significant to avoid the performance bottleneck at the coordinator and to reduce the performance overhead of single-node statements (or transactions).

Optimization for worker-side local write transactions Each node manages its own local transaction token independent of the global transaction token. In case of a single-node update transaction the transaction token of the node is only refreshed locally. In contrast to traditional approaches, each database record in the delta partition has two TID (or Transaction ID) columns for MVCC version filtering: one for global TID and another for local TID. In case of a local transaction the local TID is updated, in case of a global transaction, it reads either global or local TID (reads a global TID if there is a value in its global TID column; otherwise, reads a local TID) and updates both global and local TIDs. As a consequence, the global transactions carry two snapshot transaction tokens: one for global transaction token and second one for the current worker node's local transaction token. In the log record both global and local TIDs are also recorded if it is a global transaction. On recovery a local transaction's commit can be decided by its own local commit log record. Again, only global transactions are required to check with the global coordinator.

Optimization for multi-node write transactions A multi-node write transaction needs a global write TID as all multi-node transactions do. In a HANA scale-out system one of the server nodes becomes the transaction coordinator which manages the distributed transaction and coordinates two phase commit. Executing a distributed transaction involves multiple network communications between the coordinator node and worker nodes. Each write transaction is assigned a globally unique write transaction ID. A worker-node-first transaction, one that starts at a worker node first, should be assigned a TID from the coordinator node as well, which causes an additional network communication. Such a communication might significantly affect the performance of distributed transaction execution. This extra network communication is eliminated to improve the worker-node-first write transaction performance. We solve this problem by buffering such global write TIDs in a worker node. Buffering TIDs will not incur ordering conflicts as the TID is only used as a unique transaction identifier and has no explicit ordering criterion. When a request for a TID assignment is made the very first time, the coordinator node returns a range of TIDs which gets buffered in the worker node. The next transaction which needs a TID gets it from

the local buffer thus eliminating the extra network communication with the coordinator node. For short-running transactions this can increase the throughput by $\approx 30\%$. The key performance parameters are the size of the range and the time-frame when unused TIDs are discarded. These parameters can be calibrated and adjusted at runtime to achieve the best throughput. Discarded TIDs are not reused to avoid unnecessary communications with the coordinator for consolidation of TIDs. It is important to mention that the above described techniques allow improving the overall transactional performance without losing or mitigating transactional consistency.

2.3 Optimizing Two-Phase Commit Protocol

Two-phase commit (2PC) protocol is widely used to ensure atomicity of distributed multi-node update transactions. In HANA, we use a series of optimization techniques that attempt to reduce the network and log I/O delays during a two-phase commit to increase throughput.

Early commit acknowledgment after the first commit phase Our first optimization is to return the commit acknowledgment early after the first commit phase[9, 3] as shown in Figure 2. Right after the first commit phase and the commit log is written to the disk, the commit acknowledgment can be returned to the client. And then, the second commit phase can be done asynchronously.

For this optimization, we consider three main points: Writing the commit log entries on the worker nodes can be done asynchronously. During crash recovery, some committed transactions will be classified as in-doubt transactions, which are resolved as committed by checking the transaction's status in the coordinator. If transaction tokens are cached asynchronously on the worker nodes, the data is visible by a transaction but not by the next (local-only) transaction in the same session. This situation can be detected by storing the last transaction token information for each session at the client side. And then, until the second commit phase of the previous transaction is done, the next query can be stalled. If transactional locks are released after sending a commit acknowledgment to the client, a "false lock conflict" may arise by the next transaction in the same session. This situation can however be detected and resolved by the worker and coordinator. If this is detected, the transaction waits for a short time period until the commit notification arrives to the worker node.

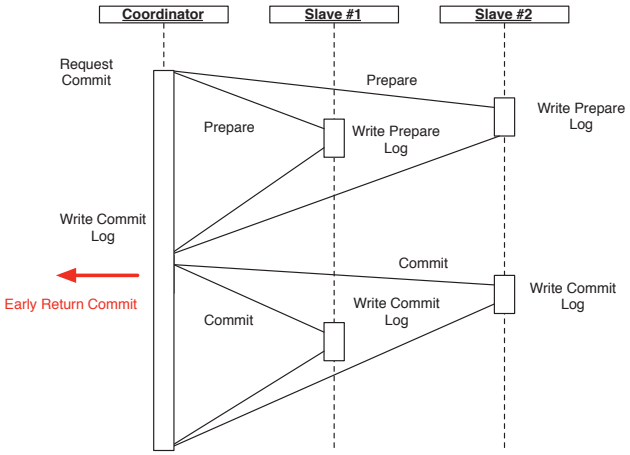


Figure 2: Returning commit ack early after first commit phase

Skipping writes of prepare logs The second optimization is to eliminate log I/Os for writing prepare-commit log entries. In a typical 2-phase-commit the prepare-commit log entry is used to ensure that the transaction's previous update logs are written to disk and to identify in-doubt transactions at the recovery time. Writing the transaction's previous update logs to disk can be ensured by only comparing the transaction-last-LSN (log sequence number) with the log-disk-last-LSN. If the log-disk-last-LSN is larger or equal than the transaction-last-LSN, it means that the transaction's update logs are already flushed to disk. If we do not write the prepare-commit log entry, we handle all the uncommitted transactions at recovery time as in-doubt transactions. Their commit status can be decided by checking with the transaction coordinator. The goal is to trade transactional throughput with recovery time, as the size of in-doubt transaction list can increase, but reduces the run-time overhead for log I/Os.

Group two-phase commit protocol This is a similar idea to the one described earlier in this Section, but instead of sending commit requests to the coordinator node individually (i.e., one for each write transaction), we can group multiple concurrent commit requests into one and send it to the coordinator node in one shot. Also, when the coordinator node multicasts a "prepare-commit" request to multiple-related worker nodes of a transaction, we can group multiple "prepare-commit" requests of multiple concurrent transactions which will go to the same worker node. By this optimization we can increase throughput of concurrent transactions. Two-phase commit itself cannot be avoided fundamentally for ensuring global atomicity to multi-node write transactions. However, by combination of the optimizations described in this subsection, we can reduce its overhead significantly.

3 Summary

In this paper, we presented details about the transaction processing system in SAP HANA. The main challenge for our system is to be able to support two opposing workloads in a single database system. Therefore, we carefully optimized our system to be able to separate transactional from analytical queries and to guarantee the best possible throughput for transactional queries.

References

[1] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[2] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[3] Ramesh Gupta, Jayant R. Haritsa, and Krithi Ramamritham. Revisiting Commit Processing in Distributed Database Systems. In *SIGMOD Conference*, pages 486–497. ACM Press, 1997.

[4] H. V. Jagadish, Inderpal Singh Mumick, and Michael Rabinovich. Scalable Versioning in Distributed Databases with Commuting Updates. In *ICDE*, pages 520–531. IEEE Computer Society, 1997.

[5] H. V. Jagadish, Inderpal Singh Mumick, and Michael Rabinovich. Asynchronous Version Advancement in a Distributed Three-Version Database. In *ICDE*, pages 424–435. IEEE Computer Society, 1998.

[6] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD Conference*, pages 603–614. ACM, 2010.

[7] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[8] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB*, 5(1):61–72, 2011.

[9] C. Mohan, Bruce G. Lindsay, and Ron Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.*, 11(4):378–396, 1986.

[10] C. Mohan, Hamid Pirahesh, and Raymond A. Lorie. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. In *SIGMOD Conference*, pages 124–133. ACM Press, 1992.

[11] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD Conference*, pages 1–2. ACM, 2009.

[12] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *SIGMOD Conference*, pages 731–742. ACM, 2012.

The Hekaton Memory-Optimized OLTP Engine

Per-Ake Larson
palarson@microsoft.com

Mike Zwilling
mikezw@microsoft.com

Kevin Farlee
kfarlee@microsoft.com

Abstract

Hekaton is a new OLTP engine optimized for memory resident data and fully integrated into SQL Server; a database can contain both regular disk-based tables and in-memory tables. In-memory (a.k.a.Hekaton) tables are fully durable and accessed using standard T-SQL. A query can reference both Hekaton tables and regular tables and a transaction can update data in both types of tables. T-SQL stored procedures that reference only Hekaton tables are compiled into machine code for further performance improvements. To allow for high concurrency the engine uses latch-free data structures and optimistic, multi-version concurrency control. This paper gives an overview of the design of the Hekaton engine and reports some initial results.

1 Introduction

SQL Server, like other major database management systems, was designed assuming that main memory is expensive and data resides on disk. This is no longer the case; today a server with 32 cores and 1TB of memory costs as little as $50K. The majority of OLTP databases fit entirely in 1TB and even the largest OLTP databases can keep the active working set in memory. Recognizing this trend SQL Server several years ago began building a database engine optimized for large main memories and many-core CPUs. The new engine, code named Hekaton, is targeted for OLTP workloads.

Hekaton has a number of features that sets it apart from other main-memory database engines. Most importantly, the Hekaton engine is integrated into SQL Server; it is not a separate product. A database can contain both Hekaton in-memory tables and regular disk-based tables. This approach offers customers major benefits compared with a separate system. First, customers avoid the hassle and expense of another DBMS. Second, only the most performance-critical tables need to be in main memory; other tables can remain regular SQL Server tables. Third, conversion can be done gradually, one table and one stored procedure at a time.

Memory optimized tables are managed by Hekaton and stored entirely in main memory. A Hekaton table can have several indexes which can be hash indexes or range indexes. Hekaton tables are durable and transactional, though non-durable tables are also supported. Hekaton tables are queried and updated using T-SQL in the same way as regular SQL Server tables. A query can reference both Hekaton tables and regular tables and a transaction can update both types of tables. Furthermore, a T-SQL stored procedure that references only Hekaton tables can be compiled into native machine code for further performance gains. This is by far the fastest way to query and modify data in Hekaton tables. Hekaton is designed for high levels of concurrency but does not rely on partitioning to achieve this. Any thread can access any row in a table without acquiring latches or locks.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

The engine uses latch-free (lock-free) data structures to avoid physical interference among threads and a new optimistic, multi-version concurrency control technique to reduce interference among transactions.

2 Design Overview

This section provides a brief overview of the design of the Hekaton engine and the guiding principles behind the design decisions. The interested reader can find more details in [2].

2.1 No partitioning

HyPer [5], Dora [8], H-store [18], and VoltDB [22] are recent systems designed for OLTP workloads and memory resident data. They partition the database by core and give one core exclusive access to a partition. Partitioning works great but only if the workload is also partitionable. If the workload partitions poorly so that transactions on average touch several partitions, performance deteriorates considerably.

Hekaton does not partition the database and any thread can access any part of the database. We prototyped and carefully evaluated a partitioned engine but came to the conclusion that a partitioned approach is not sufficiently robust to handle the wide variety of workloads customers expect SQL Server to handle.

2.2 Tables and indexes optimized for main memory

Database systems traditionally use disk-oriented storage structures where records are stored on disk pages that are brought into memory as needed. This requires a complex buffer pool where a page must be protected by latching before it can be accessed. The resulting overhead is high: a simple key lookup in a B-tree index may require thousands of instructions even when all pages are in memory.

Hekaton indexes and other data structures are designed and optimized for memory-resident data and the on-disk representation is completely different than their in-memory representation and in fact indexes are not materialized on external storage. Records are referenced directly by physical pointers, not indirectly by a logical pointer such as a page/row ID. Record pointers are stable; a record is never moved after it has been created. A table can have multiple indexes, any combination of hash indexes and range indexes. A hash index is simply an array where each entry is the head of a linked list through records. Range indexes are implemented as Bw-trees [7] which is novel version of B-trees optimized for main-memory and high concurrency.

2.3 Latch-free data structures

Machines with large numbers of CPU cores are increasingly common and core counts are expected to continue increasing. Achieving good scaling on many-core machines is critical for high throughput. Scalability suffers when the systems has shared memory locations that are updated at high rate such as latches and spinlocks and highly contended resources such as the lock manager, the tail of the transaction log, or the last page of a B-tree index.

All Hekaton's internal data structures, for example, memory allocators, hash and range indexes, and the transaction map, are entirely latch-free (lock-free) [3]. There are no latches or spinlocks on any performance-critical paths in the system.

2.4 Multi-versioning

Database systems traditionally maintain only a single version of a record and all updates are applied in place with transaction locks used to synchronize access which can limit scalability. A writer prevents all accesses to the record until the transaction commits while readers prevent the record from being updated. Because readers

block writers, even a few long read-only transactions (to prepare a report, for example) can drastically reduce update throughput.

To avoid readers competing with writers, Hekaton. like many other database systems, uses multi-versioning where an update creates a completely new version of a record. Once created, the user payload of a version is never modified. The lifetime of a version is defined by two timestamps, a begin timestamp and an end timestamp and different versions of the same record have non-overlapping lifetimes. A transaction specifies a logical read time for all reads, typically the transaction's begin time, and only versions whose lifetime overlaps the read time are visible to the transaction. Consequently, the transaction will read at most one version of a record.

Multi-versioning improves scalability because readers no longer block writers. (Writers may still conflict with writers though.) Read-only transactions have little effect on update activity; they simply read older versions of records as needed. In particular, long read-only transactions no longer reduce update transaction throughput. Furthermore, once a reader has a memory pointer to a record version that is visible to them, it can trust that the version will never change or move.

However, multi-version is not without cost. Updating a record in place is faster than creating a new version and obsolete versions no longer needed must be cleaned out. Hekaton's algorithms for cleaning out obsolete versions and reclaiming memory is described in more detail in [2].

2.5 Concurrency control

Hekaton uses a new optimistic multi-version concurrency control algorithm to provide transaction isolation; there are no locks and no lock table. The algorithm is described in detail in [6] but we provide a brief summary here.

A transaction optimistically assumes that it is running isolated (versioning makes it appear so), but before a transaction can commit it must validate that it indeed has not been interfered with by another transaction. This validation begins once a transaction has completed its normal processing and wants to commit. Any number of transactions can perform validation in parallel. The extent of validation required depends on the transaction's isolation level. Read-only transactions (regardless of isolation level) and transactions running under read-committed or snapshot isolation require no validation at all. Write-write conflicts are detected immediately when a transaction attempts to update a version and result in the transaction rolling back.

Transactions running under repeatable read or serializable isolation require validation before commit. During validation a transaction T checks whether the following two properties hold.

1. **Read stability.** If T read some version V1 during its processing, we must ensure that V1 is still the version visible to T as of the end of the transaction. This is implemented by validating that V1 has not been updated before T commits. Any update will have modified V1's end timestamp so all that is required is to check V1's timestamps. To enable this, T retains a pointer to every version that it has read.

2. **Phantom avoidance.** For serializable transactions we must also ensure that the transaction's scans would not return additional new versions. This is implemented by rescanning to check for new versions before commit. To enable this, a serializable transaction keeps tracks of all its index scans and retains enough information to be able to repeat each scan.

To avoid blocking during normal processing, Hekaton allows a limited form of speculative reads: a transaction T1 can read a version created by another transaction T2 that is still in validation. T1 cannot commit or return results to the user until T2 has committed. To enforce this, T1 takes a commit dependency on T2 which is released by T2 as soon as it has committed. If T2 aborts, it instructs T1 to abort.

The combination of multi-versioning, optimistic concurrency control with speculative reads, and latch-free data structures results in a system where a thread never blocks or stall during normal processing of a transaction.

2.6 Durability

Transaction durability is ensured by logging and checkpointing records to external storage; index operations are not logged. During recovery Hekaton tables and their indexes are rebuilt entirely from the latest checkpoint and logs.

A transaction that successfully passes validation is ready to commit. At this point it writes to the log all new versions that it has created and keys of all versions it has deleted. This is done in a single write and if the write succeeds, the transaction is irrevocably committed. Hekaton puts much less pressure on the log than regular SQL Server because nothing is written to the log until commit. Aborted transactions are not logged so aborting a transaction is cheap. Hekaton can spread the log over multiple log devices because commit ordering in Hekaton is determined by transactions' end timestamps, not by log ordering.

Checkpoints are computed by processing the log, not by scanning tables. The checkpoint algorithms use only sequential reads and writes to avoid the high overhead and latency of random IO. More details about logging and checkpointing can be found in [2].

2.7 Compilation to native code

SQL Server uses interpreter based execution mechanisms in the same ways as most traditional DBMSs. This provides great flexibility but at a high cost: even a simple transaction performing a few lookups may require several hundred thousand instructions. Hekaton maximizes run time performance by converting stored procedures written in T-SQL into customized, highly efficient machine code. The generated code contains exactly what is needed to execute the request, nothing more. As many decisions as possible are made at compile time to reduce runtime overhead. For example, all data types are known at compile time allowing the generation of efficient code. Hekaton's compilation process is described in [2].

3 High-Level Architecture

As illustrated in Figure 1, Hekaton consists of three major components.

- The Hekaton storage engine manages user data and indexes. It provides transactional operations on tables of records, hash and range indexes on the tables, and base mechanisms for storage, checkpointing, recovery and high-availability.
- The Hekaton compiler takes an abstract tree representation of a T-SQL stored procedure, including the queries within it, plus table and index metadata and compiles the procedure into native code designed to execute against tables and indexes man-aged by the Hekaton storage engine.
- The Hekaton runtime system provides integration with SQL Server resources and serves as a common library of additional functionality needed by compiled stored procedures.

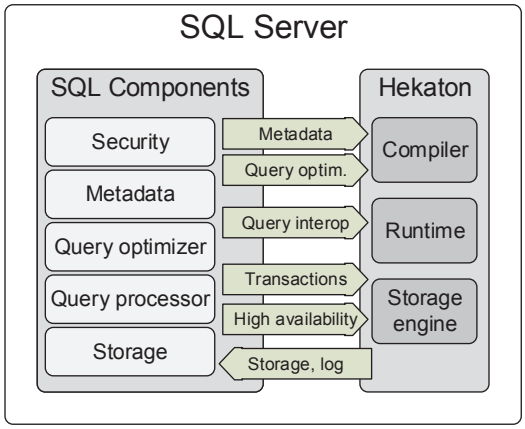


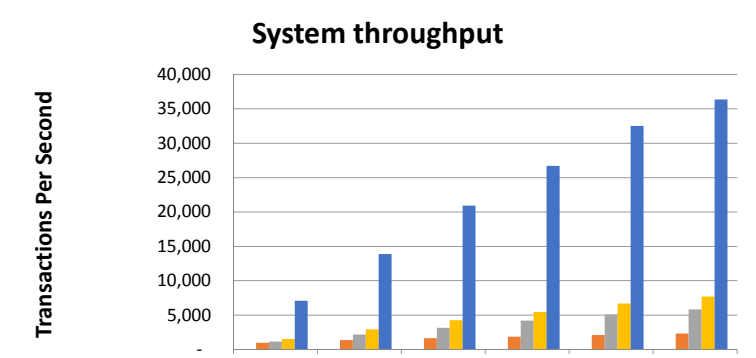
Figure 1: Hekaton components and integration with SQL Server

Hekaton leverages a number of services already available in SQL Server. The main integration points are illustrated in Figure 1.

- *Metadata:* Metadata about Hekaton tables, indexes, etc. is stored in the regular SQL Server catalog. Users view and manage them using exactly the same tools as regular tables and indexes.
- *Query optimization:* Queries embedded in compiled stored procedures are optimized using the regular SQL Server optimizer. The Hekaton compiler compiles the query plan into native code.
- *Query interop:* Hekaton provides query operators for accessing data in Hekaton tables that can be used in interpreted SQL Server query plans. There is also an operator for inserting, deleting, and updating data in Hekaton tables.
- *Transactions:* A regular SQL Server transaction can access and update data both in regular tables and Hekaton tables. Commits and aborts are fully coordinated across the two engines.
- *High availability:* Hekaton is integrated with AlwaysOn, SQL Server's high availability feature. Hekaton tables in a database fail over in the same way as other tables.
- *Storage, log:* Hekaton logs its updates to the regular SQL Server transaction log. It uses SQL Server file streams for storing checkpoints. Hekaton tables are automatically recovered in parallel while the rest of the database is recovered.

4 Performance Illustration

We now report results from an experiment that illustrates the performance and scalability improvements offered by Hekaton compared with regular SQL Server. The experiment emulates an order entry system for, say, a large online retailer. The load on the system is highly variable and during peak periods throughput is limited by lock and latch contention. The system is simply not able to take advantage of additional processor cores so throughput levels off or even decreases. When SQL Server customers experience scalability limitations, the root cause is frequently lock or latch contention. Hekaton is designed to eliminate lock and latch contention, allowing it to continue to scale with the number of processor cores.



Number of cores	2	4	6	8	10	12
SQL with contention	984	1,363	1,645	1,876	2,118	2,312
SQL without contention	1,153	2,157	3,161	4,211	5,093	5,834
Interop	1,518	2,936	4,273	5,459	6,701	7,709
Native	7,078	13,892	20,919	26,721	32,507	36,375

Figure 2: Improved throughput and scalability offered by Hekaton

The main activity is on a table SalesOrderDetails that stores data about each item ordered. The table has a unique index on the primary key which is a clustered B-tree index if the table is a regular SQL Server table and a hash index if it is Hekaton table. The workload in the experiment consists of 60 input streams, each a mix of 50% update transactions and 50% read-only transactions. Each update transaction acquires a unique sequence number, which is used as the order number, and then inserts 100 rows in the SalesOrderDetails table. A read-only transaction retrieves the order details for the latest order.

The experiment was run on a machine with 2 sockets, 12 cores (Xeon X5650, 2.67GHz), 144GB of memory, and Gigabit Ethernet network cards. External storage consisted of four 64GB Intel SSDs for data and three

80GB Fusion-IO SSDs for logs. There was sufficient memory to hold the entire database using either regular or Hekaton tables.

Figure 2 shows the throughput as the number of cores used varies. The regular SQL Server engine shows limited scalability as we increase the number of cores used. Going from 2 cores to 12 cores throughput increases from 984 to 2,312 transactions per second, only 2.3X. Latch contention limits the CPU utilization to only 40% for more than 6 cores.

Converting the table to a Hekaton table and accessing it through interop already improves throughput to 7,709 transactions per second for 12 cores, a 3.3X increase over plain SQL Server. Accessing the table through compiled stored procedures improves throughput further to 36,375 transactions per second at 12 cores which is 15.7X improvement over regular SQL Server.

The Hekaton engine shows excellent scaling. Going from 2 to 12 cores, throughput improves by 5.1X for the interop case (1,518 to 7,709 transactions per second). If the stored procedures are compiled, throughput also improves by 5.1X (7,078 to 36,375 transactions per second).

We also investigated the performance of the regular SQL Server engine when there was no contention. We partitioned the database and rewrote the stored procedure so that different transactions did not interfere with each other. The results are shown in the row labeled "SQL with no contention". Removing contention increased maximum throughput to 5,834 transaction/sec which is still lower than the throughput achieved through interop. Without contention scaling improved to 5.1X going from 2 cores to 12 cores, compared with 2.3X with contention.

5 Initial Customer Experiences

Bwin is the largest regulated online gaming company in the world, and their success depends on positive customer experiences. They use SQL Server extensively and were keen to try out an early preview of Hekaton. Prior to using Hekaton, their online gaming systems were handling about 15,000 requests per second, a huge number for most companies. However, Bwin needed to be agile and stay at ahead of the competition so they wanted access to the latest technology speed. Using Hekaton Bwin were hoping they could at least double the number of transactions. They were pretty amazed to see that the fastest tests so far have scaled to 250,000 requests per second.

Edgenet is a leading provider of product data management, product listings, and retail selling solutions, aggregating and curating product information from various suppliers, and correlating it to availability and inventory information from the stores of many retailers to present end users with a comprehensive shopping experience. The project prototyped was an inventory tracking system, which receives batch updates from each retailer with updates for all products in each of their stores. The baseline SQL measurement hit maximum performance with 2 threads, at 7,450 rows/second. Hekaton scaled smoothly up to 50 threads, yielding 126,665 rows per second.

SBI Liquidity Market(SBILM), located in Japan, provides financial trading infrastructure for tenant companies. SBILM's current volume of trading is $1.75 Trillion, around twice that of the Japanese government's budget. They are investing in their next-generation foreign exchange trading platform. The system explored aggregate trading data to calculate prices for currency pairs. The timeliness of this data is critical to profitability. When trading greatly increases, their current system yields 4 second aggregation time. Their goal is less than one second, which equates to 4000 TPS. In lab tests, they achieved a maximum of 2812 TPS with traditional SQL tables, and 5313 with Hekaton, significantly exceeding their goals.

6 Conclusion

Hekaton is a new main-memory engine under development a Microsoft. Hekaton is fully integrated into SQL Server, which allows customers to gradually convert their most performance-critical tables and applications to

take advantage of the performance improvements offered by Hekaton. Hekaton achieves high performance and scalability by using very efficient latch-free data structures, multi-versioning, optimistic concurrency control, and by compiling T-SQL stored procedure into efficient machine code. Transaction durability is ensured by logging and checkpointing to external storage. High availability and transparent failover is provided by integration with SQL Server's AlwaysOn feature. The Hekaton engine is capable of delivering more than an order of magnitude improvement in efficiency and scalability with minimal and incremental changes to user applications or tools.

7 Acknowledgements

This project owes its success to the hard work and dedication of many people – too many to name everyone here. Notably, a strong collaboration between people in the Microsoft Research Database Group, the Microsoft Jim Gray Systems Lab, and the Microsoft SQL Server Development team has powered this project since its inception.

References

[1] *VoltDB*. http://voltdb.com.

[2] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *Sigmod*, page (to appear), 2013.

[3] K. Fraser and T. L. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

[4] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[5] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[6] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[7] J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. In *ICDE*, pages 3012–313, 2013.

[8] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

Transaction Processing in the Hybrid OLTP&OLAP Main-Memory Database System HyPer

Alfons Kemper Thomas Neumann Jan Finis Florian Funke
Viktor Leis Henrik Mühe Tobias Mühlbauer Wolf Rödiger
TU München, Faculty of Informatics
firstname.lastname@in.tum.de

Abstract

Two emerging hardware trends have re-initiated the development of in-core database systems: ever increasing main-memory capacities and vast multi-core parallel processing power. Main-memory capacities of several TB allow to retain all transactional data of even the largest applications in-memory on one (or a few) servers. The vast computational power in combination with low data management overhead yields unprecedented transaction performance which allows to push transaction processing (away from application servers) into the database server and still "leaves room" for additional query processing directly on the transactional data. Thereby, the often postulated goal of real-time business intelligence, where decision makers have access to the latest version of the transactional state, becomes feasible. In this paper we will survey the HyPerScript transaction programming language, the main-memory indexing technique ART, which is decisive for high transaction processing performance, and HyPer's transaction management that allows heterogeneous workloads consisting of short pre-canned transactions, OLAP-style queries, and long interactive transactions.

1 Introduction

Main-memory or in-core/in-memory database systems have been around since the 1980s (e.g., TimesTen), but only recently has DRAM become inexpensive enough to render large enterprise applications possible on solely main-memory-resident data. This has initiated industrial interest in main-memory databases, e.g., SAP's HANA [3] or Microsoft's Hekaton [8]. Along with ever growing main-memory capacities comes a dramatic increase in compute power through multi-core parallelism. This vast performance increase renders many redundancy-based optimization techniques like materialized views obsolete, which were designed to avoid scanning large fragments of a database in an interactive query. The abundance of compute power finally allows to push data-intensive applications directly into the database server as opposed to the currently employed multi-tier architectures where large amounts of data have to be transferred to the application servers. Retaining all transactional data in-core is even possible for the largest companies, as a simple "back of the envelope" calculation reveals: Amazon.com generated a revenue of US$60 billion in 2012. Assuming an average item price of $15 and that each order line incurs stored data of about 54 bytes – as specified for the TPC-C-benchmark that models such a

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

retailer – we derive a total data volume of less than 1/4 TB per year for the order lines, which is the dominating repository in such a sales application. This estimate neither includes other data (customer and product data) which increases the volume nor the possibility to compress the data to decrease the volume. Nevertheless it is safe to assume that the yearly transactional sales data can be fit into the main memory of a large scale server with a few TB capacity. To unleash the immense computing power of such a multi-core server, a radical reengineering of database technology is required in several areas: low-overhead data representation, compiled instead of interpreted code, high-performance indexing and low-cost transaction synchronization are addressed here. Such a low-overhead database on a multi-core server would not even be saturated by the transaction (Tx) processing as a simple calculation reveals: On average, Amazon.com has only 127 sales Tx per second (in peak times they report a few thousand), while a main-memory database system like HyPer achieves around 100,000 Tx per second in the TPC-C benchmark. Thus, besides the mission critical transaction processing there is room for additional OLAP-style query processing – if they don't interfere with each other. HyPer allows this via a highly efficient virtual memory snapshotting mechanism. These virtual memory snapshots are created frequently and shield the complex OLAP query processing entirely from the mission-critical OLTP processing without any kind of (software) concurrency control. These snapshots are also leveraged for tentative execution of long transactions whose effect are then, after validation, applied to the main database as a short install transaction.

2 Transaction Scripting and Compilation

Transactions are implemented in HyPerScript, a SQL-based programming language. The SQL query language used for OLAP-style queries thereby is a subset of HyPerScript. For illustration purposes, we show the *complete* implementation of the well-known *newOrder* transaction of the TPC-C benchmark [14].

```
create procedure newOrder (w_id integer not null, d_id integer not null, c_id integer not null,
      table positions(line_number integer not null, supware integer not null,
                      itemid integer not null, qty integer not null),
      datetime timestamp not null)  // note the TABLE-valued parameter above
{  select w_tax from warehouse w where w.w_id=w_id; // w_tax value used later
   select c_discount from customer c // c_discount used in orderline insert
       where c_w_id=w_id and c_d_id=d_id and c.c_id=c_id;
   select d_next_o_id as o_id,d_tax from district d // get the next o_id
       where d_w_id=w_id and d.d_id=d_id;
   update district set d_next_o_id=o_id+1 // increment the next o_id
       where d_w_id=w_id and district.d_id=d_id;

   select count(*) as cnt from positions; // how many items are ordered
   select case when count(*)=0 then 1 else 0 end as all_local
       from positions where supware<>w_id;
   insert into "order" values (o_id,d_id,w_id,c_id,datetime,0,cnt,all_local);
   insert into neworder values (o_id,d_id,w_id); // insert reference to order

   update stock
   set s_quantity=case when s_quantity>qty then s_quantity-qty else s_quantity+91-qty end,
       s_remote_cnt=s_remote_cnt+case when supware<>w_id then 1 else 0 end,
       s_order_cnt=s_order_cnt+case when supware=w_id then 1 else 0 end
   from positions where s_w_id=supware and s_i_id=itemid;

   insert into orderline // insert all the order positions
      select o_id,d_id,w_id,line_number,itemid,supware,null,qty,
             qty*i_price*(1.0+w_tax+d_tax)*(1.0-c_discount),
             case d_id when 1 then s_dist_01 when 2 then s_dist_02 when 3 then s_dist_03
                       when 4 ... when 9 then s_dist_09 when 10 then s_dist_10 end
      from positions, item, stock
      where itemid=i_id and s_w_id=supware and s_i_id=itemid
   returning count(*) as inserted; // how many were inserted?

   if (inserted<cnt) rollback; // not all ==> invalid item ==> abort
};
```

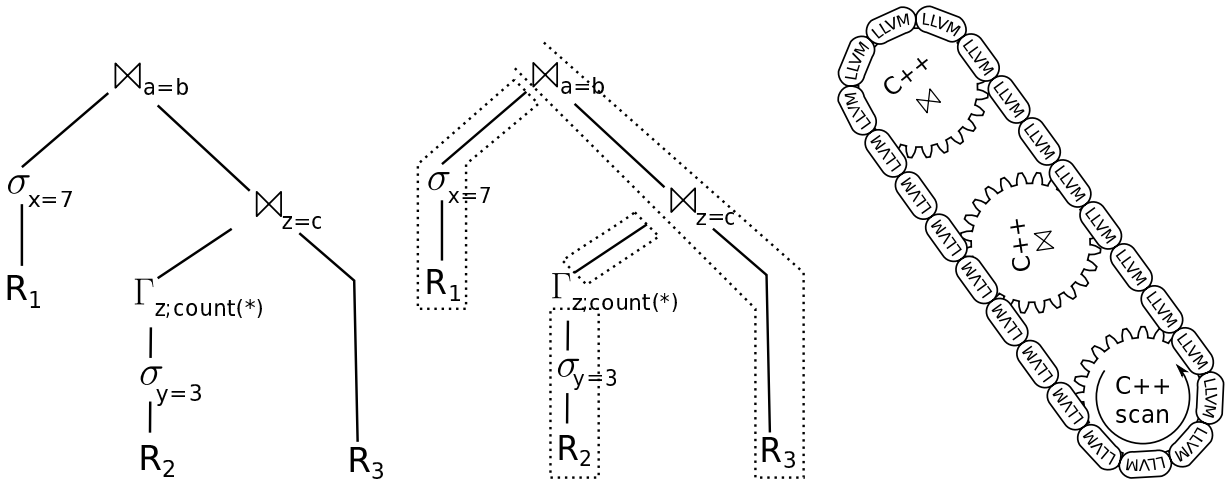


Figure 1: Compiling SQL queries into LLVM Code

The declarative HyPerScript language exhibits the following uncommon constructs that are present in the above script:

Table parameter. Frequently, a stored procedure is invoked for an entire collection of tuples, each consisting of several attribute values. Flattening the collection would result in a large number of parameters; a more elegant solution is provided by HyPerScript's table parameter that allows to pass an entire table – as demonstrated by the *positions* table in the *newOrder* script.

Reusing query results. In HyPerScript a query result can be reused in subsequent statements by referring to a prior assigned variable. For example, the first query of our *newOrder* script determines the tax rate *w_tax* of the particular warehouse. This *w_tax* value is later used in the insert statement that populates the *orderline* table with the item's price including the applicable tax.

Using such a declarative scripting language has a number of advantages: (1) the declarative nature allows for much more elegant and succinct code than in imperative languages – cf. the complete *newOrder* script; (2) the SQL statements can be optimized and executed like regular queries in the same query engine; (3) declarative HyPerScripts are much more amenable for security analysis than imperative programs. Thus, using the declarative HyPerScript language allows us to safely run compiled transactions within the database server process without any costly inter-procedural communication and context switching.

Stored procedures as well as regular (stand-alone) SQL queries are compiled into LLVM code [13]. LLVM constitutes a machine independent assembly language that is then further compiled and optimized for the particular server machine. The left-hand side of Fig. 1 shows a logical algebra plan for a three-way join including some selections and an (early) aggregation. Instead of the recently propagated vector-wise processing technique [2], HyPer's query engine relies on a data-centric execution model that exploits pipelining as much as possible. For this purpose, the query evaluation plan (QEP) is segmented into pipelines that end at pipeline breakers. As shown in the middle of Fig. 1, our example query has four such pipelines: the leftmost pipeline ends at the hash-table build of the upper join; R_2-tuples are scanned and selected up to the (hash) *groupify*. From there on, they are forwarded to the hash-table build of the lower join. Finally, R_3-tuples are propelled "in one go" via the first join all the way up to the upper join and produce output tuples by probing the hash-join table.

Thus, once a data object is accessed it is processed as much as possible. This way the query processing achieves maximal data locality as an object is transferred to a machine register as few times as possible.

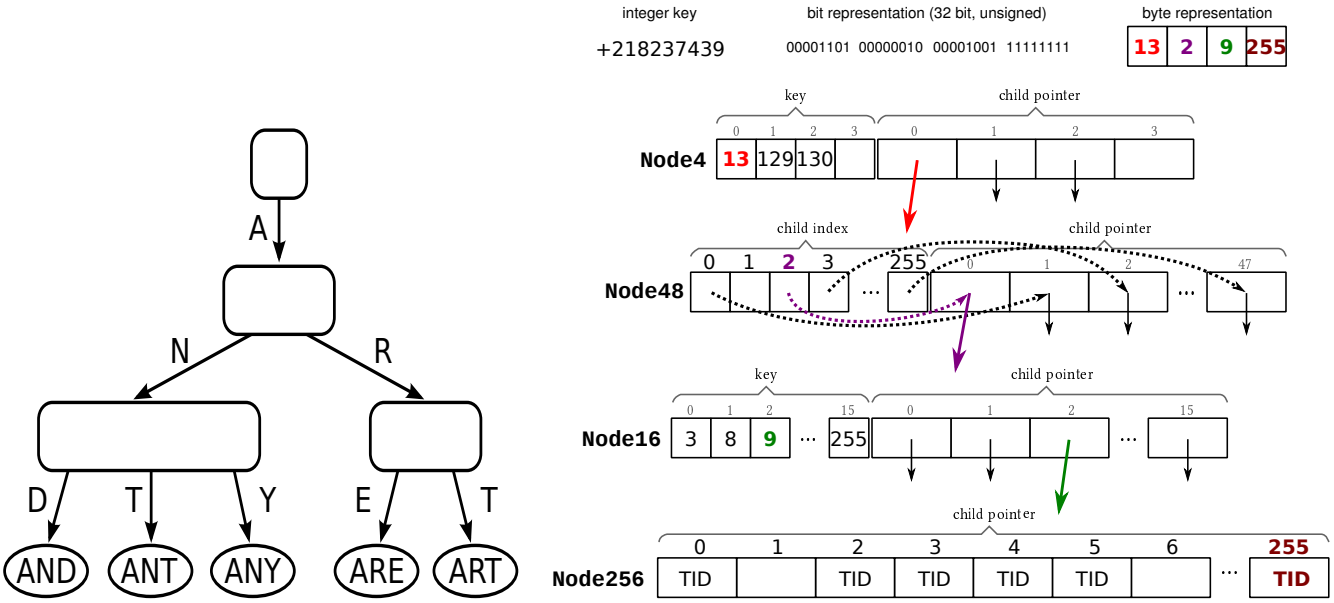


Figure 2: The Adaptive Radix Tree ART: (left) general idea of different sized nodes, (right) a sample path for the key 218237439 traversing all four node types

The right-hand side of Fig. 1 illustrates the translation of queries into corresponding LLVM/C++ code. The typical algorithms of a query engine, such as scans, hash-join table building and probing, grouping and aggregation, are pre-implemented in the high-level programming language C++. These C++ building blocks are "glued together" by generated LLVM – as (metaphorically) sketched for the rightmost pipeline of the example QEP. The generated LLVM code (the chain) makes the C++ query operators (the cog wheels) work together in evaluating an entire pipeline.

3 ARTful Indexing in Main-Memory Databases

The efficiency of transaction processing largely depends on which index structures are used, as exemplified by the first three *select*-statements of the *newOrder* implementation. In main-memory, dictionary-like data structures supporting insert, update, and delete are often implemented as hash tables or comparison-based trees (e.g. self-balancing binary trees or B-trees). Hashing is usually much faster than a tree as it offers constant lookup time in contrast to the logarithmic behavior of comparison-based trees. The advantage of trees is that the data is stored in sorted order, which enables additional operations like range scan, minimum, maximum, and prefix lookup.

The radix tree, also known as trie, prefix tree, or digital search tree, is another dictionary-like data structure. In contrast to comparison-based structures, which compare opaque key values using a comparison function, radix trees directly use the binary representation of the key. Although radix trees are often introduced as a data structure for storing character strings (cf., left hand side of Fig. 2), they can be used to store any data type by considering values as strings of bits or bytes.

The complexity of radix trees for insert, lookup, and delete is $O(k)$ where k is the length of the key. The access time is *independent* of the number of elements stored. Besides the length of the key, the height of a radix tree depends on the number of children each node has. For example, a radix tree with a fanout of 256 that stores 32 bit integers has a height of 4.

So far radix trees suffered from space underutilization problems as typically an array of 256 pointers was

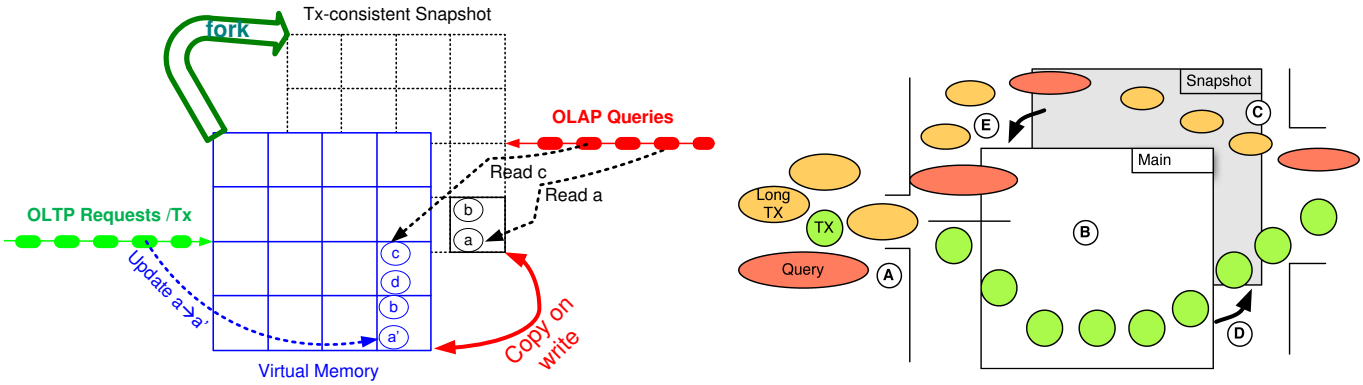


Figure 3: Virtual Memory Snapshots: (left) to separate OLTP & OLAP; (right) OLAP queries and long Tx are delegated to the snapshot (A), short transactions are executed on the main (B), long Tx are optimistically executed on the snapshot (C) and re-executed after validation as an apply transaction (E) on the main

allocated for each node – even though some nodes might have a very low fan-out compared to others. Therefore, we developed the Adaptive Radix Tree (ART), which uses four different node types that can handle up to (i) 4, (ii) 16, (iii) 48, and (iv) 256 entries. Thereby, a good space utilization of ART-trees is guaranteed while still being able to achieve a maximum height of k for k-byte keys. That is, 32 bit integers are indexed with a tree of height 4, 64 bit integers require height 8. These adaptive nodes are exemplified by the sample path for the 32-bit key 218237439 consisting of the the 4 byte-chunks 13&2&9&255 on the right-hand side of Fig. 2. This path starts at the root, which happens to be of type *Node4*, and then covers the three other node types. The structural representation of these node types varies, as illustrated in the figure. In designing the node structure the trade-off between space utilization and intra-node search performance was taken into account.

Besides adaptive nodes, we employ two well-known techniques that reduce both the tree height and the space consumption. First, we build the tree lazily, i.e., any path that leads to a single leaf is truncated. As a consequence, the leaf is stored higher up in the tree. Only when another key with a shared prefix is inserted, an additional inner node is created as a "goalpost" between the two leaves. The second technique, path compression, removes each common path (e.g., "http://" when indexing URLs) and instead stores the path as an additional prefix of the following inner node. This avoids cache-inefficient chains of one-way nodes. When indexing long keys, lazy expansion and path compression are very effective in reducing the tree height. Additionally, the two optimizations allow to bound the worst-case space consumption per key/value pair to 52 bytes – even for arbitrarily long keys [9].

4 Isolation of Long Running Transactions

Originally, HyPer focussed on the execution of short, pre-canned transactions *and* OLAP-style, read-only queries which are executed on a snapshot of the data that is generated using the UNIX *fork*-mechanism [7, 10]. The snapshot is kept consistent by the memory management unit (MMU) via the copy-on-write procedure that will (automatically) detect shared pages and copy them prior to any update, as illustrated on the left of Fig. 3.

In this architecture the OLTP process "owns" the database and periodically (e.g., in the order of seconds or minutes) forks an OLAP process. This OLAP process constitutes a fresh transaction-consistent snapshot of the database. Thereby, the OLAP processing is completely separated from the OLTP processing without any (software) concurrency control mechanism. Whenever an object (such as a in the figure) on a shared page is modified, the MMU automatically creates a page copy via the copy-on-write mechanism.

The OLTP transactions are executed serially on (partitions of) the transactional database state – as pioneered

by H-Store/VoltDB [6, 15]. Even though this yields unprecedented performance, serial execution is restricted to short and pre-canned transactions.

This makes main-memory database systems using serial execution unsuitable for "ill-natured" transactions like long-running OLAP-style queries or transactions querying external data – even if they occur rarely in the workload. In our approach [11], which we refer to as *tentative execution*, the coexistence of short and long-running transactions in main-memory database systems does not require recommissioning traditional concurrency control techniques like two phase locking. Instead, the key idea is to tentatively execute long-running transactions on a transaction-consistent database snapshot, that exists for OLAP queries anyway. Thereby, a long transaction is converted into a short "apply transaction" which is then, after validation, re-executed on the main database, as illustrated on the right of Fig. 3.

Different mechanisms can be used to separate the workload into short and long transactions. A simple approach is limiting the runtime or number of tuples each transaction is allowed to use before it has to finish. When a transaction exceeds this allotment – which can vary depending on the transaction's complexity or the number of partitions it accesses – it is rolled back using the undo log and re-executed using tentative execution.

Our approach is optimistic in that it queues and then executes transactions on a consistent snapshot of the database. This is advantageous as no concurrency control is required to execute short and apply transactions. Similar to other optimistic execution concepts a validation phase is required which makes some form of monitoring necessary.

During the apply phase, the effects of the transaction as performed on the snapshot are validated on the main database and then applied. This is done by injecting an "apply transaction" into the serial execution queue of the main database. As opposed to the transaction that ran on the snapshot, the apply transaction only needs to validate the work done on the snapshot, not re-execute the original transaction in its entirety or wait for external resources.

Specifically, we distinguish between two cases: When *serializability* is requested, all reads have to be validated. To achieve this, it is checked whether or not the read performed on the snapshot is identical to what would have been read on the main database. Depending on the monitoring granularity, the action performed here ranges from actually performing the read a second time on the main database to comparing version counters between snapshot and main.

When *snapshot isolation* is used, the apply transaction ensures that none of the tuples written on the snapshot have changed in the main database, therefore guaranteeing that the write sets of both the tentative transaction as well as all transactions that have committed on the main database after the snapshot was created are disjoint. This is achieved by either comparing the current tuple values to those saved in the log or by checking that all version counters for written tuples are equal both during the execution on the snapshot and on the main database.

5 Conclusion and Ongoing Work

The high performance of HyPer is due to several design decisions: (1) compiling SQL queries and HyPerScript transactions into LLVM assembler code instead of interpreted execution; (2) the new radix tree indexing which offers performance similar to hash tables while allowing range scans; (3) the virtual memory snapshot mechanism which entirely shields OLTP from OLAP without *any* (software controlled) synchronization; (4) lock-free partition-serial execution of short transactions while (5) long transactions are executed optimistically on the snapshot and then applied to the main database like a short transaction. Currently we are working on parallelizing the query execution [1], compacting the working set of the transactional database [5], supporting versioned hierarchical data [4], bulk data loading, and scaling out to processor clusters [12].

References

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main-memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[2] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *DaMoN*, 2005.

[3] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[4] J. Finis, R. Brunel, A. Kemper, T. Neumann, F. Färber, and N. May. DeltaNI: An efficient labeling scheme for versioned hierarchical data. In *SIGMOD*, 2013.

[5] F. Funke, A. Kemper, and T. Neumann. Compacting transactional data in hybrid OLTP&OLAP databases. *PVLDB*, 5(11):1424–1435, 2012.

[6] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main-memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[7] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main-memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[8] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[9] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful indexing for main-memory databases. In *ICDE*, 2013.

[10] H. Mühe, A. Kemper, and T. Neumann. How to efficiently snapshot transactional data: hardware or software controlled? In *DaMoN*, pages 17–26, 2011.

[11] H. Mühe, A. Kemper, and T. Neumann. Executing long-running transactions in synchronization-free main memory database systems. In *CIDR*, 2013.

[12] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. ScyPer: Elastic OLAP throughput on transactional data. In *Workshop on Data analytics in the Cloud (DanaC)*, 2013.

[13] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 2011.

[14] Transaction Processing Performance Council. TPC-C specification. `www.tpc.org/tpcc/spec/TPC-C\_v5-11.pdf`, 2010.

[15] VoltDB. Technical Overview. `http://www.voltdb.com`, March 2010.

Modularity and Scalability in Calvin

Alexander Thomson
Google[‡]
agt@google.com

Daniel J. Abadi
Yale University
dna@cs.yale.edu

Abstract

Calvin is a transaction scheduling and replication management layer for distributed storage systems. By first writing transaction requests to a durable, replicated log, and then using a concurrency control mechanism that emulates a deterministic serial execution of the log's transaction requests, Calvin supports strongly consistent replication and fully ACID distributed transactions while incurring significantly lower inter-partition transaction coordination costs than traditional distributed database systems.

Furthermore, Calvin's declarative specification of target concurrency-control behavior allows system components to avoid interacting with actual transaction scheduling mechanisms—whereas in traditional DBMSs, the analogous components often have to explicitly observe concurrency control modules' (highly nondeterministic) procedural behaviors in order to function correctly.

1 Introduction

The task of building parallel and distributed systems designed for *embarrassingly parallel* computational problems is relatively straightforward. Data-parallel tasks with reasonably sparse and coarse-grained dependencies between computational tasks—such as many analytics applications—fall nicely into this category. In particular, parallelizing a workload across many machines is easiest when the amount of real-time coordination required between machines is lowest.

The general problem of online transaction processing, however, does *not* fall into this category. OLTP systems generally guarantee isolation between transactions using row-level locks, and any transaction that incurs synchronous communication between multiple machines (e.g. by performing a remote read or running a distributed commit protocol) must hold its locks for the full duration of this communication. As more transactions span multiple machines, lock contention becomes a bigger barrier to achieving high transactional throughput.

For systems that synchronously replicate transaction effects at commit time via Paxos [11] (e.g. Spanner, Megastore, Spinnaker), the problem is further exacerbated, since transactions' contention footprints[1] extend for the full Paxos replication latency [8, 1, 14].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]This material is supported by the National Science Foundation under Grant Number IIS-1249722.

[‡]Work done while the author was at Yale.

[1]The "contention footprint" of a transaction is the period of time during which it may limit concurrent execution of conflicting transactions. For systems that use locking, this corresponds directly to the time spent holding locks. In optimistic systems like Megastore, it is the time from when the transaction begins executing until the validation phase is complete, since other commits during this time window may cause the transaction to abort.

One approach to overcoming this limitation is to reduce transactional support. For example, Microsoft's SQL Azure [4], Google's Bigtable [6], and Oracle's NoSQL Database [17] limit each transaction to accessing records within a single small data partition. For certain "embarrassingly partitionable" OLTP applications, solutions that provide limited transactional support are useful. For many applications, however, this approach is a poor fit: if a transaction ever spans data on multiple partitions, the developer is forced to split it into multiple separate transactions—and therefore inherits the burden of reasoning about data placement, atomicity, and isolation.

Calvin takes a different approach to ameliorating this scalability bottleneck: it reduces the amount of coordination required while executing distributed transactions. Calvin rearranges the transaction execution pipeline so that all distributed agreement that is not directly data-dependent (i.e. all inter-partition communication except for serving remote reads) occurs in efficient batch operations *before* transactions begin acquiring locks and executing.

Specifically, Calvin divides the transaction execution pipeline into three stages—logging, scheduling, and execution. When a client submits a request to Calvin to execute a transaction, this transaction *request* (as opposed to ARIES-style physical actions) is immediately appended to a durable log, before any actual execution begins. Calvin's scheduling mechanism then processes this request log, deciding when each transaction should be executed in a way that maintains an invariant slightly stronger than serializable isolation: transaction execution may be parallelized but must be equivalent to a serial execution (a) in the order that transaction requests were logged and (b) that does not non-deterministically cause transactions to abort. Then, once a transaction begins executing, it is disallowed from performing any non-deterministic behaviors[2].

Calvin's processing of a series of transaction requests is therefore equivalent to a deterministic serial execution in log-specified order. As a result, the log of transaction requests itself serves as an ultimate "source of truth" about the state of a database. Even if all other information about a the database were discarded, its state at any point in history could be recovered by deterministically replaying the operation log[3].

This architecture provides several advantages over traditional (nondeterministic) database system designs:

- **Near-linear scalability.** Since all distributed agreement (both between replicas and between partitions within a replica) needed to commit a transaction occurs *before* Calvin executes the transaction, no distributed commit protocols are required, ameliorating the primary barrier to scaling ACID systems. We have demonstrated this scalability with a Calvin deployment that executed half a million TPC-C transactions per second[4] while partitioning data across one hundred commodity EC2 machines [21].
- **Strongly consistent replication.** Multiple replicas of a Calvin deployment remain perfectly consistent as long as they use the same operation request log to drive transaction execution. Once a transaction is written to the log, no further inter-replica coordination is required to synchronize replicas' states. This allows strongly consistent replication—even across geographically separated data centers—at no cost to transaction throughput [21].
- **Highly modular system architecture.** By explicitly describing the emulated serial execution in the form of the operation request log, Calvin provides other system components with a *declarative* specification of the concurrency control mechanisms' behavior. Components that traditionally interact closely with the transaction scheduling system in standard nondeterministic database systems (e.g. logging, replication, storage) are thus completely decoupled from Calvin's concurrency control implementation. Standard interfaces for each Calvin component make it easy to swap in different implementations of each component to quickly prototype systems with different properties.
- **Main-memory OLTP performance over disk-resident data.** When Calvin begins processing a transaction request by adding it to the log, it may also perform a cursory analysis of the transaction request and send a

[2]Calls to functions such as GetTime() and Random() are intercepted by Calvin, and replaced by deterministic functions that use the time at which the client submitted the transaction, rather than the local system time of the server executing the transaction.

[3]Calvin uses this property (along with frequent checkpointing) for durability and disaster recovery.

[4]Note that TPC-C New Order transactions frequently access records spanning multiple data partitions *and* incur high lock contention.

prefetch "hint" to each storage backend component that the transaction will access once it begins executing. The storage engine can therefore begin bringing the transaction's read- and write-sets into memory even before the request is appended to the log, so that all relevant data will be memory-resident once the transaction begins executing.

2 Calvin's Modular Implementation

Database management systems are notoriously monolithic pieces of software, largely because they use complex concurrency control mechanisms to orchestrate the simultaneous execution of many transactions [9]. Many attempts have been made—with varying success—to build clean interfaces between various components, decoupling transaction coordination, buffer pool management, logging/recovery mechanisms, data storage structures, replication coordination, query optimization, and other processes from one another [2, 5, 7, 13, 12, 16].

We believe that the fundamental difficulties in unbundling database components lies in the way concurrency control protocols are traditionally described. Besides being highly non-deterministic, concurrency control algorithms are usually framed (and specified and implemented) in a very *procedural* way. This means that system components must often explicitly observe internal state of the concurrency control module to interact with it correctly. These internal dependencies (particularly for logging and recovery) become extremely apparent in modular systems that are otherwise successful at separating database system components [13, 16].

Whereas serializable isolation requires equivalence to *some* serial execution of transactions (but is agnostic as to what particular order is emulated), Calvin's determinism invariant restricts concurrency control mechanisms to maintain equivalence to a specific, pre-determined order. This design decision was originally motivated by the need to reduce coordination costs between participants in a transaction, but we found that establishing an a priori transaction ordering is also useful as a *declarative* specification of concurrency control behavior.

In Calvin, database system components that traditionally interact closely with the concurrency control manager can instead gain the same information simply by reading from the (immutable) transaction request log.

Thus, in addition to reducing coordination costs between instantiations of the same component on different machines, Calvin's log simplified the coordination between components. This section gives an overview of the three independent parts of every Calvin deployment—the log, scheduler, and storage backend—and their individual interfaces and subcomponents.

2.1 Log

A Calvin deployment appends new transaction *requests* to a global log, which is then readable by other system components. Implementations of the log interface may vary from very simple to quite sophisticated:

- **Local log.** The simplest possible implementation of a log component appends new transaction requests to a logfile on the local filesystem to which other components have read-only access. This implementation is neither fault-tolerant nor scalable, but it works well for single-server storage system deployments.
- **Sharded, Paxos-replicated log.** At the opposite extreme from the local log is a scalable, fault-tolerant log implementation that we designed for extremely high throughput and strongly consistent WAN replication. Here, a group of front-end servers collect client requests into batches. Each batch is assigned a globally unique ID and written to an independent, asynchronously replicated block storage service such as Voldemort or Cassandra [19, 10]. Once a batch of transaction requests is durable on enough replicas, its GUID is appended to a Paxos "MetaLog". To achieve essentially unbounded throughput scalability, Paxos proposers may also group many request batch GUIDs into each proposal. Readers can then reassemble the global transaction request log by concatenating batches in the order that their GUIDs appear in the Paxos MetaLog.

2.1.1 Log-forwarding

Distributed log implementations may (a) forward the read- and write-sets of new (not-yet committed) transaction requests to the storage backends that they will access (so that the data can be prefetched into memory *before* transaction execution begins) and (b) analyze a committed transaction request and, if it can determine which schedulers will need to process the request, actively forwards the request to those specific schedulers (otherwise it forwards it to all schedulers). Our implementation provides a subcomponent that can be plugged into any actual log implementation to automatically handle these forwarding tasks.

2.1.2 Recovery Manager

Recovery managers in database systems are notoriously cross-dependent with concurrency control managers and data storage backends. Popular recovery protocols such as ARIES log every change to index data structures (including, for example, the internal details of each B+ tree page split), which means that they must understand what inconsistent states could arise in every index data structure update. Furthermore, recovery managers commonly rely on direct knowledge of record and page identifiers in the storage layer in order to generate log records, and may store their own data structures (e.g. LSNs) inside the data pages themselves.

Calvin leverages its deterministic execution invariant to replace physical logging with logical logging. Recovery is performed by loading database state from a recent checkpoint, then deterministically replaying all transactions in the log after this point, which will bring the recovering machine to the same state as any non-crashed replica. Calvin's recovery manager is therefore very simple and entirely agnostic to implementation details of the log, scheduler, and storage backend—so long as they respect Calvin's determinism invariant.

2.2 Separating Transaction Scheduling from Data Storage

In traditional database systems, actively executing transactions request locks immediately before each read and write operation. There are thus direct calls to the concurrency control code from within the access methods and storage manager. Calvin's concurrency control mechanisms diverge from this design: a Scheduler component examines a transaction before it begins executing and decides when it is safe to execute the *whole* transaction, then hands the transaction request off to the storage backend for execution with no additional oversight. The storage backend therefore does not need to have any knowledge of the concurrency control mechanism or implementation. Once a transaction begins running, the storage backend can be sure that it can process it to completion without worrying about concurrent transactions accessing the same data.

Since the storage manager does not make any direct calls to the concurrency control manager, it becomes straightforward to plug any data store implementation into Calvin, from simple key-value stores to more sophisticated NoSQL stores such as LevelDB, to full relational stores (Section 4 describes some of the storage backends that we have implemented over the course of creating various Calvin configurations). However, each storage backend must provide enough information prior to transactional execution in order for the scheduler to make a well-informed decision about when each transaction can safely (from a concurrency control perspective) execute. This is done by building a wrapper around every storage backend that implements two methods:

- **`RUN(action)`** receives a command in a language that the data store understands (e.g. `get(k)` or `put(k,v)` for a key-value store, or a SQL command for a relational store) and executes it on the underlying data store.
- **`ANALYZE(action)`** takes the same command, but instead of executing it, returns its *read- and write-sets*—collections of logical objects that will be read or created/updated when `RUN` is called on the command. These objects can be individual entities (such as key-value pairs or relational tuples) or ranges of entities. The scheduler leverages the `ANALYZE` method to determine which transactions can be run concurrently by the storage backend. In some cases, the `ANALYZE` method can derive the read and write set of the command using static analysis. In other cases, it may have to perform a 'dry run' of the command (at only one replica,

with no isolation) to see what data items it accesses. Then, when the same command is eventually `RUN`, the storage backend must double-check that the initial observation of what would be read or written to was still accurate. (Since the `RUN` method is called in deterministic execution mode, all replicas will perform this same double-check step and arrive at the same result.)

This wrapper around the storage backend enables the clean separation of transaction scheduling from storage management. This approach deviates significantly from previous attempts to separate database systems' transactional components from their data components [13, 12, 16] in that Calvin does not separate transaction *execution* from the data component at all—it only separates transaction *scheduling*.

Just as the storage backend is pluggable in Calvin, so is the transaction scheduler. Our prototype currently supports three different schedulers that use transactions' read-/write-sets in different ways:

- **Serial execution à la H-Store/VoltDB [18].** This scheduler simply ignores the read and write set information about each transaction and never allows two transactions to be sent to the storage backend at the same time. However, if data is partitioned across different storage backends, it allows different transactions to be sent to different storage backends at the same time.
- **Deterministic locking.** This scheduler uses a standard lock manager to lock the logical entities that are received from calling the ANALYZE method. If the lock manager determines that the logical entities of two transactions do not conflict, they can by executed by the storage backend concurrently. In order to avoid nondeterministic behavior, the ANALYZE method is called for each transaction in the order that the transactions appear in the transaction log, and the lock manager locks entities in this order. This ensures concurrent execution equivalent to the serial transaction order in the log, and also makes deadlock impossible (and the associated nondeterministic transaction aborts) [20, 21].
- **Very lightweight locking (VLL).** VLL is similar to the deterministic locking scheduler described above, in that it requests locks in the order of the transactional log, and is therefore deterministic and deadlock free. However, it uses per-data-item semaphores instead of a hash table in order to implement the lock manager. We originally proposed storing these semaphores alongside the data items that they control [15] for improved cache locality. However, this violates our modular implementation goal, so Calvin supports two versions of VLL—one which co-locates semaphores with data, and a more modular one in which the scheduler tracks all semaphores in a separate data structure.

Despite major differences between the internals of these transaction scheduling protocols, each implements a common interface and adheres to Calvin's deterministic scheduling invariant, so switching between them is as simple as changing a command line flag with which the Calvin binary is invoked. A single Calvin deployment could even use different schedulers for different replicas or even partitions within the same replica.

3 Transactional Interface

Clients of a Calvin system deployment may invoke transactions using three distinct mechanisms:

- **Built-in Stored Procedures.** Any storage backend API call can be invoked as a stored procedure with transactional semantics. For example, we implemented the TPC-C transactions that we benchmarked in [21, 15] as built-in API calls to a TPC-C-specific storage backend implementation. This is the transaction execution method that yields the best performance. However, while we made every effort to make it easy to extend or re-implement Calvin's storage backend components, it is by far the most development-intensive way for an application to run complex multi-operation transactions.
- **Lua Snippet Transactions.** Alternatively Clients can submit transactions that bundle together multiple storage API calls using client-specified logic as implemented in Lua code. The client simply constructs a Lua code snippet and sends it as a string to Calvin, which performs some analysis to make sure the code is well-behaved. Thereafter, it can be scheduled and executed as if it were a single stored procedure.

- **Optimistic Client-side Transactions.** Calvin also allows client-side transaction execution in which an OCC validation phase is executed as a deterministic stored procedure at commit time. Here, a client application starts a transaction session, performs any set of storage API calls (writes are buffered locally), and then submits a commit request for the session. Upon receiving the commit request, Calvin checks version timestamps for each record read and written by the transaction. If all of them precede the timestamp at which the client's session started, then this validation phase succeeds and the buffered writes are applied—otherwise the transaction "aborts"[5] and the client must start it again using a new session.

We expect that most application developers will choose to mostly use optimistic client-side transactions while building new applications. Specific performance-sensitive transactions can then be replaced with Lua snippets or built-in stored procedures as needed.

4 Beyond Main-Memory OLTP

Although we originally developed Calvin as a transaction scheduling and replication layer explicitly for main-memory OLTP database systems—and with specific log, scheduler, and storage implementations in mind—we have found it very easy to create Calvin configurations with a broad array of characteristics simply by swapping out implementations of various components.

4.1 CalvinDB

The distributed OLTP database system described in [21]—which we now refer to as CalvinDB—provided a basic CRUD interface to a memory-resident key-value store. Our early prefetching experiments used a very rudimentary disk-based backend that performed poorly. Newer versions are backed by a LevelDB-based store and provide stronger prefetching support, so Calvin can reliably process transactional workloads that sometimes access disk-resident data with main-memory-only throughput performance.

In addition, we have extended CalvinDB's interface to support secondary indexes and a subset of SQL for declarative data manipulation[6]. We found that Calvin's deterministic semantics makes it surprisingly painless to implement atomic database *schema* changes with no system or application downtime whatsoever—a challenge that is frequently a source of headache for architects of distributed database systems.

4.2 CalvinFS

Scalable metadata management and WAN replication for distributed file systems are hot research topics in the systems community due to the ever-increasing scalability and availability requirements of modern applications.

We therefore created CalvinFS, a distributed file system that supports strongly consistent WAN replication. CalvinFS stores data blocks that make up file contents in an easy-to-scale, non-transactional block store, and uses Calvin for transactional storage of file *metadata*. This turned out to be an extremely good fit for Calvin, because real-time metadata updates for distributed file systems in many ways resembles a *not*-embarrassingly-partitionable OLTP workload in two important ways:

- File systems traditionally support linearizable updates—which strongly resemble ACID transactions—and recent snapshot reads.
- Although most operations involve updating metadata for a single file (e.g. reading, writing to, or appending to a file), certain operations (such as creating and deleting files) involve atomically updating metadata entries

[5] Note that since this OCC validation phase is itself logged and executed as a standard built-in transaction, its outcome is entirely deterministic and proceeds identically at every system replica.

[6] Specifically, Calvin currently requires SQL statements that perform updates to identify all updated rows by primary key, and queries' WHERE clauses are restricted to predicates on columns on which a secondary index is maintained.

for both the file and its parent directory. Since these entries may not be stored on the same metadata server, these represent distributed transaction of exactly the type that Calvin processes very efficiently.

We demonstrated that CalvinFS can scale to billions of files and process hundreds of thousands of updates and millions of reads per second while maintaining consistently low read latencies. Like CalvinDB, it can survive entire data center failures with only small performance hiccups and no loss of availability.

4.3 Calvin in Costume: Mimicking Hyder and H-Store

To further demonstrate the flexibility of Calvin's modular implementation, we created configurations that duplicate the transaction execution protocols, logging mechanisms, and data storage structures of Hyder [3] and H-Store [18]. Both configurations involved modifying or extending only a few of Calvin's components; a surprisingly small amount of additional code was required to reproduce these two distinct architectures.

4.3.1 Calyder

Hyder is a transaction processing system for shared flash storage. In Hyder, an **executor** server runs each transaction based on a current snapshot view of the database, then adds an after-image of the transaction's updates (called an **intention**) to a distributed flash-resident log. Intentions are then applied to the database state by a deterministic **meld** process that performs optimistic validation checks, applying any transaction whose read-/write-sets have not been updated since the snapshot that the executor used to execute the transaction [3].

We created Calyder, a Calvin instantiation that uses a shared-flash log implementation based on Hyder's, and where a simple "executor" component at each Calvin server accepts transaction requests from clients and executes them using our optimistic client-side transaction mechanism (in which the "executor" acts as the client). Each scheduler then deterministically applies optimistic validation checks, exactly like Hyder's meld procedure.

Although our flash log implementation was somewhat simpler and slower than Hyder's, and our data storage mechanism involved additional copying of inserted record values, we were able to recreate performance measurements that resembled Hyder's, but with slightly higher transaction abort rates due to our implementation's somewhat longer delays between the execution and meld (optimistic validation) phases.

4.3.2 H-Cal

H-Store is a main-memory OLTP database system that executes transactions serially using a single thread per partition, eschewing all logging, locking, and latching and relying on in-memory replication for durability [18].

We created H-Cal, which uses a "fake" log component implementation in that doesn't actually record any durable state, but simply forwards transaction requests to the partitions at which they will execute, then executes each partition's transactions serially in a single thread using the H-Store scheduler discussed above.

Like H-Store, H-Cal achieves extremely high throughput for embarrassingly partitionable workloads, but its performance degrades steeply when transactions may span multiple partitions[7].

5 Conclusion

Four years of work on deterministic transaction scheduling in distributed storage systems have culminated in a robust prototype with excellent scalability and replication characteristics. Its modular and extensible implemen-

[7] H-Cal's performance actually degrades more gracefully than H-Store's in the presence of distributed transactions, since H-Store does not use a deterministic execution invariant to avoid distributed commit protocols. However, the current implementation of VoltDB, a commercialization of the H-Store project, implements a logical logging mechanism and deterministic execution invariant nearly identical to Calvin's [22].

tation allows new ideas to be quickly integrated and instantly tested in deployments that partition data across hundreds of machines and consistently replicate state across geographically disparate data centers.

We plan to release the Calvin codebase under an open-source license, with the hope that researchers and system architects will find Calvin useful, both in our provided configurations and as a common substrate for quickly prototyping new systems by swapping in novel implementations of individual components.

References

[1] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[2] D. Batoory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. Genesis: An extensible database management system. *IEEE Transactions on Software Engineering*, 1988.

[3] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *CIDR*, 2011.

[4] D. G. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full sql language support in microsoft sql azure. In *SIGMOD*, 2010.

[5] M. J. Carey, D. J. Dewitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. V. The exodus extensible dbms project: An overview. In *Readings in Object-Oriented Database Systems*, 1990.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *TOCS*, 2008.

[7] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB*, 2000.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[9] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. *Architecture of a Database System*. 2007.

[10] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 2010.

[11] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

[12] D. Lomet and M. F. Mokbel. Locking key ranges with unbundled transaction services. *VLDB*, 2009.

[13] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR*, 2009.

[14] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *CoRR*, 2011.

[15] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 145–156. VLDB Endowment, 2013.

[16] R. C. Sears. *Stasis: Flexible Transactional Storage*. PhD thesis, EECS Department, UC Berkeley, 2010.

[17] M. Seltzer. Oracle nosql database. In *Oracle White Paper*, 2011.

[18] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, 2007.

[19] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project voldemort. In *FAST*, 2012.

[20] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *VLDB*, 2010.

[21] A. Thomson, T. Diamond, S. chun Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[22] VoltDB. Website. voltdb.com.

The BW-Tree: A Latch-Free B-Tree for Log-Structured Flash Storage

Justin Levandoski Sudipta Sengupta
Microsoft Research
Redmond, WA
{justinle,sudipta}@microsoft.com

Abstract

The Bw-Tree is a high performance latch-free B-tree index that exploits log-structured storage. Its design addresses two emerging hardware platform trends. (1) Multi-core and main memory hierarchy: the Bw-tree is completely latch-free; it performs state changes (e.g., record updates, splits) as "deltas" prepended to prior state, installing new state via an atomic compare-and-swap instruction on an indirection page address mapping table. This improves performance by avoiding thread latch blocking while also improving multi-core cache behavior. (2) Flash storage: the Bw-tree organizes storage in a log-structured manner (using "delta" records) on flash to exploit fast sequential writes and mitigate adverse performance impact of random writes. This also reduces write amplification and extends device lifetime. This article provides an overview of the Bw-tree architecture and its technical innovations that achieve very high performance on modern hardware.

1 Introduction

A B-tree index supports high performance key-sequential access to both individual keys and designated subranges of keys. It is the combination of random and range access that has made B-trees the indexing method of choice within database systems and stand-alone atomic record stores. But the hardware infrastructure for which the B-tree was designed has changed dramatically. That infrastructure used processors whose uni-processor performance increased with Moore's Law, limiting the need for high levels of concurrency on a single machine. It used disks for persistent storage. Disk latency is now analogous to a round trip to Pluto [14]. Those days are gone. In this article, we provide an overview of the Bw-tree [12], a new B-tree whose design enables very high performance in the new hardware environment that has recently emerged. The Bw-tree addresses two important trends:

Design for multi-core. We now live in a multi-core world where uni-core speed will at best increase modestly. We need to better exploit a large number of cores by addressing at least two important aspects:

1. Multi-core CPUs mandate high concurrency. But, as the level of concurrency increases, latches are more likely to block, limiting scalability [1].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

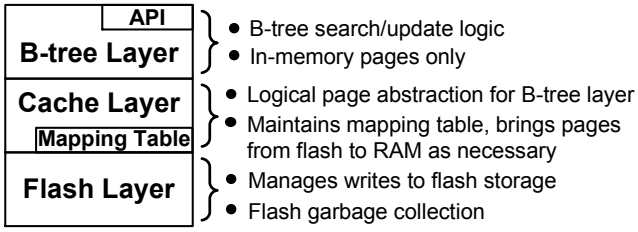


Figure 1: The Bw-tree architecture.

2. Good multi-core processor performance depends on high CPU cache hit ratios in a hierarchical cache architecture. Updating memory in place results in cache invalidations, so how and when updates are done needs great care.

Addressing the first issue, the Bw-tree is latch-free, ensuring a thread never yields or even re-directs its activity in the face of conflicts. Addressing the second issue, the Bw-tree performs "delta" updates that avoid updating a page in place, hence preserving previously cached lines of pages.

Design for Modern Storage Devices. Hard disk I/O operations per second (IOPS) performance is limited by disk seek time. Flash storage offers higher IOPS at lower cost. This is key to reducing costs for OLTP systems. Indeed, current storage systems such as Amazon's DynamoDB include explicit ability to exploit flash [5]. The Bw-tree targets flash storage as well. Flash has some performance idiosyncracies, however. While flash has fast random and sequential reads, it needs an erase cycle prior to re-write, hence uses "copy-on-write" and garbage collection mechanisms to provide the illusion of "in-place" updates. This makes random writes significantly less efficient than sequential writes in terms of both performance and wearing out the device [8]. While flash SSDs use a mapping layer (FTL) in order to hide the "copy-on-write" nature of writes and provide the abstraction of "logical" page, this cannot hide the noticeable slowdown in performance associated with random writes. Even high-end FusionIO drives exhibit a 3x faster sequential write performance than random writes [3]. The Bw-tree performs log structuring itself at its storage layer. This approach avoids dependence on the FTL and ensures that our write performance is high for both high-end and low-end flash devices.

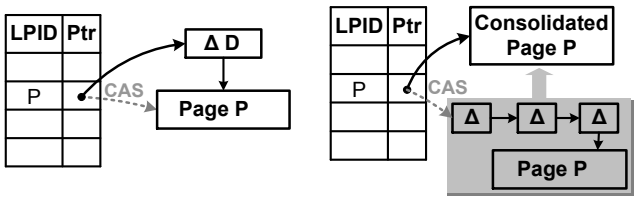
This article provides an overview of the Bw-tree. We first discuss the high-level architecture of the Bw-tree. We then describe the novel techniques used to achieve in-memory latch-free behavior. Next, we provide a brief discussion of our log-structuring technique to achieve persistence, and how the Bw-tree supports transactional functionality when part of a lager system (e.g., a deuteronomy-style architecture [11, 13]). We end by discussing the Bw-tree's role in Hekaton, Microsoft SQL Server's memory-optimized database engine.

2 The Bw-Tree Design

This section highlights the novel features of the Bw-tree. We begin by presenting the Bw-tree architecture and then discuss specific techniques that make the Bw-tree latch-free and optimized for log structured storage.

2.1 Bw-tree Architecture

The Bw-tree is a classic B+-tree [2] in many respects. It provides logarithmic access to keyed records from a one-dimensional key range, while providing linear time access to sub-ranges. Figure 1 depicts the Bw-tree architecture. The top *B-tree* layer provides the access method API, and is responsible for the search, record update, and structure modification logic common to a B-tree. The *cache layer* serves the *B-tree* layer with in-memory pages; it is responsible for swapping out pages into flash under memory pressure and bringing them back into memory when the B-tree layer accesses them. It is also responsible for installing "delta" updates on a page in a latch-free manner. The *flash layer* implements our log-structured store (LSS).



(a) Delta updates (b) Consolidating a page

Figure 2: Delta updates and consolidation.

This design is extremely versatile since (1) it is architecturally compatible with existing database kernels; (2) it is suitable as a stand-alone atomic record store, or "data component" in a decoupled transactional system [11, 13]; (3) due to its latch-freedom, it can serve as an efficient range index in a main-memory database (that is independent of the flash layer). In fact, the Bw-tree is the ordered index within Microsoft's Hekaton main-memory optimized database engine [7].

2.2 The Mapping Table

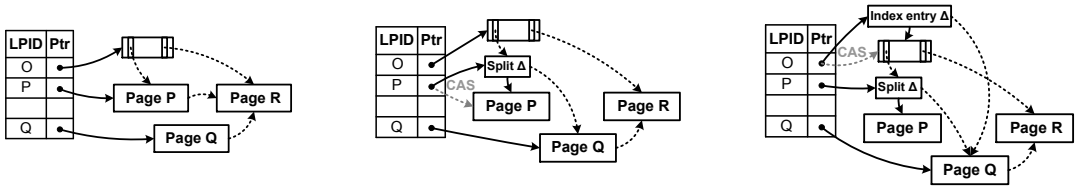
Our cache layer maintains a *mapping table*, that maps logical pages to physical pages; logical pages are identified by a logical "page identifier" or PID. The mapping table translates a PID into either (1) *a flash offset*, the address of a page on stable storage, or (2) *a memory pointer*, the address of the page in main memory. The mapping table is the central location for managing our "paginated" tree. All links between Bw-tree nodes are PIDs, not physical pointers. The mapping table enables the physical location of a Bw-tree node (page) to change on every update and every time a page is written to stable storage, without requiring that the location change propagate to the root of the tree, because inter-node links are PIDs that do not change. This "relocation" tolerance enables both delta updating of the node in main memory and log structuring of our stable storage.

Bw-tree nodes are thus logical and do not occupy fixed physical locations, either on stable storage or in main memory. This means we have flexibility in how we physically represent nodes. Furthermore, we permit page size to be elastic, meaning we can split pages when convenient as size constraints do not impose a splitting requirement.

2.3 Latch-Free In-Memory Operations

In our Bw-tree design, threads never block on Bw-tree pages in memory because we do not use latches. Being latch-free permits us to drive multi-core processors to close to 100% utilization. Instead of latches, we install state changes using compare and swap (CAS) instructions[1]. The Bw-tree provides an asynchronous operation completion pathway when it needs to fetch a page from stable storage (the LSS); this pathway would be taken rarely when the workload has a working set that is captured by the cache layer's page swapout mechanism (and fits within the provided memory limit).

This persistence of thread execution helps preserve core instruction caches, and avoids thread idle time and context switch costs. Further, the Bw-tree performs node updates via"delta updates" (attaching the update to an existing page), not via update-in-place (updating the existing page memory). Avoiding update-in-place reduces CPU cache invalidation, resulting in higher cache hit ratios. Reducing cache misses increases the instructions executed per cycle. This section describes our techniques for eliminating latches for in-memory operations.



(a) Creating sibling page Q (b) Installing split delta (c) Installing index entry delta

Figure 3: Latch-free split example.

2.3.1 Delta Updating

The Bw-tree updates pages by creating a delta record (describing the change) and prepending it to an existing page state (the delta record contains a pointer to the existing page state). It installs the (new) memory address of the delta record into the page's slot in the mapping table using a CAS instruction. If the CAS succeeds, the delta record address becomes the new physical "root" address of the page, thus updating the page. This strategy is used both for data changes (e.g., inserting a record) and management changes (e.g., spliting a page or flushing it to stable storage). Delta updating simultaneously enables latch-free access in the Bw-tree and preserves processor data caches by avoiding update-in-place. Figure 2(a) depicts a delta update record D prepended to page P; the dashed line represents P's original address, while the solid line to D represents P's new address.

We occasionally consolidate pages by creating a new page that applies all delta changes to a search optimized base page. This reduces memory footprint and improves search performance. A consolidated form of the page is also installed with a CAS, as depicted in Figure 2(b) showing the consolidation of page P with its deltas into a new "Consolidated Page P". We garbage collect the prior page (with deltas) by placing it on a pending list to be reclaimed when safe (i.e., when no other threads may access it) using an epoch based mechanism. Garbage collection and the CAS failure protocol are described in our full paper [12].

2.3.2 Structure Modifications

Structure modification operations (SMOs) such as node splits and merges introduce changes to more than one page. This presents a problem in a latch-free environment since (a) we cannot change multiple pages with a single CAS and (b) we cannot employ latches to protect parts of our index during the SMO. All Bw-tree SMOs are performed in a latch-free manner; to our knowledge this has never been done before. The main idea is to break an SMO into a sequence of atomic actions, each on a single page and installable via a CAS. We describe a latch-free page split. Page merge SMOs are described in [12].

The Bw-tree employs the B-link atomic split installation technique that works in two phases [10] as depicted in Figure 3. We split an existing page P by first creating a new page Q and initializing it with the records of the upper half of the key range (Figure 3(a)). We install Q in a new entry in the mapping table that is not yet visible to the rest of the tree. We then install a "split delta" on P (Figure 3(b)) that logically describes the split and provides a side-link to new sibling Q. We then post a (seach key, PID) index term for Q at parent O with a delta record, again using a CAS (Figure 3(c)). In order to make sure that no thread has to wait for an SMO to complete, a thread that sees a partial SMO will complete it before proceeding with its own operation.

[1]CAS is an atomic instruction that compares a given *old* value to a *current* value at location L, if the values are equal the instruction writes a *new* value to L, replacing *current*.

2.4 Log Structured Store

2.4.1 Caching

The cache layer is responsible for reading, flushing, and swapping pages between memory and flash. It maintains the mapping table and provides the abstraction of logical pages to the Bw-tree layer. Pages in main memory are occasionally written (flushed) to stable storage for a number of reasons. For instance, the Bw-tree may assist in transaction log checkpointing if it is part of a transactional system such as Deuteronomy [11, 13], or to reduce memory usage. Flushing is also required for implementing structure modifications correctly in the face of crash and subsequent recovery. Flushing and "swap out" of a page installs a flash offset in the mapping table and permits reclaiming page memory.

2.4.2 Storage Management

Our LSS has the usual advantages of log structuring [15]. Pages are written sequentially in a large batch, eliminating any write bottleneck by reducing the number of separate write I/Os required. The cache manager marshals bytes from the pointer representation of the page in main memory into a linear representation that can be written to the flush buffer. An important difference from [15] is the notion of *incremental flushing* that allows only the changed portion of a page to be flushed, thus further increasing the efficiency of log-structured writes and reducing the amount of garbage created on flash, and hence, write amplification.

Incremental flushing. To keep track of which part of the page is on stable storage and where it is, we use a *flush delta record*, which is installed by updating the mapping table entry for the page using a CAS. Flush delta records also record which changes to a page have been flushed so that subsequent flushes send *only* incremental page changes to stable storage. This can dramatically reduce how much data is written during a page flush, increasing the number of page updates that fit in the flush buffer, and hence reducing the number of I/O's per page. There is a penalty on reads, however, as a page read into memory requires multiple I/Os to fetch the non-contiguous parts of the page. This penalty is mitigated by the very high random read performance of flash.

Garbage collection. When the log usage exceeds configurable thresholds, the LSS cleaner reclaims garbage records in the log. It operates from the earliest written portion of the log, passing over orphaned records and copying valid records to the end of the log. Delta flushing reduces pressure on the LSS cleaner by reducing the amount of storage used per page. This reduces the "write amplification" that is a characteristic of log structuring. During cleaning, LSS makes pages and their deltas contiguous on flash for improved access performance.

2.5 Managing Transactional Logs

The Bw-tree is an atomic record store that can be "plugged" into a transactional system. When part of such a system, the Bw-tree layer must manage the transactional aspects imposed on it. To do this, we tag each update operation with a unique identifier that is typically the log sequence number (LSN) of the update on the transactional log (maintained elsewhere, e.g., in a transactional component [11]). LSNs are managed so as to support recovery idempotence, i.e., ensuring that operations are executed at most once, and to support transaction log management.

A major issue in a transactional system is enforcement of the write-ahead log protocol (WAL). A decoupled Deuteronomy [11] architecture enforces the WAL by a transactional component sending an LSN value, called the EOSL, representing its end of stable log to a data component (e.g., the Bw-tree). The data component is then allowed to make all operations stable with LSNs *less than or equal to* EOSL. Like conventional systems, the Bw-tree flushes pages lazily while honoring the WAL. Unconventionally, we do not block when performing a checkpoint that requires flushing a page whose updates are more recent than (have LSNs greater than) the EOSL used to enforce WAL [12]. Instead, because the recent updates are separate deltas from the rest of the page (a

property we guarantee in deciding when to consolidate a page), we can remove the "recent" updates (not on the stable transactional log) from pages when flushing.

3 Hekaton: The Bw-tree in Action

Hekaton is Microsoft SQL Server's memory-optimized database engine targeting OLTP workloads [4]. The engine is designed for high levels of concurrency and achieves great performance, demonstrating speedups of greater than an order of magnitude on real customer workloads [6]. A number of novel technical advancements help achieve such performance. For instance, Hekaton uses a new optimistic, multi-version concurrency control technique to avoid interference among transactions [9]. Also, the engine uses latch-free (lock-free) data structures in order to avoid physical interference among threads. Hekaton supports two access methods: hash indexes and ordered indexes. The Bw-tree serves as Hekaton's ordered index.

The Hekaton team considered several alternatives for its ordered index. One alternative was a skip list, which also provides key-ordered access, logarithmic search overhead, and can be implemented to provide latch-free updates. In several head-to-head performance evaluations, the Bw-tree consistently outperformed skip lists by a factor of 3-4x. Clearly, it did not achieve this advantage by merely being latch-free. Rather, the Bw-tree improvement stemmed from much better processor cache hit ratios, attributable to our no update-in-place approach, and the fundamental cache efficiency of a B-tree based structure. All experimental numbers, including cache hit comparisons, are reported in [12].

4 Acknowledgements

This is joint work with David Lomet. We would like to also thank the Hekaton team and several other product groups at Microsoft for providing us valuable feedback that improved the design of the Bw-tree.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB*, pages 266–277, 1999.

[2] D. Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[3] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *SIGMOD*, pages 25–36, 2011.

[4] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, 2013.

[5] Amazon DynamoDB. **`http://aws.amazon.com/dynamodb/`**.

[6] How Fast is Project Codenamed Hekaton? It's Wicked Fast. **`http://tinyurl.com/bzjync9`**.

[7] Hekaton Breaks Through. **`http://research.microsoft.com/en-us/news/features/hekaton-122012.aspx`**.

[8] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 10:1–10:9, 2009.

[9] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB*, 5(4):286–297, 2012.

[10] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *TODS*, 6(4):650–670, 1981.

[11] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction Support for Cloud Data. In *CIDR*, pages 123–133, 2011.

[12] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*, 2013.

[13] D. Lomet, A. Fekete, G. Weikum, and M. Zwilling. Unbundling Transaction Services in the Cloud. In *CIDR*, pages 123–133, 2009.

[14] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet. AlphaSort: A Cache-Sensitive Parallel External Sort. *VLDB Journal*, 4(4):603–627, 1995.

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

Call for Participation

39th International Conference on
Very Large Data Bases
Riva del Garda, Trento, August 26-30, 2013
http://www.vldb.org/2013/

Very Large Data Bases

VLDB is a premier annual international forum for data management and database researchers, vendors, practitioners, application developers, and users, covering current issues in data management, database and information systems research. The conference will feature 3 ***Keynote Talks***, 97 ***Research Presentations***, 6 ***Tutorials***, 48 ***Demonstrations***, 2 ***Panels***, an ***Industry Vision Session***, and 16 satellite ***Workshops***.

Registration is now open on the VLDB 2013 web site: **http://www.vldb.org/2013/**
The deadline for early registration is July 25, 2013.

VLDB 2013 will take place at the town of *Riva del Garda*, Italy. It is located close to the city of *Trento*, on the north shore of *Lake Garda*. The list of its famous guests includes Goethe, Freud, Nietzsche, the Mann brothers, Kafka, Lawrence, and more recently James Bond.

The shores of Lake Garda are full of picturesque villages, such as *Malcesine*, *Torri del Benaco*, *Sirmione* (where Maria Callas owned a villa), and many others. The cities of *Verona*, as well as *Trento*, where the Council of Trent took place in the 16th century, are within easy reach. *Venice*, *Padua*, and *Mantua*, with their rich cultural heritage, can be visited in a single-day trip. A little bit further away, *Ravenna* with its world-famous early Christian mosaics, as well as *Florence,* the birthplace of the Renaissance, and the entire region of *Tuscany* are very interesting and popular destinations.

Nature lovers should not miss the opportunity to visit the *Dolomite Mountains*, part of the *Alps* and a UNESCO World Heritage Site. These magnificent and impressive mountains offer numerous opportunities for hikes, mountaineering, rock climbing, as well as for relaxing at the shores of one of the many lakes.

The conference is organized by **dbTrento**, the Database and Information Management Group of the Information Engineering and Computer Science Department (DISI), at the University of Trento.

VLDB ***General Chairs***:
Themis Palpanas (University of Trento)
Yannis Velegrakis (University of Trento)

VLDB ***Program Chairs***:
Michael Böhlen (University of Zurich)
Christoph Koch (EPFL)

VLDB 2013

39th International Conference on
Very Large Data Bases 2013
Riva del Garda, Trento, August 26th - 30th

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398