Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 140

Systems Group, Department of Computer Science, ETH Zurich

Efficient Scan in Log-Structured Memory Data Stores

by

Kevin Bocksrocker

Supervised by

Prof. Donald Kossmann
Markus Pilman

March 2015 – September 2015

Informatik
Computer Science

Acknowledgements

I would like to thank my supervisors Prof. Donald Kossmann and Markus Pilman for sharing their knowledge with me and the great support during the thesis. I learned a lot while working on this project — even though discussions with Markus sometimes got a bit heated before realizing that we were both meaning the same.

Special thanks go to my family for supporting me and to all my friends for making life more worthwhile.

Abstract

Novel storage solutions like RAMCloud promise never seen before latency by storing all their data in main memory. To efficiently store and retrieve objects from memory some systems reuse ideas from log-structured file systems and utilise a log like data structure to hold the data objects in DRAM. Instead of updating data in place, with log-structured memory any modification to the data generates a new data entry which is appended sequentially to the log's head. An indexing structure (often a hash table) mapping an object's key to the location in the log of its most recently written value is used to enable fast random read access to any object by key. Due to the append-only nature the log grows over time and as such a garbage collection algorithm is needed to reclaim space by stripping inaccessible objects from the log.

While those systems provide exceptional random access performance (retrieving or writing an object with arbitrary key) they lack an efficient way to perform a scan over the whole data set. Scanning over all currently stored objects usually requires traversing the hash table and retrieving its associated data from the log. This approach has suboptimal performance due to poor CPU cache locality as the data tuples are scattered throughout the log under normal load.

Main goal of this master thesis is to devise an optimized log-structured memory data store offering a dedicated scan operation. This scan operation should be as efficient as possible while trying to preserve the very high random access performance the log-structured data layout enables in the first place. The system consists of the implementation of two different approaches: A baseline approach modeling the current state of the art log-structured memory algorithms and one novel approach optimized for both scan and random access operations. These implementations build on top of a full featured data store to read and write key-value pairs in DRAM including optimized memory allocation and garbage collection algorithms. Finally, the different approaches are evaluated regarding their performance under different workloads.

Contents

Chapter 1

Introduction

Traditionally data is stored in different systems for processing customer / business transactions (OLTP) and for analytical queries supporting the business in the decision-making process (OLAP). Periodically a ETL (Extract-Transform-Load) process is started to extract changed data from the OLTP system and insert it into a separate OLAP system where the analytical queries are run. This leads to a delay in the "data freshness" in the OLAP system as new data is not available immediately. With the increasing importance of analytical queries this gap in freshness is becoming more and more a problem. Running the analytic queries directly on the OLTP database or the other way around is often not possible because many systems are designed and optimized for one workload: OLTP systems support fast data retrieval and manipulation involving only a small subset of the data (like data belonging to one single customer) by short-running transactions. OLAP systems on the other hand have to process large amounts of data (like all orders placed in a specific time frame) in long-running read-only transactions involving complex queries.

These systems also often differ in the way they scale across multiple machines: OLTP systems typically use partitioning where the data in a partition is exclusively owned by a single machine. This approach scales well when processing transactions that only access one partition but executing transactions across partitions usually requires expensive coordination between the machines. OLAP systems on the other hand often employ a shared-data approach where the complete data set is accessible by every instance in the system. As a downside these systems often lack ACID transactions as data can be accessed and modified by all instances independently.

Recently, the Tell database system [11] has shown that it is possible to efficiently run OLTP workloads on a shared-data architecture by separating the processing from the storage layer. Tell runs on top of a shared-data store implemented as an ordinary key-value store. The storage layer is completely unaware of any running transactions as transaction handling is implemented in the processing layer using a novel way of distributed snapshot isolation. Due to the increasing need to run OLTP and OLAP queries on the same dataset the next step is to also support OLAP workloads on top of this storage layer. The Tell architecture provides a good starting point as it provides great OLTP performance on the storage architecture usually used to implement OLAP systems.

Tell builds upon RamCloud[14] as the storage layer — a key-value store keeping all its data in main memory. RamCloud enables fast retrieval and modification of its data by using a log-structured memory allocator to organize and manage all records in DRAM[16]. Inspired by similar work in filesystems this approach appends new data sequential at the head of a large buffer and uses a hash table to enable fast lookup of an element by its key. Because data is never deleted memory will be exhausted at some point and a garbage collection algorithm rewrites parts of the log to clean it from old elements. In addition RamCloud uses modern networking technologies like Infiniband to minimize network latency. RamCloud works well when accessing a record by its key but lacks an option to scan and process large amounts of data — which is required to execute OLAP queries in an efficient way.

1.1 Motivation

Motivated by the requirement to process OLTP and OLAP workloads on the same data set this thesis presents a reimplementation of the storage layer for use in the Tell database system. The new implementation reuses the ideas of a log-structured memory allocator and extends the data structures to enable a fast and efficient scan over the data stored in the system. The log-structured approach was chosen because it already showed stellar performance when used with a OLTP workload. As such this system tries to retain the high random-access performance required for OLTP while making the scan as fast as possible.

Chapter 2

Related Work

The idea of log-structured memory for use in a data store was first proposed by the RamCloud storage system[16]. As the Tell project was based on RamCloud in the beginning the system proposed here shares many similarities with RamCloud. As part of another master thesis[20] a scan over the hash table in RamCloud was implemented but was quickly abandoned because it did not perform well.

FoundationDB is a system that conceptually shares a lot of similarities with Tell: Both systems separate the processing from the storage layer and provide support for ACID transactions in the processing layer on top of a shared-data key-value store. FoundationDB was recently acquired by Apple and was shut down[1] — not much information about its inner working is available.

Recent work by Microsoft[10] also focuses on building a transactional component on top of a dedicated data component. The system reuses ideas from traditional log-structured systems to write data to storage but represents the data itself using a page based layout[9].

Recently a multitude of in-memory systems appeared that try to enable support for OLTP and OLAP workloads on top of the same data. One example is HyPer[8] which has a single dedicated process handling all OLTP queries and executes a fork of this process whenever a new OLAP query arrives. Data consistency is reached by relying on the virtual memory management system provided by the processor and operation system: Unmodified memory pages are shared between the OLTP and the OLAP processes and a private copy in the OLTP process is made whenever data is modified in memory. HyPer is only able to run on one single server which severely limits its scalability.

Another hybrid in-memory system supporting both OLTP and OLAP transactions on the same dataset is AIM[2]. Contrary to Tell this system is designed for event processing and the reaction to events under specific conditions. The implementation has no support for transactions or indexes. As such AIM is only suitable for specialized use cases while Tell attempts to provide a system that can be used with a wide range of OLTP and OLAP workloads.

[1] `http://techcrunch.com/2015/03/24/apple-acquires-durable-database-company-foundationdb/`

Chapter 3

Design

This chapter provides a detailed look into the design and the interplay of the storage and the processing layer. The storage layer is comprised of multiple storage nodes all running the storage server and providing a shared-data interface to its clients. Accordingly multiple (possibly different) processing nodes form the processing layer and are responsible for the transaction processing on top of the shared-data store.

The server running on the storage nodes supports multiple storage backends implementing a common interface. In this thesis the backend based on a log-structured memory allocator was implemented. In addition other backends are worked on but are out of scope of this thesis.

The chapter describes the data model used by the system, the concurrency control mechanisms providing the ACID guarantees, the supported data operations, the design of the log-structured memory storage backend and finally how data can be accessed over the network.

3.1 Data Model

The storage nodes support multiple tables accessed using an unique name which is internally mapped to a 64 bit table ID. Each table is associated with a schema defined by the user to which all records stored in the table have to adhere. The schema consists of a set of fields addressable using an unique field name and the type of the data stored in the field. It supports the common data types like fixed-width 16, 32 and 64 bit integers, single and double precision floating-point values as well as a variable-length string and a large-binary-objects (BLOB) type. A field can also be NULL which behaves like the NULL value known from SQL.

Records can only be retrieved from a storage node using the 64 bit record ID unique for each table. This record ID has to be assigned explicitly by the client when writing a new record. The IDs can be supplied by the user or generated by incrementing a distributed counter. To provide record lookup using a key other than the record ID an index based on a distributed B-Tree is used. These indexes can then be used to perform the mapping from any arbitrary key to the record ID of the target record. The distributed counter and the indexes are solely managed by the processing nodes and treated like regular data in the storage server.

The processing nodes are also solely responsible for sharding data between multiple storage nodes using a simple sharding algorithm: Every storage node is assigned a ID in a contiguous range (i.e. 0 to 4), when accessing a record the client hashes the table and record ID together and takes the hash modulo the number of servers. The resulting number determines the storage node responsible for the record.

3.2 Concurrency Control

The data store provides transactional guarantees by using snapshot isolation (SI) via Multiversion Concurrency Control (MVCC) [1]. On transaction start a snapshot is created containing only the records written by transactions that have already committed at that time. The new transaction is only able to read elements from this snapshot. A write in the transaction only succeeds when no other transaction (committed or uncommitted) modified the element outside of the snapshot — i.e. the newest version of the element is the same as in the snapshot. In the case of conflicting modifications the transaction aborts and reverts all the modifications that were written successfully prior to the conflict. With this mechanism the system is able to provide full ACID guarantees to its executing transactions.

When modifying an element MVCC does not override any existing data but keeps older versions of the record and then simply inserts a new version containing the modification. As such a transaction's snapshot can be created implicitly by associating it with a list of versions it can access. Global snapshot state is managed by a dedicated service called the *Commit Manager*: When starting a transaction the commit manager service assigns a monotonic increasing ID to the transaction and sends a list containing the IDs of transactions that have already committed as part of a *snapshot descriptor*. When writing new data the version of the record corresponds to the ID assigned to the transaction. More information can be found in the original Tell paper [11, Section 4].

The storage nodes themselves are not aware of any running transactions: When retrieving or writing a record on the storage node the client sends the snapshot descriptor of the transaction with its request. For read requests the storage node can create the transaction's snapshot implicitly from the information contained in the descriptor by locating the most recently written version written by the transactions whose IDs (version) are contained in the descriptor. This way the server is also able to detect a write-conflict if the most recently written version of the record was written by a transaction whose ID is not contained in the descriptor. Because a single storage node is unaware of the modifications made by the transaction on different storage nodes the server itself is only able to detect a conflict but unable to resolve it (i.e. by aborting the transaction). As such it is the responsibility of the processing node to revert all changes made within the conflicting transaction by issuing revert commands to all the involved storage nodes. In order to rollback a running transaction in situations where the processing node crashes, the client stores the keys of the records it is about to modify in a special table before performing the actual modification. After a processing node crash other processing nodes can retrieve the keys and revert all modifications of the unfinished transaction.

3.3 Non-Transactional Tables

Because the storage nodes are unaware of any transactions the processing node is more flexible in the way it handles a conflict. For example, this allows the processing nodes to provide a "non-transactional" interface for tables in addition to the "transactional" interface where a conflict leads to an abort of the transaction. They are primarily used to store the indexes as a write-write conflict of an index node can be resolved by simply retrying the operation.

Non-Transactional tables are implemented in the processing node and only use the conflict-detection of the storage nodes' snapshot isolation mechanism. A read operation is executed with a specially crafted snapshot descriptor containing the highest possible transaction ID as base version. The resulting snapshot descriptor matches all version numbers that can be possibly written and as such the storage nodes always returns the most recently written data for the key. To perform a compare-and-swap operation on an element with version v a write operation of version $v + 1$ is executed with a snapshot descriptor containing only version v in its snapshot. In the case that the element was modified by another operation the version number of the most recently written element will be higher than v and a conflict will be detected when trying to write the element. This also prevents the ABA problem often associated with the use of compare-and-swap operations as all operations are strictly serialized by the monotonic increasing version number. The processing node can then try to resolve the conflict depending on the type of data that is stored.

Concurrent transactional and non-transactional access to a single table is not supported as the version numbers have a different semantic. When creating a table it has to be declared as either “transactional” or “non-transactional” depending on the way the table needs to be accessed.

3.4 Data Operations

The storage node provides a simple interface to the processing nodes to retrieve and store data:

- A get operation returning the associated data for a given table and record ID and snapshot descriptor.
- An insert operation writing the record into the data store, returning an error in case the record already exists.
- An update operation overriding an existing record by creating a new version, returning an error in the case the modification failed due to a write-write conflict.
- A remove operation marking the record as deleted in the transaction’s version, returning an error in the case the modification failed due to a write-write conflict.
- A revert operation undoing the last modification (issued by the processing node when rolling back a transaction).

In addition, the storage node supports a scan operation to read multiple elements from a table at once. The system uses a shared scan to execute multiple scan queries at once: All scan queries that arrived during a specific timeframe are batched together and only one scan processing all queries together is active all the time. Because data is scanned only once for all queries this results in an increase in throughput as shown in [6]. As only one scan is active the scan can be executed in multiple threads simultaneously as it does not have to compete for resources with other active scans. This further results in a decreased response time. Queries are queued as long as a previous scan is still active, after the scan completes all queries waiting are dequeued and the scan restarts.

Scans are often only interested in a subset of the data. Consequently each scan request can be associated with a selection query containing conditions every record in the result set has to meet. To enable efficient processing of the requests in the storage nodes all selection queries have to be formulated in the conjunctive normal form, i.e. as a AND of multiple OR clauses. In this form evaluation of the query only needs a single bitvector with every bit corresponding to the result of one OR clause. Evaluating a query then proceeds as follows: The bitvector is initialized with every bit set to 0. Before evaluating a predicate the bitvector is checked if the corresponding bit of the OR-clause is already set. If it is, then a previous predicate from this OR-block already evaluated to true and the predicate can be skipped because the whole block already evaluated to true. If the bit is not set then the predicate is evaluated against the record’s data and if it evaluates to true the bit in the vector is set. After evaluating all predicates the bitvector needs to be checked — if all bits are set to 1 then the record matches the query. In the case a record matches a selection query the associated snapshot descriptor is compared to the version of the record. If the version is contained in the snapshot it is written into the query's network buffer and sent to the processing node.

When processing a record all queries of the different scans in the batch have to be evaluated to check if the record matches for each individual query string: To profit from better cache locality on these strings all query strings are copied into a contiguous memory block and then accessed sequentially from the scan threads.

As often only a subset of the fields in a record or simply an aggregation is required by the processing node the storage node also supports projection and aggregation mechanisms (sum, min, max, count). Join and Group-by algorithms are not supported and have to be implemented in the processing node.

3.5 Storage Backend

The implementation of the storage backend is based on the implementation proposed in [16] as implemented in RamCloud. It was adapted to support Multi-Version Concurrency Control and an efficient scan over all data contained in the log.

At the heart of the storage backend is the log and the hash table. The log is only written sequentially: New records are appended at the current log head and the head is advanced to point after the appended record — The remaining log is immutable. To enable fast random lookup of a record a hash table is used mapping the key to the memory address in the log where the associated record is stored. Different versions of a record are separately written to the log and form a "version chain" implemented as a linked list. The list is sorted by decreasing version number and the hash table always points to the newest element in the chain. To retrieve an element from a snapshot, the lookup operation first locates the newest version and follows the linked list until it encounters the first element contained in the snapshot. Implementations of these data structures are lock-free to enable concurrent access from multiple threads without the slow down of lock-congestion.

The log is divided into smaller independent memory segments called log pages. These pages have a fixed size of 8MB and form a linked list. When trying to write a new element into the log and there is not enough space left in the current head page a new head page is allocated pointing to the old and the element is written into the new page. Entries can only be appended to the head page. This makes it easier to deal with the memory exhaustion that the append-only nature of the log leads to sooner or later. When space is running low a garbage collection algorithm is required to identify segments that contain data which is no longer used — records that were overwritten and can no longer be accessed through any active snapshot. The garbage collection then rewrites the page by copying all records that are still active to the log head. Afterwards the page can be removed from the linked list and freed for later reuse.

In the past years many lock-free hash table implementations were developed each with trade-offs making it more or less suitable for use in the storage backend. Fundamentally hash tables can be categorized into closed and open addressing hash tables: Closed addressing hash tables like the one proposed in [13] use a fixed size bucket array. The buckets store a pointer to the head of a linked list holding the actual elements. Conflicts are resolved by simply appending the conflicting element to this bucket list. Another implementation proposed in [17] extends the concept to allow the bucket array to be resized but requires additional "boundary" elements which have to be skipped. All elements are allocated separately on the heap thus a lot of random accesses to memory are required. This makes them more susceptible to cache misses which are quite expensive in modern CPUs.

On the other hand open addressing hash tables – like the one proposed in [15] – store the data directly in the array: In case two different keys hash to the same bucket the conflicting element is stored in another bucket. The order in which buckets are probed has to be deterministic in order to be able to locate the bucket again when looking up the element later on. The memory access pattern in open addressing tables is more sequential than in closed addressing tables especially when conflicts are resolved by probing the adjacent buckets. The number of elements that can be inserted is limited by the size of the bucket array — contrary to closed addressing tables that only suffer from a performance hit as the overflow list becomes longer. A resizable open addressing hash table was proposed in [5] but it lacks a performance evaluation on a modern x64 based system. After the table reaches a certain size a new larger array is allocated and the threads migrate the elements from the old table to the new table.

When choosing the hash function the hash function used for sharding must be different than the hash function of the hash table in the storage node. Otherwise distribution of the keys in the hash table might be imbalanced with a number of buckets experiencing high conflict rates while other buckets are never accessed.

The version chain follows the approach from [7]: Inserting a new element between two elements can be done by copying the next pointer of the previous node into the next pointer of the own node and performing a compare-and-swap to change the next pointer of the previous element to point to the newly inserted element. Deletion has to tag the next-pointer of the node with a special "deleted" bit before it can be removed from the list. Another thread might try to insert a new element right after the element-to-be-deleted and will succeed without knowing that the element was removed in the meantime. First flagging the next pointer with a "deleted" bit will ensure that the compare-and-swap of the inserting thread will fail and the removal of the node can be detected.

The data structures involved in the log-structured memory architecture leave room for two fundamentally different approaches when implementing a full table scan: One scanning over the hash table and version chain, probing each element encountered from the log. The other performs a scan directly over the log and processes all records contained in it. Previous work by [20] has shown that a scan over the hash table does not perform well for RamCloud. As such this thesis concentrates on the possible performance gains when scanning data over the log. The hash table based scan is used as a baseline.

These approaches impose some requirements on how the data structures have to be designed in order to work efficiently. Scans are only ever performed over one table at a time hence both approaches profit if they only have to process the data that is actually relevant to them. Regarding the hash table this means allocating one large table shared by all storage tables or many smaller hash tables for every storage table created in the system. The same applies to the log: One log shared by all tables would mean that a scan operation also has to scan over all the "uninteresting" data from other tables. For the actual implementation a fixed size, open addressing hash table shared between all storage tables was chosen. These tables exhibit a better cache locality and are thus faster to scan. To keep the complexity low the chosen implementation is not resizable so a large table allocated at startup and shared by all storage tables is less likely to run out of space compared to allocating many smaller hash tables. The log was implemented in a way that uses one dedicated log per storage table. In the evaluation of the scan approaches only one table was used so this design choice has no impact on the direct outcome of the results.

A scan over the log also requires that all information required to process a record is stored alongside in the log. This includes the immutable key and the version the record was written (the "valid-from" version). In order to decide if a record is part of a given snapshot the scan also needs to know if the element in the log was overwritten by a newer version. If the newer

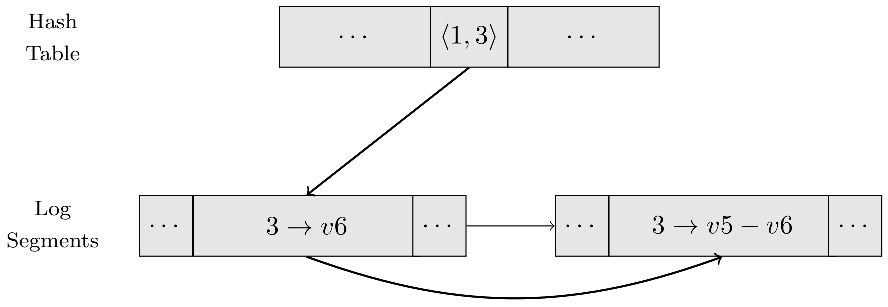

Figure 3.1: The interaction between the different data structures: The hash table stores a pointer to the log for element with record ID 3 in table 1. The log contains the immutable key and “valid-from” version in addition to the mutable “valid-to” and next-pointer — followed by the actual immutable record data.

version is already contained in the snapshot then the scan must not include the older element. Detecting if the current element is still valid would require a lookup in the hash table to check if the version chain contains a more recently written element. The additional access to the hash table will slow down the scan due to the non-sequential memory access. To solve this problem we relax the immutability constraint of the log and add a mutable “valid-to” field to every record written to the log. After overwriting an existing record the “valid-to” field of the old record is set to the “valid-from” version of the new record. By looking at the “valid-from” and “valid-to” versions of a record and comparing them with the snapshot descriptor the scan can decide if the record needs to be included in the snapshot or must be discarded.

The performance of the scan is also impacted by the amount of old, inaccessible records in the log. Garbage has to be skipped when scanning over the page leading to unnecessary and costly cache misses and performance drops. To keep the log pages as compact as possible the garbage collection is performed as part of the scan. While scanning over a page the utilization is measured and if it is below a certain threshold the next scan over the page will recycle the page as it scans through it. The additional cost of the garbage collection is amortized by a shorter runtime for following scans. Recycling the page involves moving the elements that are still “alive” to a new memory location but the element is still referenced by its old location in the version chain. Consequently the next pointer of the previous element in the linked list needs to be adjusted to point to the new memory location. In combination with the mutable “valid-to” field of every record this poses a problem: To remove an element from the linked list it has to be marked as deleted before it can actually be removed. After copying the record to the head of the log another thread setting the mutable “valid-to” version might still have a pointer to the record in the old location. The update by the thread will be lost after the garbage collection thread “fixed” the version chain to point to the new memory location. The solution is to store the next pointer of the linked list directly in the log besides the “valid-to” field. When copying a record during garbage collection the “valid-to” and “next” fields are copied from the old record. To remove the old element and insert the new element in the version chain a double-word compare-and-swap[1] is executed on both mutable fields of the old record, marking the old element as “deleted” and changing the next pointer to the copy of the element. The compare-and-swap will fail if another thread modified the mutable fields in the meantime, the changes can be copied over to the new record and the compare-and-swap can be retried. Threads setting the “valid-to” version also

[1]A double-width compare-and-swap is supported in the x86_64 architecture through the *CMPXCHG16B* instruction: It performs a 128 bit (2 words) compare-and-swap on a single contiguous memory location.

need to perform a double-width compare-and-swap on both fields: The operation will fail in case the garbage collection removed the record from the chain before the thread was able to modify the version.

Figure 3.1 shows the interaction between the different data structures for the record with ID 3 inserted into the table with ID 1. The hash table points to the most recently written element (version 6), the next pointer embedded in the record points to the element overridden by the new version. As such it contains a "valid" from version of 5 and a "valid-to" version of 6.

At this time the implementation supports no form of replication: All data is lost when terminating a storage node. The replication-problem is orthogonal to the problem presented here (efficient scan) and as such it is left for future work.

3.6 Data Access

Processing nodes communicate with the storage nodes using the low-latency Infiniband network in order to reduce the amount of time spent retrieving data. A high-throughput, low-latency network is quite important in shared data architectures as every piece of data has to be shipped from the storage nodes to the processing nodes. Measurements with the Tell system (see [11, Section 6.6]) have shown an increase in throughput by a factor of 6 when running on Infiniband compared to normal 10Gbit Ethernet. As such a high-performance Infiniband network stack is required to connect the storage and processing nodes. Unfortunately no existing implementation satisfied our requirements. Most libraries – like rsocket[2] – try to emulate a traditional socket-based interface on top of Infiniband but these approaches suffer a lot from the emulation overhead.

The RamCloud Infiniband library provides a blocking interface to a server, i.e. when sending a request the issuing thread is blocked until the response arrives. In the case of Tell[11, Section 6.3.1] this limited the throughput as the client thread is unable to do any work during this wait time. Increasing the number of worker threads on the client to allow execution of another thread while the original thread is blocked does not scale well as the OS has to perform a lot of (quite expensive) context switches between those threads. With one of the reasons for Infiniband's high throughput being that a data transfer does not involve the kernel one would like to bypass the kernel as much as possible.

Thus the network stack should allow communication to take place in a non-blocking fashion, i.e. in the background without blocking the processing thread. One common approach for non-blocking IO operations is to supply a callback function with every request which gets invoked as soon as the corresponding response is received (often also called a *continuation*). This solution performs great but becomes difficult for the user to program with as soon as multiple servers and message are involved: The user has to take care of creating and maintaining state information shared between the different continuations and react differently to events depending on this state. This often requires the user to write a lot of code responsible for "gluing" together the different code paths and results in interfaces that are difficult to use — A experience we made ourselves while trying to build a SQL processor on top of an callback-based TellDB interface.

Our solution is to adept a model based on coorporative multithreading: Each transaction started in a processing node is executed in its own user-level thread (called a *fiber*). Each request made in the transaction is sent in the background and a "future" object is returned to the user which acts as a proxy for the not-yet-received data. The transaction's code is not interrupted until the user wants to access data through the future which has not yet been received from the server. In this case the network stack transparently "yields" the fiber by

[2]http://git.openfabrics.org/?p=~shefty/librdmacm.git;a=summary

performing a context switch and schedules another yielded fiber that received the required data in the meantime. Executing each transaction in a fiber permits the system to multiplex many concurrently running transactions with very fast context switches between fibers as the kernel is not involved anymore. Returning futures enables the network stack to send requests in the background and gives the user the flexibility to decide when to wait for the respective response. This approach combines the performance of the non-blocking programming model with the ease of use of the blocking model.

When implementing transactions the user should keep in mind that the fiber executing the transaction only supports cooperative multi-threading: While a fiber executes the transaction's code the remaining transactions are unable to regain control by themselves (non-preemptive) and the processing of incoming responses is stalled, thus increasing the response latency of other transactions. To keep the overall response latency low, transactions should dispatch long running computations to their own thread or perform regular yields to hand the control back to the scheduler and allow other transactions to execute.

Contrary to the datagram-based or stream-based transfer models of UDP and TCP, Infiniband adapters transfer data in a message-based model. As such the network throughput can be limited by two factors, the maximum message rate or the maximum bandwidth. In the case of a data store messages tend to be rather small: A simple get request containing a table ID and a record ID normally amounts to only a few bytes of data. The size of the following response from the server depends on the size of the retrieved tuple but might also be only a few bytes in size.

Measurements using the Send Bandwidth tool from the OFED Infiniband Performance Test tools[3] between two machines connected by Mellanox 40Gbit/s QDR cards (40Gbit/s signaling, 32Gbit/s data) show a limit of 7Mpps (Million messages per second) for 128 byte messages. This results in an average bandwidth of $\sim$ 6.9Gbit/s — less than 25% of the total available bandwidth. When sending larger messages of 16KB the message rate drops to 0.25Mpps but throughput increases to $\sim$ 29Gbit/s. As a consequence the throughput over Infiniband is not limited by bandwidth but rather by the number of requests send over the network.

To increase throughput the "logical" messages exchanged between storage and processing nodes are batched together into a single message sent over the network. As a side effect this message batching delays the sending of a request to a later point in time, this results in a latency increase of individual messages but the overall throughput achievable when sending small messages will be higher.

[3]http://git.openfabrics.org/?p=~grockah/perftest.git;a=summary

Chapter 4

Implementation

Following the introduction of the basic mechanisms of the system this chapter provides a more detailed description regarding the concrete implementation. It shows the inner workings of the different components involved in the Tell ecosystem.

During the course of this master thesis work was done on four individual components: *InfinIO*, a library providing a low-latency, high-throughput Infiniband network stack. *TellStore*, an implementation of the storage node with a Log-Structured Memory storage backend. The *CommitManager*, a small service managing the global snapshot information. And finally a port from the RamCloud backend to the TellStore backend of the *Bd-Tree*[11, Section 5.2.1], a latch-free B+Tree developed as part of previous work on the Tell project.

The chapter describes the implementation of the most important data structures involved in the system: The *snapshot descriptor* used to manage the global snapshot state in the system and the different parts of the storage backend – hash table, log and version chain – followed by a description of the two scan approaches.

The storage nodes expose the operations supported by the storage backend over the network. A number of network threads is dedicated to handle *get* and *put* requests from clients and a number of dedicated scan threads solely responsible for executing the scan. For the best performance the number of threads should be chosen such that each thread can run on a dedicated core.

4.1 Snapshot Descriptor

The global snapshot state is managed by a dedicated *Commit Manager* service. Clients connect to the commit manager in order to start and commit transactions.

At the core of the commit manager lies the *snapshot descriptor* data structure storing the state of all transactions in the system. The snapshot descriptor consists of a *base version* and a "rolling" bitmap with each bit storing whether the transaction committed or is still active. The base version denotes the version-offset of the first bit in the bitmap — e.g. with a base version of 10 the first bit stores the status of the transaction with version 11. This design prevents the bitmap from increasing in size as the system starts and commits more and more transactions over time: Eventually all transactions having IDs smaller than i will commit and the bitmap can be "rolled-forward" by setting the base version to i and discarding the bits of all transactions before i. When starting a new transaction the size of the bitmap is incremented by one and the corresponding version number is assigned to the new transaction. The commit manager then sends the assigned version and a copy of the snapshot descriptor to the client. Upon commit of a transaction the commit manager only has to set the associated bit in the version bitmap and – if applicable – adjusts the base version.

The commit manager also keeps track of the *lowest active version*. This number denotes the lowest version number that is visible in any currently active snapshot. This information is used by the storage node to determine if a record written by MVCC can be discarded.

4.2 Hash Table

The implementation of the hash table is inspired by the approach described in [15]. It is a lock-free, open-addressing hash table consisting of a large, fixed size bucket array allocated once at startup and shared between all tables. Conflict detection is resolved by linear-probing with an interval of 1 as this reduces costly CPU cache misses when accessing buckets in a non-sequential way.

Each bucket is 24 byte in size and consists of the table ID, the record ID and a pointer to the record's data. The 3 least-significant bits of the pointer are used to encode the current state of the bucket and as such pointers are required to be aligned on a 8 byte boundary (which is normally the case in the x64 architecture). A bucket can either be in the state "free", "inserting", "set", "deleted" or "invalid" with initially all buckets being "free".

The implementation imposes a number of restrictions on the data in the hash table:

- No element with table ID 0 can be inserted.
- The stored pointers must be unique, i.e. different buckets are not allowed to store the same pointer.
- After removing an element from the hash table the associated pointer can only be reinserted again after making sure that no other thread still has a reference to it.

These restrictions are necessary to make sure that no ABA problems arise during operations in the hash table and all apply in this concrete use-case: Table IDs are assigned by the storage nodes starting with ID 1, the pointer to a record is unique and only inserted once and memory occupied by a record is only reused after being garbage collected.

The hash table supports four different operations on the data: Retrieving the record pointer associated with a table and record ID, inserting a new record pointer and updating/deleting an existing record.

Lookup To lookup a value the table ID and record ID are each multiplied with a random salt chosen at startup and then hashed together using the hash functions provided by the Boost library[1]. The index of the bucket is determined from the hash value modulo the array size. At first the pointer encoding the state is loaded atomically and if the state is "free" then the element is not in the hash table and a null pointer indicating "not-found" is returned. In the case it is "set" the table and then the record ID are loaded in two separate atomic instructions. Finally the pointer has to be retrieved again and if it was changed then another thread modified the bucket in the meantime and the bucket has to be checked again. If the pointer was unchanged and the key (i.e. table and record ID) read from the bucket matches the target key the element is found and the associated pointer is returned. Otherwise and when the state of the pointer is neither "free" nor "set" the bucket is skipped and the next one is checked.

[1] http://www.boost.org/doc/libs/1_59_0/doc/html/hash/reference.html#boost.hash_combine

Insertion Inserting an element looks up the bucket using the hash and searches for the first bucket B_1 that is either "free" or "deleted". Unless the target key was already encountered on the probe path the insert claims the bucket B_1 by performing a compare-and-swap to set the pointer to the record pointer with an encoded state of "inserting". After claiming a bucket the operation writes the record ID and table ID (in this order) and starts searching for potential conflicts: It rechecks every element on the probe path until it encounters a "free" bucket. For every bucket B_2 encountered that is in the "set" or "inserting" state it checks if it presents a potential conflict by loading the pointer, record and table ID as described in the lookup operation.

If no conflict was detected the insert performs a compare-and-swap on its own bucket changing the pointer from the "insert" state into the "set" state and when successful returns. In case the compare-and-swap fails then another thread detected a conflict and aborted the insert by moving it into the "invalid" state. Consequently the insert failed and has to release the bucket B_1 by writing a table ID of 0 and setting the state to "deleted" before returning — Setting the table ID to the special value 0 is required so that no other key can conflict.

If a conflict was detected and the conflicting bucket B_2 is already in the "set" state then another insert operation succeeded. The state of the bucket B_1 has to first be transitioned into the "invalid" state so that other threads skip the bucket. Then the bucket can be released like described earlier. In case the conflicting bucket B_2 is still in the "inserting" state the operation tries to resolve the conflict by letting the insert operation in the bucket that is closest to the start bucket proceed. If B_2 is located on the probe path after bucket B_1 then a compare-and-swap on the pointer tries to move the bucket B_2 into the "invalid" state. If B_2 is located before B_1 then the operation backs off and releases the bucket B_1 (as described earlier).

This mechanism is required to prevent two threads from concurrently inserting the element at the same time. By scanning over the whole probe path backing off (and aborting) when encountering a conflicting bucket located earlier in the path and actively moving buckets into the "invalid" state that are located after the own bucket the implementation ensures that only one concurrent insert of the same key succeeds while also ensuring at least one operation is able to complete the insertion. The special table ID of 0 is required to indicate if the thread claiming a bucket is already in the conflict-detection mode or is still busy writing the record and table ID. Alternatively this situation could be detected by encoding an additional "conflict-detection" state in the pointer.

Update The bucket of the existing element is located as in the normal lookup operation. To update an element a compare-and-swap is performed on the bucket's pointer changing the pointer to a new value. The compare-and-swap is necessary to make sure that the bucket was not modifed (e.g. deleted) by another concurrent operation. The implementation also allows the user to pass the expected pointer (i.e. read by an earlier lookup) and the update will use the user-supplied pointer when performing the compare-and-swap.

Delete Deleting a entry from the hash table proceeds similar to the update operation: The bucket's pointer is first updated with a null pointer encoding the "invalid" state to prevent other threads from modifying the bucket. It is then released by writing a table ID of 0 and moving the bucket into the "deleted" state.

The described implementation is sensible to deletions: When searching for an element that is not in the hash table or after an insert the lookup has to iterate over all buckets until it finds a bucket in the "free" state. As deleted buckets are unable to transition back into the "free" state the number of "free" buckets decreases over time and as such the number of buckets that have to be checked increases. The implementation in [15] uses a counter associated with each bucket that limits the number of buckets that have to be probed. This improvement will work but storing an additional counter increases memory requirements by 8 byte from 24 byte per bucket to 32 byte (an increase of 33%).

While no exact measurements on the collision rate of the hash table were performed during system profiling only a small amount of the time needed to process a request was spent in the hash table – indicating that the collision rate is reasonably low – so we decided to concentrate on other parts of the system. Future work might reexamine the collision rate and eventually implement the bound mechanism.

The implementation is lock-free but not wait-free for scenarios in which a single bucket is heavily updated by a lot of concurrent threads: In this case a thread after reading pointer, table ID and record ID always sees a different value when re-reading the pointer (because another thread made progress and updated the value) so the thread has to re-read the bucket's data again and again.

4.3 Log

All records written to a storage node are stored in memory allocated by a log-structured memory allocator based on the ideas presented in [16].

At the beginning each log consists of a single empty log page. As described earlier the log pages form a linked list with every page storing a pointer to the previous page. In addition every log page contains an offset value pointing past all entries already written to the page and a counter used by the garbage collection to store the amount of garbage the page had during the last scan. The offset value also encodes a bit that indicates if a page is "sealed" and no other entries can be appended to the page. At the beginning all bytes in the remaining "data region" of the log are set to 0.

Entries are written sequentially and contiguously after this small page header. Every entry in the log starts with a 32 bit field storing its size, followed by a 32 bit user-defined type ID and the data of the record. Inserting a new element proceeds as following: The memory location of the next (potentially) available entry is read from the offset value stored in the log page. To reserve space for the entry a compare-and-swap is performed at this memory location trying to set this value from 0 to the required size of the log entry. The compare-and-swap succeeds if no other thread tried to write a log entry at the same time as all bytes in the log page are initialized to 0. In the case the operation fails another thread already acquired this entry, the size of the entry is read and the operation tries to acquire the log element after the current one. After successfully reserving space in the log page the thread tries to adjust the offset value to point after the currently written element. It tries to compare-and-swap the old offset value to point to the memory location past the written element. In case the operation fails and the current offset value does not point *at least* past the inserted element the operation is retried.

The synchronization point when acquiring space in a log page happens when the thread successfully wrote the size into the log. The offset value is only updated lazily and works as a shortcut pointing "approximately" past the last entry. This behavior is needed for the log-scan: To reach all entries while scanning over a page, the size of every entry has to be written first as this size is used to determine the position of the next element in the page.

In case there is no more space left in the head log page the thread allocates a new page from a buffer pool. The address of the current head log page is written to the next pointer and a compare-and-swap is executed to replace the old head. In the case the operation fails, another thread already allocated a new head, the page is released back into the buffer pool and the insertion is retried.

4.4 Version Chain

The version chain implements a linked list as proposed by [7] that builds on top of the hash table and the log: The pointer to the head of the linked list is stored in the hash table for every record with each next pointer stored as a mutable field in the log alongside the record data. The records in the list are sorted such that the head always points to the record with the highest version number with earlier versions following from newest to oldest.

The implementation provides the operations needed to implement the data retrieval and manipulation operations provided by the storage nodes — Namely, lookup, insertion, update, delete and revert.

Lookup To look up an element the head pointer is retrieved from the hash table and the linked list is followed until the first record is reached that is contained in the snapshot. While iterating over the list and encountering an element marked as deleted the iterator tries to fix the link by ultimately removing the element: It performs a compare-and-swap to set the next pointer of the previous element to point to the element after the deleted element. In case the fix-up fails and the iterator can not guarantee its consistency – i.e. both the previous and current element were marked as invalid in the meantime – it resets itself to the beginning of the version chain and restarts iteration from there on.

Insertion / Update / Delete All three data modification operations are similar at their core: By checking the version chain for the element the operation determines if the modification is valid or constitutes a write-write conflict. Then the data of the new record is written to the log – or in the case of a deletion a special tombstone value – and the next field is set to point to the previous element. Finally the entry is inserted into the version chain by updating the pointer in the hash table from the old to the new element. If the update fails because another thread already inserted another element a write-write conflict must have occurred, the already written log entry is marked as invalid and the conflict is reported to the client. If the insertion into the version chain failed the “valid-to” version of the element that was overridden is set to the “valid-from” version of the new element.

Revert The revert operation is the only operation that actively removes data from the version chain. It resets the version chain to an earlier state by removing the most recently written version and is used to rollback modifications by an aborting transaction. It locates the head element of the version chain by using the hash table, marks the head element as “deleted” and updates the head pointer to point to the element following the current head. Afterwards the revert operation has to clear the “valid-to” version of the previous element.

To prevent race conditions with other operations on the element in the time between replacing the head element and resetting the “valid-to” field any insert / update or delete operations have to abort in case the “valid-to” version has not yet been reset by the revert. This still preserves the semantic of a write-write conflict: Elements can only be reverted when the transaction that wrote the element is still active, consequently other write operations would have to fail with a write-write conflict as the uncommitted transaction will never be contained in another snapshot. Additionally, the processing node must also make sure that all other modification operations sent to the storage nodes as part of an aborting transaction have been completely executed before issuing revert commands.

4.5 Hash Table Scan

The scan begins at the first bucket of the hash table and if the bucket is in the state “set” the bucket is read as usual. Afterwards the version chain is followed and the queries attached to the scan are evaluated for every element in the chain. After reaching the end of the version chain and in case the bucket’s state was not “set” the scan moves on to the next bucket until it reaches the end of the bucket array.

The multi-threaded scan is implemented by dividing the bucket array into distinct and equally sized partitions. Each partition is then processed by a dedicated scan thread. This approach is sensitive to skew in the data distribution of the bucket array – i.e. a partition containing much more entries than the other partitions – as new scans can only be scheduled after all scan threads completed the scan. In our implementation the number of partitions equals the number of scan threads and the hash function is chosen in a way that tries to distribute keys equally over the whole bucket space. Alternatively one could create smaller partitions and push those into a queue shared between the scan threads with every thread retrieving a new partition from the queue after the thread processed its current partition.

Currently there exists no garbage collection implementation when using the hash table based scan. The implementation was omitted for time reasons as the scan over the hash table does not benefit from a compacted log like the log based scan does. Future work might implement a dedicated garbage collection algorithm.

4.6 Log Scan

The scan over the log begins at the head page of the table’s log. As all required information needed to process the records are stored alongside its data the scan can simply pick the first entry from the log page and evaluate all queries attached to the scan using the key, “valid-from” and “valid-to” version stored in the log. After reaching the end of the log page the next pointer of the page is followed to reach the previous head page.

Multi-threaded scan is implemented by keeping an approximate counter of the number of pages currently stored in the log. At the beginning of the scan this counter is read and the set of log pages is partitioned by following the linked list and assigning every scan thread a subset of the linked list.

Garbage collection is performed as part of the scan: While scanning over a page the implementation keeps track of the amount of inaccessible objects. An object is inaccessible if it satisfies one of the following conditions.

- It is marked as invalid because insertion of the object into the version chain failed.
- The *valid-to* field is set to a version equal or lower the current *lowest active version* in which case no transaction is able to read the object.
- The object marks a deletion tombstone and its *valid-from* field is set to a version equal or lower the current *lowest active version* in which case no transaction is able to read the object the tombstone marks as deleted.

This garbage counter is then stored in the context field of the log page when the page and all its entries are sealed (i.e. no more entries can be appended to the page and all entries have been completely written). The next time a scan is performed over the page this garbage counter is read and if the ratio of inaccessible to accessible objects crosses a certain threshold (currently set at 50%) the page will be garbage collected. During garbage collection the scan processes the page as normal but copies every accessible object it encounters into a new private log. The reference to the object in the version chain is then replaced with a reference to the copy of the object. After all accessible objects have been copied the log page is removed from the log and the page is marked for reuse through the Safe Memory Reclamation mechanism. When the scan completed processing all pages the pages of the private log are appended to the head of the table's main log.

Chapter 5

Evaluation

The system is evaluated using a simple benchmark mixing OLTP and OLAP queries and a microbenchmark contrasting the performance of the two different scan approaches supported by the system.

All benchmarks were executed on a cluster of 12 machines equipped with two Quad-Core Intel Xeon E5-2609 processors in every machine running at 2.4Ghz. Each processor in the machines forms a NUMA unit and is connected to 64GB of DDR3-RAM. In all runs the storage server and processing client were bound to NUMA node 0. The servers in the cluster were connected using a Mellanox QDR InfiniBand network card providing a signaling rate of 40Gbps and a data rate of 32GBps. The cluster was running Debian 8 and every software component developed during this thesis was built with GCC 4.9.2 in Release-Mode with processor-specific and link-time optimization enabled.

5.1 Scan-Benchmark

Before evaluting the performance of the whole system this section first takes a look at the performance of the hash table based and the log based scan implementations.

5.1.1 Methodology

The benchmark was executed with one storage server running two threads executing the scans and two network threads. One single Processing client was running with two network threads connecting to their respective counter part on the storage. In addition one commit manager was running on an additional dedicated machine. Garbage collection was completely disabled.

One single table was created on the storage server and populate with 100M records of 200 bytes in size (resulting in $\sim$ 18.5GiB data). The storage node was started with 48GiB reserved for the log and space for 2^28 entries ($\sim$ 134M) in the hash table resulting in a load factor of 0.75. As only one storage table was created no data from other storage tables is stored in the hash table. The inserted records occupied $\sim$ 22.35GiB in the log consisting of the record data including the 40 byte header of each log entry. As a baseline the time required to retrieve the same records as selected by the scan using normal, single-record *get* operations was measured. To evaluate the performance of log-fragmentation in the log-based scan an additional run was executed where after inserting the records another 100M records were written into the log overwriting the previously inserted records uniformly at random. The measurements were repeated 4 times with a selection query matching on 100%, 50%, 25%, 12.5% of all records contained in the table.

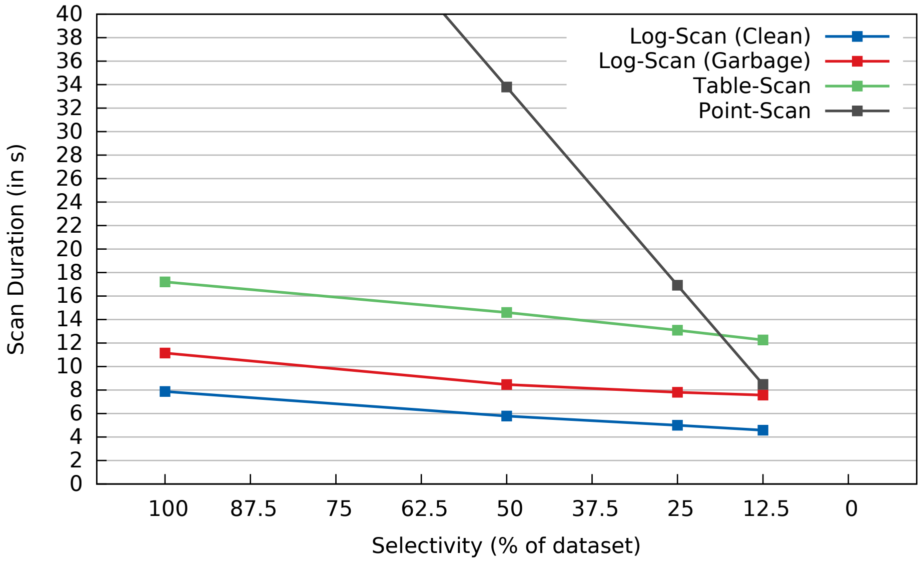

Figure 5.1: Duration of a scan over a table with 100M elements of 200 bytes ($\sim$ 18.5GiB data)

Finally the impact of garbage-collection is examined closer by populating a table with 80M elements of 200 bytes ($\sim$ 14.9GiB data) and then inserting another 80M records overwriting the existing tuples. In this case garbage collection was enabled and a log-based scan is executed three times consecutively: The first run scanning over the garbage contained in the table, the second run performing garbage collection and third run after compacting.

5.1.2 Results

Figure 5.1 shows the runtime of the different scan approaches depending on their selectivity: In all cases the log-based scan is the fastest when the log is clean from any garbage requiring 7.9 seconds to scan over $\sim$ 22.35GiB of log data. The scan approach was able to process 12.6M records per second ($\sim$ 24.3Gbit/s) and network utilization was $\sim$ 20.33Gbit/s. In the second scenario where the log contains the same amount of garbage as useful data the scan takes around 3 seconds longer processing around 9M records per second. During the 11.1s runtime the scan was able to process $\sim$ 44.7GiB of data ($\sim$ 34.91Gbit/s) and network utilization was $\sim$ 14.35GBit/s. On the other hand the table-scan required twice the time as a clean log-scan clearly showing the performance penalty of cache-misses in modern CPUs. The hash table scan was able to process 5.8M records per second ($\sim$ 11.2Gbit/s) with a network utilization of $\sim$ 9.3Gbit/s. Throughput when retrieving data record by record using the "point-scan" was 1.4M records per second.

When decreasing the selectivity the scan duration decreased accordingly as less records have to be copied and sent over the network: Except for the "point-scan" the response time decreased by 25% when cutting the selectivity by halve. Contrary to the other approaches the "point-scan" consisting of ordinary *get* operations only accesses the data that it really needs thus the response time scales linearly with the selectivity. The log and table scans have to scan over the whole data set even when the query is only interested in a subset but this way queries can also profit from the shared scan[1].

[1]The impact of the shared scan is investigated in the following benchmark

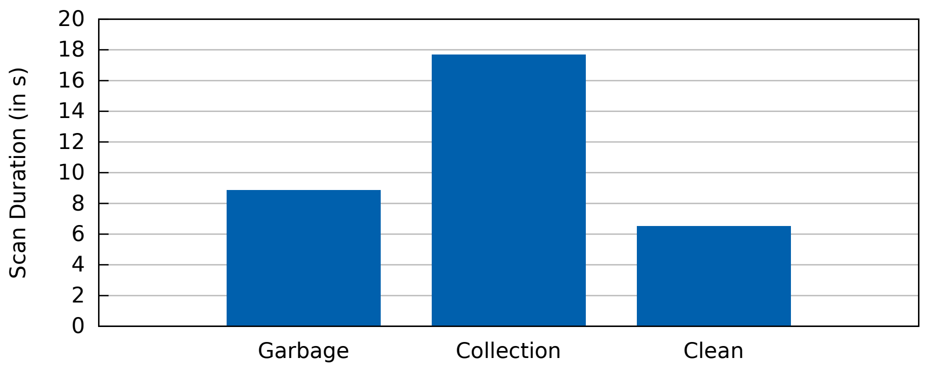

Figure 5.2: Duration of 3 consecutive log-based scans over a table with 80M elements of 200 bytes ($\sim$ 14.9GiB data) and 100% selectivity: First run with 50% garbage contained in the table, second run while performing garbage collection and third run after compacting

This indicates that a scan is only worth-while when scanning over a large portion of the data: As soon as selectivity drops below 10% a range index retrieving the required data record by record should be considered as an alternative. On the other hand an additional index requires work maintaining the index when inserting and removing elements from the table.

Impact of Garbage-Collection

Figure 5.2 shows the impact of garbage and the cost paid when performing garbage collection as part of the scan. At first a scan is performed over 80M records where on average every second is invalid, in the second run the garbage is collected as part of the scan and compacted, the third run finally runs on the compacted log. Simply scanning over the garbage results in an increase in runtime of about 2.3 seconds to a total of 8.8s compared to 6.5s running on the compacted log. Performing the garbage collection results in a spike of 17.6s needed for one scan run. This cost of nearly 8 seconds has to be paid once: After garbage collection the optimal scan performance is restored for further scans. In this case after 4 scan runs the cost of garbage collection has been amortized again.

It should be noted that this scenario shows one extreme as every second record in the log is invalidated. In reality tables that are scanned heavily are garbage collected more incrementally before such a large amounts of garbage can accumulate. Nonetheless the results give a good insight into the performance penalty and gains by performing frequent garbage collection.

5.1.3 Discussion

In all cases the hash table based scan is greatly outperformed by the log based: In an optimal scenario the log-scan is twice as fast as the table-scan. Even with a huge amount of garbage accumulated in the log the log-scan still beats the competition as it benefits from the fast sequential memory access modern CPUs provide. For a high selectivity the log-scan is always the preferred choice but when decreasing the selectivity below 10% the overhead of scanning over all records increases — making a range index lookup issuing a number of individual read requests a viable alternative that should be taken into consideration.

5.2 Mixed-Workload Benchmark

In order to test the complete system a benchmark was devised to measure the performance under different workloads. As one of the main motivations behind an efficient scan in log-structured memory was to enable the execution of a mixed OLTP and OLAP workload the focus of this benchmark lies in this area.

It was inspired by the TPC-C[19] benchmark which is a popular OLTP benchmark but much simplified and augmented with a dedicated OLAP workload.

5.2.1 Methodology

The benchmark models the typical user behavior when interacting with a retail system: Random searches for items, placing new orders and then paying for the order. These transactional operations were augmented with a simple analytic query performing an aggregation on the ordered items in a specific time window. In order to keep the database size constant during the whole run a batch query pruning all orders older than a specific time is running in the background. As we are mostly interested in the scale-out and performance interactions between OLTP and OLAP workloads the benchmarks focus on these two metrics.

Every run was executed for 9 minutes. From this runtime the first 3 minutes and the last minute were omitted to let the system warm-up and cool-down. This leads to a steady-state measurement interval of 5 minutes.

Each storage server instance was running with two threads executing the scans and two network threads. Processing clients were running one load generation thread and two network threads connecting to their respective counter part on the storage. In addition one commit manager was running on an additional dedicated machine. While the client would be able to run one more processing thread on its fourth core this design allows us to equally scale the processing power of the clients and the server. The size of the hash table and the log were set big enough so that no component becomes the bottleneck by being overloaded[2]. All benchmarks were only executed using the log-based scan approach as it was the one that showed superior performance in the scan microbenchmark. The hash-table-based scan was omitted for time reasons but future work may evaluate the hash-table-based scan in the same way as the evaluation using the log-based scan described in this chapter. Garbage collection was set to an interval of 60 seconds.

Prior to each benchmark run the database was populated with an existing dataset. The amount of elements inserted during population depends on the scale factor S which equals the number of storage servers running. This scale factor is required to keep the amount of data stored on each machine constant when evaluating the scale-out of the system. Otherwise the system would be prone to experience super-linear scaling as the size of the dataset stored on each machine would be smaller when running in a configuration with higher number of machines.

The scale-out experiments had to be capped at a maximum of 4 storage/processing nodes (i.e. 8 machines) because the commit manager started to become a bottleneck when scaling out to more machines especially for the mixed workload. A way to scale-out the commit manager to more machines was shown in [11, Section 4.2]. Future work has to revisit the current commit manager implementation in order to fix this bottleneck. Nonetheless, it was also shown that this similar system is able to scale to at least 8 nodes.

[2]For every storage server 8GB were allocated for the hash table and 32GB for the log

0	1	2	3	4	5	6	7	8	9
BAR	OUGHT	ABLE	PRI	PRES	ESE	ANTI	CALLY	ATION	EING

Table 5.1: Syllable list for name generation

5.2.2 Schema

The sample database consists of 3 different tables with each table having one index.

Item The *item* table contains all items that the retail store is selling. Each item has a unique ID, a name, a small description, a price and is associated with a specific category. In addition, the table has an multi-column index on the *category* and *price* field storing the unique ID. It is sorted and supports prefix queries which can be used to retrieve items of a specific category according to a price range.

Category names are generated similarly as last names are generated in the TPC-C benchmark[19, Section 4.3.2.3]. For each run the number of categories n is fixed beforehand. In order to generate a valid category name a random number is generated between 0 and $n-1$. The final category string is then generated by computing the modulo 10 of n and appending the corresponding syllable taken from Table 5.1 to the category string, n is then divided by 10 and the process is repeated for the new value of n until $n = 0$. For example, the corresponding category string for $n = 8$ would be "ATION" while the string for $n = 18$ would be "ATIONOUGHT".

The detailed schema of the table is as follows:

Field	Type	Description
itemId	Key	Integer key (as used in the storage)
name	String	Fixed string containing 12 characters
description	String	Fixed string containing 20 characters
category	String	Variable length string
price	Integer	Random number between 10 and 10,000

The table is read-only and populated before starting each benchmark run: The number of categories is fixed at $2\,500 \times S$ categories and the table is populated with $2\,500\,000 \times S$ items. While inserting the elements, every item is assigned to a category uniformly at random (leading to an average of 1 000 items per category) and appointed with an monotonic increasing item ID. Every entry in the *item* table has a fixed size of 36 bytes plus the length of the category string which varies between 3 bytes and 20 bytes — resulting in a typical table size of 120MB $\times S$ or 220MB $\times S$ when including the record header associated with each element.

Customer All customers that have an account in the retail store's system are stored in the *customer* table. Each customer has a unique ID, a unique username and an associated street, city and country. The table has an unique index on the username providing a mapping from username to customer ID.

Customer usernames are generated as in the TPC-C benchmark[19, Section 4.3.2.3] depending on the customer ID: The customer ID n is taken modulo 1000 and for each of the 3 digits (starting from least-significant) the corresponding syllable from Table 5.1 is appended to the username. As each username has to be unique the ID n is divided by 1000 with an integer division and the resulting number is appended to the username as string. For example, the username for $n = 12$ would be "ABLEOUGHTBAR0" and for $n = 5012$ the username would be "ABLEOUGHTBAR5".

The detailed schema of the table is as follows:

Field	Type	Description
customerId	Key	Unique integer key (as used in the storage)
name	String	Variable length string
street	String	Fixed string containing 10 characters
city	String	Fixed string containing 10 characters
country	String	Fixed string containing 10 characters

Like the *item* table, the table is read-only and is populated with $n = 2\,500 \times S$ customer records before the start of each benchmark run. All customer IDs are unique and in the range $(1, n)$. Every entry in the *customer* table has a fixed size of 30 bytes plus the length of the customer name string which typically varies between 10 and 20 bytes — resulting in a table size of 120KB $\times$ S or 220KB $\times$ S when including the record header associated with each element.

Orderline Customers must be able to place orders into the system which are stored in the *orderline* table. Each item in an order generates a new entry into the *orderline* table storing the customer ID, a order ID that is unique to the order, the item ID, the current timestamp, the quantity, the total price (quantity multiplied with the ordered item's price) and a status that is either "unpaid" or "paid". The table is denormalized so analytic queries can scan over the data more efficiently without requiring a potential join over a dedicated *order* table containing the common information of the order like time or status. In addition the table has an index on the customer ID and order ID field to enable efficient lookups of the items in a customer's order.

The detailed schema of the table is as follows:

Field	Type	Description
id	Key	Unique storage key (implicit - not used)
orderId	Big Integer	Unique integer ID
customerId	Foreign Key	Key of the customer
itemId	Foreign Key	Key of the item
time	Big Integer	Current timestamp (64 bit)
quantity	Integer	Random number between 1 and 10
price	Integer	Quantity * Item price
status	Integer	Either "unpaid" (1) or "paid" (2)

It is the only table in the benchmark that is written, as such the population of the table depends on the workload and is described later for each workload separately. Every entry in the *orderline* table has a fixed size of 44 bytes or 84 bytes when including the record header used in the storage backend.

5.2.3 Workload

The query load consists of 3 workloads that are executed separately.

OLTP Workload

The queries in this session try to cover a range of different OLTP access patterns like index lookups (point and range), point lookups as well as inserting and updating the database. The queries model the interactions of a typical user with a retail store and consist of five different queries each running in their own transaction. When started, each session will issue these transactions one after another in the order they are presented here.

Load generation of the OLTP workload is modeled as a closed queuing network: Every client in the system keeps a constant number of active sessions in the system and the potential throughput of the system is measured. After a session completes its execution another session is started. In all following benchmarks this number was set to 16 per client. This number was evaluated and set beforehand as it seems to put a good load on the system without overloading it. Increasing the number of concurrently active transactions will only result in an increase in response time. The behavior can be compared with e.g. a set of webservers connected to the system processing only a fixed amount of sessions at the same time.

This type of load generation was chosen as in real OLTP scenarios one is often interested in the throughput of the system, i.e. how many queries per second can the system sustain. The response time of OLTP systems is often bound by a "Service-Level Agreement" (SLA) stating that queries have to complete within a specific timeframe. In the most cases a OLTP query was started as part of a user request and the SLA wants to guarantee that the user has the feeling of receiving a response immediately. As such an limit of 500ms was imposed on the response time of all queries in the OLTP workload: Queries taking longer than this are not counted as part of the throughput. This limit is set arbitrarily but should be low enough for a typical user to be "instant" (for example, the TPC-C benchmark sets this limit at 5s).

OLTP Queries

The OLTP workload consists of the following five queries:

Category-Lookup At the beginning of each session the customer starts with a random browse for items where the user wants to see the least expensive items from a specific category: From the existing categories a name *c* is generated uniformly at random together with a random number *n* between 10 and 20 (also uniformly). A prefix-search is then performed on the price-index of the *item* table for the first (least expensive) item of the chosen category. The following *n* items are then retrieved from the *item* table. The transaction is read-only and performs a range prefix-lookup on an index where only the top *n* results are of interest. The query can be formulated in SQL in the following way:

```
n := Random number between 10 and 20
c := Name of a randomly selected category
SELECT * FROM item WHERE category = c ORDER BY price ASC LIMIT n;
```

Item-Lookup Besides browsing for items in a category the user wants to see information about specific items: Between 10 and 20 valid item IDs are generated uniformly at random and the details of those products are fetched. This models a read-only transaction retrieving a few records without requiring an index.

```
n := Random number between 10 and 20
i := List of n randomly selected item IDs
SELECT * FROM item WHERE itemId IN i;
```

Customer-Lookup Before placing a new order the user has to login into his personal account: A random (and valid) customer name is generated uniformly and the information associated with the account is retrieved. It is a very short-running read-only transaction requiring a point-lookup on an index and retrieving one element from the customer table.

```
c := Name of a randomly selected customer
SELECT * FROM customer WHERE name = c;
```

New-Order After browsing for a category and a set of specific items and having logged in, the customer is ready to place a new order: First the transaction generates a unique and not yet assigned number of the order. It then selects between 5 and 15 items returned by the previous run of the *Category-Lookup* and *Item-Lookup* queries and places a new order into the orderline table using the order ID and customer ID of the user retrieved with the *Customer-Lookup* query. Each inserted element has the current timestamp, a random quantity for each item and is marked as "unpaid". The transaction is insert only into the *orderline* table with inserts into the corresponding index.

```
cid := ID of the customer
oid := Unique ID associated with this particular order
items := Between 5 and 15 items consisting of <id, price, quantity>

foreach i in items:
  INSERT orderline (customerId, orderId, itemId, time, quantity, price, status)
      VALUES (cid, oid, i.id, now(), i.quantity, i.price * i.quantity, UNPAID);
```

Process-Order The last step in a customer's session is to pay for the previously ordered items: The index is used to query all items placed in the order with the customer and order ID, the status is then set to "paid" and the updated entries are written back. The transaction constitutes a read-modify-write cycle on elements selected by a range lookup through the order-index.

```
cid := ID of the customer
oid := ID of the order
UPDATE orderline SET status = PAID WHERE customerID = cid AND orderID = oid;
```

This workload will produce in average a load consisting of 30 read operations on the *item* table, 1 record on the *customer* table, 10 reads from the *orderline* and 10 insertions and 10 updates to the *orderline* table.

OLAP Workload

In addition to the OLTP query mix we developed a small workload consisting of one query describing a typical analytic access pattern using scans exclusively.

Contrary to the OLTP workload the OLAP load generation is modeled as an open queuing network: Throughput is fixed at a specific rate and new queries are generated periodically to reach the fixed throughput rate. As such, the response time of the OLAP queries is measured.

When modeling the OLTP and OLAP load generators both as closed queuing networks and running them together in a mixed workload one would increase the load on the system regarding two different variables: As the number of clients in the system increases more OLTP sessions are active leading to an change in the size of the *orderline* table. In addition, the number of running OLAP sessions increases which leads to an increase in the number of concurrently running scan queries. As a result the load on the scan infrastructure is increased regarding two dependent load-variables – the amount of data in the system and the number of active queries – that could potentially influence each other. This lead us to model the OLAP load generator as an open system: OLAP query throughput is fixed at all times during the scale-out experiments and only the amount of data is scaled.

This type of load generation is also closer to a real world scenario where throughput of OLAP queries is often fixed — besides some seldom ad-hoc queries there is only a need to run a specific set of OLAP queries periodically. In these cases the response time is of more interest.

OLAP Queries

The OLAP workload consists of the following query.

Aggregated-Sales With their operations backed by a blazing fast database the big bosses of the retail store want to know how many expensive cars they are able to buy: A 5 second timeframe in the last 180 seconds is picked uniformly at random and a scan selecting all placed orders in that timeframe is executed aggregating on the sum of the price field. This models a long-running read-only analytic query using a scan with selection and aggregation. It can be written in SQL in the following form.

```
t := Random timestamp between now() and (now() - 180s)
SELECT sum(price) FROM orderline WHERE time >= t AND time <= t + 5s;
```

Mixed Workload

When simply combining the OLTP workload and OLAP workload and run them together as a mixed workload the benchmark will not produce any consistent and meaningful results: The response time of any OLAP query involving a scan will depend on the size of the data that is scanned over. In the course of time the OLTP workload will insert more and more data into the system, the scan operation has to process this additional data and as such the response time of OLAP queries will increase. The resulting response time metric would suffer from high deviation as queries would gradually require more and more time to complete.

As such we propose a third workload that runs as a batch job in parallel to the OLTP and OLAP workloads and will prune the database (particularly the *orderline* table) from old data. It is run only once every 5 seconds by a single client. The combination of all 3 workloads results in a mixed workload that should exhibit constant performance when running during a longer period of time.

Mixed Queries

The mixed workload consists of the queries from the OLTP workload (as described in Section 5.2.3) and from the OLAP workload (as described in Section 5.2.3) with the addition of one batch query.

Clean-Orders Prunes the *orderline* table from any order that was paid by the customer and is older than 180 seconds. It performs a scan over the *orderline* table selecting only the relevant tuples with a projection to filter the columns that are not necessary (i.e. only key, customer ID and order ID are selected). Afterwards all selected entries are deleted from the database. The query is only issued once every 5 seconds by a single client. Formulated as SQL query:

```
DELETE FROM orderline WHERE time < (now() - 180s) AND status = PAID;
```

5.2.4 Results

The performance of the system is evaluated regarding its behavior when scaling-out using the three different workloads described earlier. The experiments try to show the most interesting metrics from the most interesting configurations of all the possible combinations. All experiments were run with varying number of storage servers and processing clients with the number of servers and clients scaled equally (i.e. 1 server with 1 client, 2 servers with 2 clients, etc). This restriction

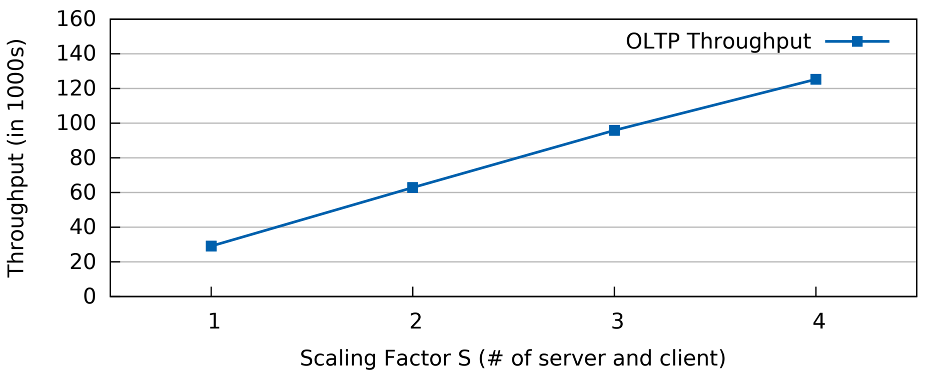

Figure 5.3: Scale-Out for 1 to 4 storage server / processing clients showing total throughput when running the OLTP workload only

was necessary to control the size of the experiments as this would increase the number of possible benchmark configurations even further.

At first the system is evaluated using the OLTP workload only, followed by an evaluation of the OLAP workload and finally in the mixed workload.

OLTP-Only Workload

The workload was run in configurations for 1 to 4 storage server and processing client machines and the average throughput per second was measured in Figure 5.3. As described earlier the measurement was performed over a window of 5 minutes. Prior to each run the database was populated according to the corresponding scaling factor (i.e. number of storage servers running) except for the *orderline* table which started empty and was populated during the warm-up phase.

As can be seen the increase in throughput is nearly linear when scaling out to 4 machines. When increasing the number of machines from a 1 server/client configuration to 2 server/clients the increase is even super-linear[3]. A possible and common explanation for this would be that with two server machines running, the amount of fast L3-Cache in the server is doubled. As such the probability is higher that often retrieved data is located in the cache — like the root node from heavily used indexes. With the latency for accessing data from the L3 cache being only about 30 cycles compared to main memory which additionally requires up to 50 nanoseconds more this could result in a small but visible increase in performance.

With an average of $\sim$ 30k Tps per client the complete OLTP workload can be executed 6,000 times per second by every client (as one run of the workload consists of 5 transactions). Every workload on average consists of 41 reads, 10 inserts and 10 updates, this leads to 246k reads, 60k inserts and 60k updates per second and client or to a total amount of 366k operations per second and client.

Figure 5.4 details the average response time for the five different queries involved in the workload. From there we can see that the super-linear scale-out is mostly due to the *New-Order* transaction where the average response time decreases from 1.9ms to 1.7ms when running the query in a configuration with 2 storage servers and processing clients. As it is the only query that exhibits a decrease in response time it seems unlikely that the system is profiting from a larger CPU cache from which all queries would profit. The *New-Order* query is the only one

[3]To be sure the super-linear speedup was not due to measurement errors the experiments were repeated and did indeed yield the same result

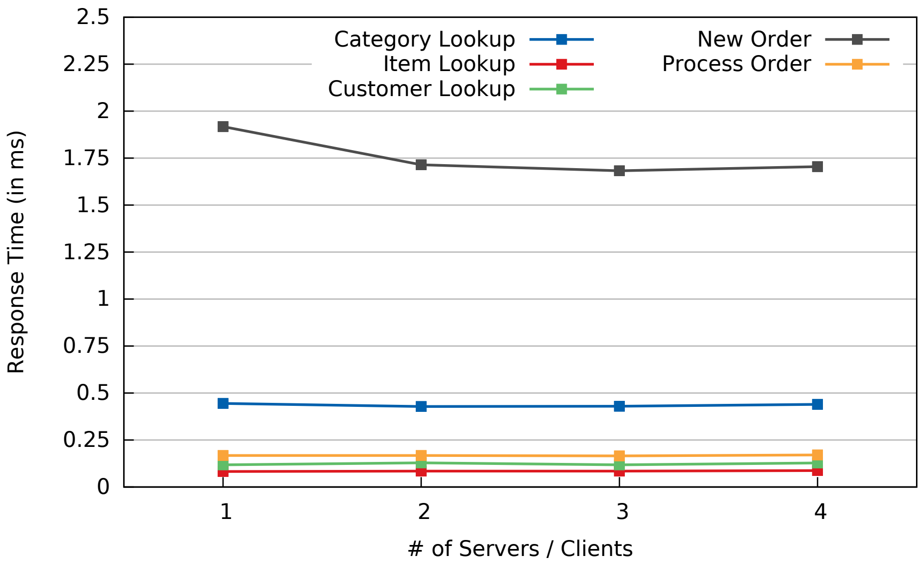

Figure 5.4: Scale-Out for 1 to 4 storage server / processing clients showing response times for individual queries when running the OLTP workload only

inserting data into the index so it could be the case that the behavior of the index has an impact on the response time — further profiling will be required to determine the exact cause for this.

Except for the *New-Order* transaction the overall response times remained nearly identical when scaling the system out underlining the great scalability behavior of the system. Unsurprisingly the *Item-Lookup* query requires the least amount of time (only around 84μs on average) as it only consists of point lookups directly on the storage without requiring an index lookup first. Behind the *Item-Lookup* query follows the *Customer-Lookup* query with an average of about 120μs which involves a single index lookup and retrieval of the single element. Not far behind is the *Process-Order* transaction requiring around 170μs which constitutes a range lookup on a simple index (only integer keys) and a read-modify-write cycle. Finally the *Category-Lookup* requires around 460μs to complete on average as it operates on a larger index (containing a string field) with additional get-requests to the storage. With response times below 2ms every query fulfilled the previously set SLA by a far and wide margin.

During all runs the error of the average Tps was well below 0.5% but the data showed small periods of time where Tps was considerably lower: In the run with 4 storage server / processing clients the average Tps was 125k with a minimum Tps of 97k Tps during the 5 minute run — a distinct hit in the throughput. This was also reflected across the response times of the individual queries, while the response time for the *Category-Lookup* averaged at 0.44ms, it maxed out at 247ms — a factor of 500! To further investigate this phenomenon Figure 5.5 shows the achieved throughput-per-second over the course of all the runs (excluding the warm-up and cool-down period).

The reader can easily recognize the small but periodic drops in throughput roughly every 60 seconds. This significance of the drop is dependent on the scaling factor but with a drop of 30k Tps the difference is noticeable when running in a configuration of 4 server/clients. Interestingly this interval equals the interval the garbage collection was set to run. Indeed disabling the garbage collection mechanism results in a steady throughput performance (not shown here for

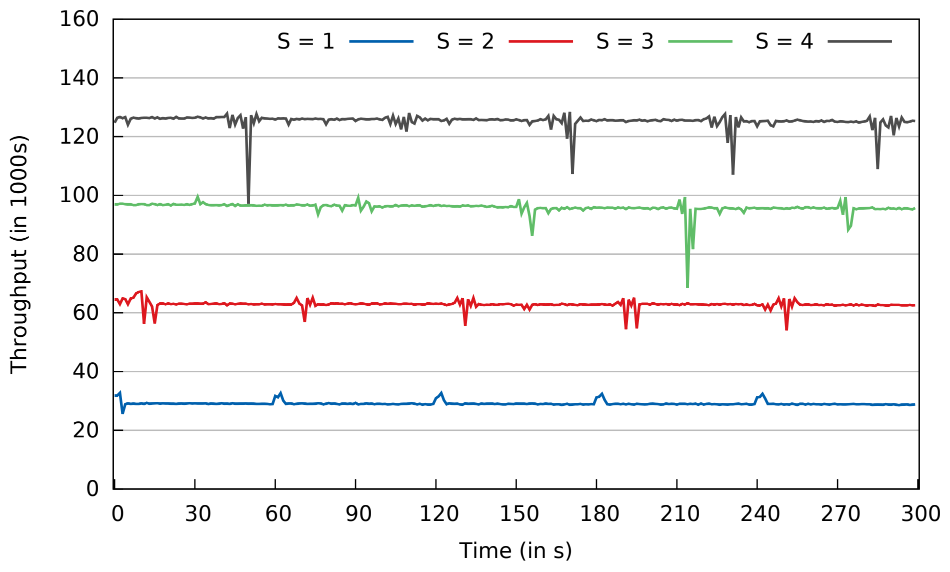

Figure 5.5: Throughput per second as time progresses of the scale-out from 1 to 4 storage server / processing clients

brevity). This is even more interesting as garbage collection runs entirely in the scan threads which are isolated from the network threads. The increased pressure from the garbage collection on the shared resources of the processor (i.e. L3-Cache and DRAM) leads to an decrease in OLTP performance (a situation we will encounter again later when running the mixed workload).

Confusingly the throughput increases while garbage collection is running when using only one storage server and one processing client. This is surprising as only the scan is able to profit from regular garbage collection which is not used in the OLTP only workload. The exact cause of this anomaly requires deeper system profiling and remains to be investigated.

OLAP-Only Workload

After taking a look at the pure OLTP performance of the system the benchmark is executed running the OLAP workload only. The *item* and *customer* tables of the database are populated like before and in addition 3 minutes worth of data is inserted into the *orderline* table. As the OLTP-Only workload was able to insert 60k entries per second and client into the *orderline* table a population size of $180 \times 60\,000 \times S$ records was chosen with S being the scaling-factor (i.e. number of storage servers). This leads to a population of 10.8M records per storage server or a total data size of $\sim$ 900MB in each server's log[4]. It should be noted that this way all tuples will be written sequentially into the log of each storage node without any invalid data in between. As such the executed scan operations are able to benefit the most from sequential read of the log-based scan and are not slowed down by any concurrently garbage collection operations — The impact of garbage collection will be visible when evaluating the system using the mixed workload.

Figure 5.6 shows the average response time for 1 to 4 storage server / processing client machines when running the OLAP workload for two fixed OLAP throughputs. One for a fixed throughput of 2.5 queries per second where every 0.4s a new *Aggregated-Sales* query is started

[4]The data volume of 900MB consist of the 44 byte data for each record plus the 40 byte header

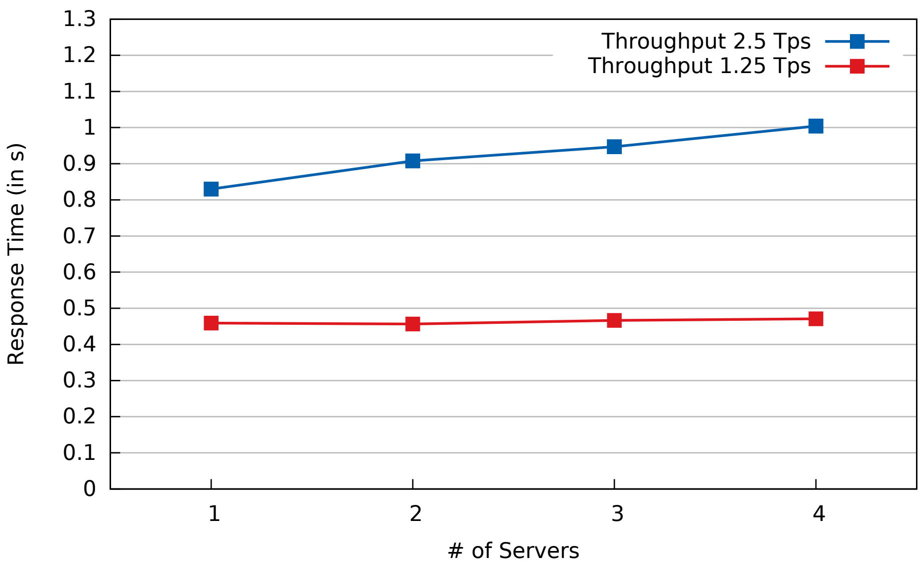

Figure 5.6: Scale-Out for 1 to 4 storage server / processing clients when running the OLAP workload

and one for 1.25 queries per second where every 0.8s a new query is started. In the best case we expect the response time to stay constant when scaling-out: With an increasing number of servers the amount of overall data increases as well through the scaling factor — The ratio between available processing power and amount of data stays the same. In reality the response time stays constant when running with a throughput of 1.25Tps but increases by about 50ms when loading the system with 2.5Tps (in all runs the deviation was below 1%).

This small increase in response time is due to the batch processing mechanism of the scan: After starting the scan operation the request has to wait until the storage server processed the previous batch before it can be started. Because a scan operation can only complete in the client after the scan requests was processed by all servers in the system the overall response time is also influenced by the highest wait time in the system. With four machines the probability that a scan operation on one machine has to wait for a longer time is higher than when running the scan on only one machine — A effect that will be further amplified when the data is not sharded equally between machines in which case one machine also has to scan over more data.

When issuing scan operations at a throughput of 1.25Tps the scans execute always on their own and are never batched (as the previous scan query has already completed). In this case we see that a single storage server is able to scan over the 10.8M records in about 450ms resulting in a performance of 24M tuples processed per second and server or in 41ns needed per tuple.

Interestingly when doubling the throughput and queries are starting to be processed together in a single batch the response time also nearly doubles. When issuing queries at a rate of 2.5 queries per second (i.e. every 400ms) and an average response time of 830ms (for $S = 1$) between 2 and 3 queries are typically processed together in one batch. Because of the query batching the same amount of data will be scanned as when executing only one query, only the query parameters (i.e. selection and aggregation predicates) have to be evaluated for each batched query individually. The jump in response time from 450ms when evaluating only one query to 830ms when evaluating between 2 and 3 queries at once indicates that evaluating the query parameters requires a not-insignificant amount of time. Trying to increase the throughput to 5 OLAP queries per second results in system overload: More queries are started than can be processed at the same time. As a result more and more queries have to be processed per batch leading to an increase in response time which in turn leads to more queries waiting to be processed — until the system runs out of resources. This suspicion is confirmed when running the scan in a low-level system profiler: Even with a few queries batched together the runtime of the code responsible for evaluating the query dominates the runtime of the code performing the actual scan of the data. Future work needs to revisit this part of the system in order to improve performance.

Mixed Workload

Finally the system is evaluated regarding the performance when running the mixed workload. As in the previous benchmarks the *item* and *customer* tables of the database were populated according to their scaling factor and as in the OLAP-only runs 3 minutes worth of data was inserted into the *orderline* table. The throughput of the OLTP query was fixed at 1.25 Tps while 16 OLTP sessions were concurrently active for each client.

Running the mixed-workload benchmark turned out to be a challenge from the beginning: Particularly the *Clean-Orders* transaction proved to be troublesome. When executed as part of the regular OLTP queries the *Clean-Orders* transaction could not keep up with the insertion load even when running in a fairly small setup with 3 storage server and 3 processing client instances. As more tuple were inserted per second than the clean orders transaction was able to remove, the transaction had to process more and more data during one run leading to an ever increasing runtime. This in turn increased the length of the snapshot descriptor's version bitmap as the long running transaction prevented the bitmap from being "rolled forward". Because of this increase in length from a few bytes to a few kilobytes – data which has to be sent with every single request – the throughput decreased as the load on the network increased. The decrease in throughput lead to a decrease in the number of tuple inserted per second and after a while the *Clean-Orders* transaction was able to handle the lower load again. When executing a heavier load (4 storage nodes and 4 processing nodes) the Commit Manager reached its maximum internal size limit of 64KB ($\sim$ 500k version entries) preventing the system from starting more transactions.

This has shown that the current implementation of the snapshot descriptor data structure is not well suited to support both long and short running transactions. A possible improvement would be to compress the bitmap for long runs of committed transactions. This would improve space usage but potentially leads to a decrease in lookup performance. The position of the bit for a given version can no longer be easily computed and it will not be possible to query the version bitmap in constant time. In addition the descriptor may be cached in the storage node for the duration of a transaction but this would require an additional command to the storage node to evict the descriptor from the cache upon completion of the transaction.

In the end all experiments were executed with one additional processing thread in the master client being exclusive to the clean order transaction (running on the client's spare core). Without having to share resources with other transactions in the same processing thread the clean order transaction was able to keep up with the load more reliably.

The actual data of the average throughput per second for the OLTP and the average response time for the OLAP queries can be seen in Figure 5.7. The OLTP queries scale slightly worse in the mixed workload as when running in the OLTP-only workload but with an overall decrease in performance: While the system reaches around 125k Tps when running the OLTP workload it is only able to handle 88.5k Tps in the mixed workload on 4 storage server / processing clients. The hit in a 1 server/client configuration is less significant, dropping from 29k Tps to 25.5k Tps.

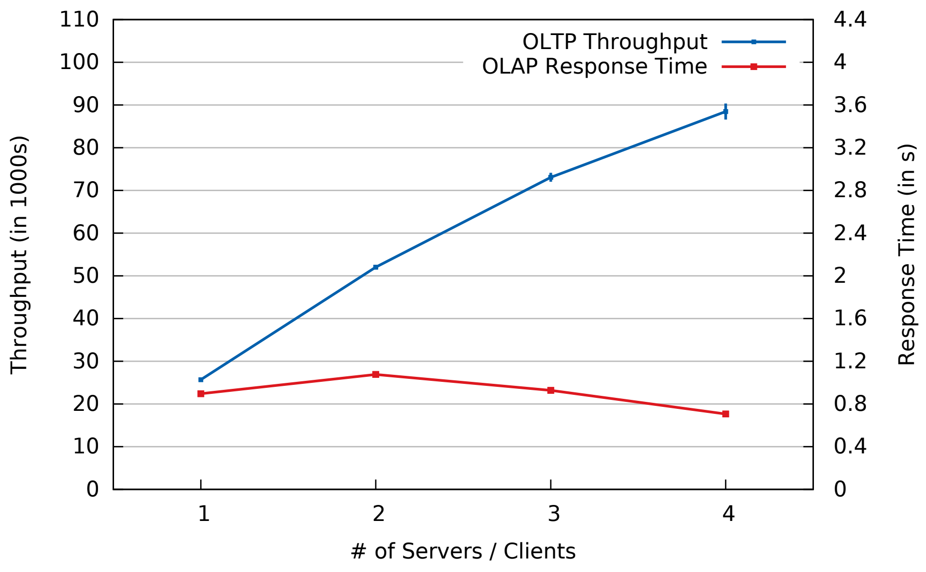

Figure 5.7: Scale-Out for 1 to 4 storage server / processing clients when running the mixed workload

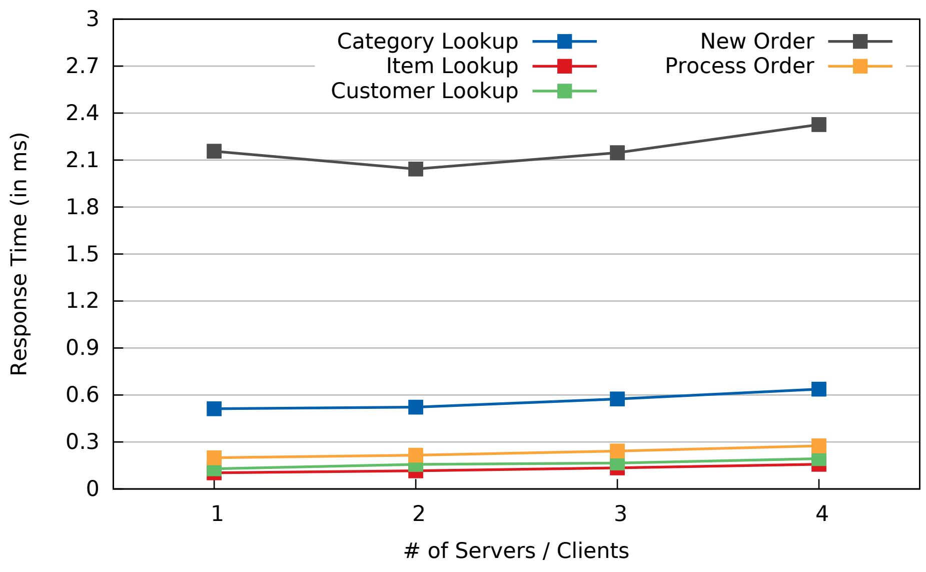

Figure 5.8: Scale-Out for 1 to 4 storage server / processing clients showing response times for individual queries when running the mixed workload

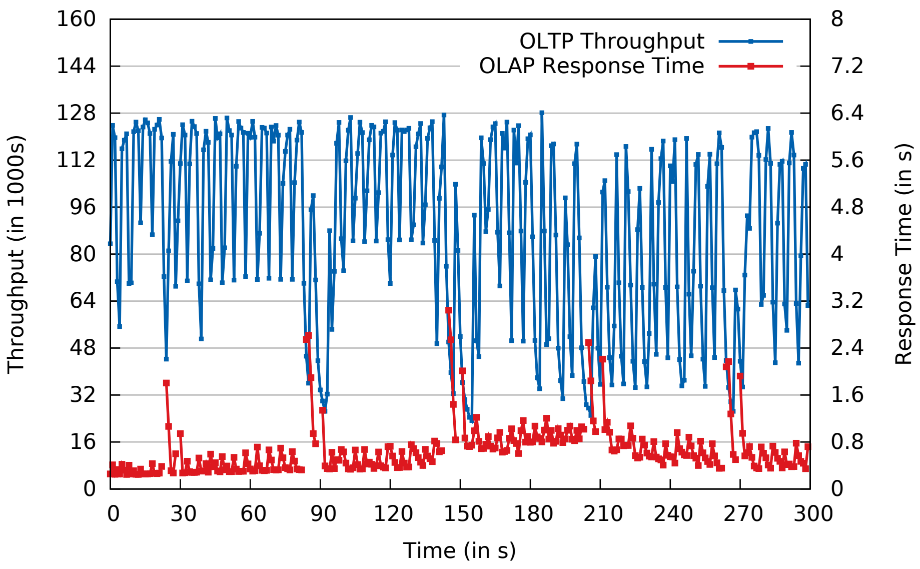

Figure 5.9: OLTP throughput and OLAP response time during one run with 4 storage servers and 4 processing clients

The response times of the individual queries as shown in Figure 5.8 paints the same picture compared to running the queries in the OLTP-only workload — only slower by a few percent.

Interestingly, the response time of the OLAP queries decreases when scaling from 1 to 4 machines but this is due to the slightly worse scale-out of the OLTP queries: While every client executes 25.5k Tps for the configuration with 1 storage server / processing client when scaling to 4 server/clients every client is only able to execute 22k Tps (for a total of 88 Tps) — a decrease of slightly more than 10%. As a result less data is generated by every client, the size of the *orderline* table on each storage server shard decreases which lets the scan complete faster as it has to scan over a smaller amount of data. This undermines the difficulty of designing reliable and meaningful mixed-workload benchmarks as a change in behavior of one workload results in a change of behavior in the other workload.

The average throughput per second and response time suffered from a high deviation. Looking at the throughput/response-time values over time in Figure 5.9 when running the benchmark in a configuration of 4 storage server / processing clients reveals the instability of the system. The throughput oscillates between its maximum of nearly 130k Tps and a minimum of around 90k Tps every 5 seconds in addition to the garbage-collection drop every 60 seconds as seen in the OLTP workload. The reason for this can be found in the *Clean-Orders* transaction which is running every 5 seconds and takes about 2 to 3 seconds to complete its run. During this time the transaction is issuing delete operations en-masse as it has to cope with the amount of data generated by 4 clients. As the delete operations are handled by the network threads in the storage server these operations are contending for resources with the operations issued by the OLTP workload.

Clearly the *Clean-Orders* transaction has a huge impact on the performance but omitting the transaction from the query mix results in the problem described at the beginning: Over time the OLTP transactions increase the amount of data in the system and in turn the OLAP queries have to process more and more data. Figure 5.10 shows the throughput per second for OLTP

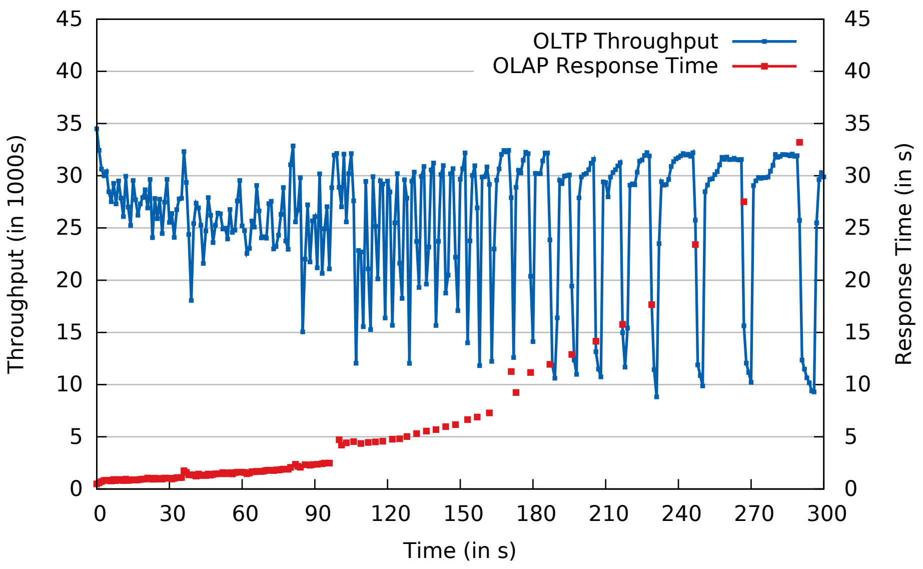

Figure 5.10: Throughput of OLTP and response time of OLAP queries during one run with 1 storage server and 1 processing client (*Clean-Order* transaction disabled and without the 3 minutes warmup-period)

queries and response time of the OLAP query when running on 1 storage server / processing client without the *Clean-Orders* transaction in the mix. In the beginning the OLTP throughput starts as high as 35k Tps but then decreases as the system warms-up. After about 100 seconds the response time spikes by a factor of 2 and throughput starts to drop as the system starts to overload. This overload happens in two different ways amplifying each other: The amount of data continually increases resulting in a longer scan duration, because the scan takes longer more queries are batched together resulting in an even longer response time as more data and more queries have to be processed at the same time.

The regular drops in OLTP throughput can be again attributed to garbage collection: The *New-Order* transaction inserts a number of *orderline* items and *Process-Order* updates the previously written items leaving the previously inserted records as garbage — older elements are never touched again. As such those log pages have to be garbage collection once and remain compacted for all following scans. The pages still containing the garbage are mostly located in the front of the log and as such are processed and garbage collected right in the beginning of the scan leading to the drop in the beginning right after a new scan starts and the slow recovery of OLTP performance afterwards.

5.2.5 Discussion

The benchmarks using the different workloads proved to produce some interesting results. The OLTP workload was able to scale linearly with the number of storage/processing nodes and peaked at around 125k transactions per second. Response time was generally very low with many queries requiring less than one millisecond to complete. Throughput over time was constant except for small drops in performance as the garbage collection was running in the background. The current garbage collection algorithm is designed to run together with scans thus a different (more efficient) garbage collection algorithm is required in the absence of any scans. The OLAP workload also performed quite well when scanning over the log with being able to process over 24 million records per second when data was compacted and static. When executing queries together in a batch the amount of time spent evaluating each query dominated the overall scan time, as such alternative strategies for query evaluation need to be evaluated.

This chapter also showed that designing a benchmark modeling a mixed workload is much more difficult than designing a benchmark for OLTP or OLAP alone: Benchmarks with OLAP workloads require a constant database size to produce meaningful results while to model a real-life OLTP workload the insertion of more data is important. Trying to fix the size of the dataset by regularly pruning it from old data has a huge impact on OLTP performance as the system has to process the deletions. On the other hand, omitting this pruning-mechanism leads to a significant drop in OLAP performance and in the worst case even in system overload.

This is especially true for implementations that are designed for high OLTP throughput as is the case with Log-Structured memory. A common column-store implementation might be affected in a different way by this interaction between the workloads as those systems often favor OLAP performance over OLTP performance.

As such a more suitable benchmark has to be devised that takes these interactions into account: The load put on the system under test by the different workloads needs to be generated in a more controlled way to produce metrics that are comparable between different systems. A future benchmark has to prevent one workload from massively influencing the performance of the other workload. The OLAP performance metric also must take into account the increase of the database size over time and adjust the metric depending on the this rate. At the time of this thesis research in this area remained scarce with the most notable development being the CH-benCHmark[3] which takes the OLTP throughput and OLAP response time into account when calculating its metrics. But the proposed benchmark still leaves many parameters uncontrolled like the number of active OLTP and OLAP streams.

Chapter 6

Conclusion

The evaluation of the system has shown that it is possible to implement a fast scan in a log-structured memory data store while also retaining the high random-access performance. The novel log-based scan approach is able to process 12.6M records per second on one storage server alone. It is twice as fast as a scan over the hash table when no garbage is in the log and faster by a factor of 1.5 when the log is heavily fragmented. The execution model based on user-level threads and futures allows the client to process many transactions in parallel while other transactions are waiting for data. The network stack keeps synchronization overhead between OS threads to a minimum and with the transparent message batching it is able to utilize the Infiniband network to its fullest. One processing node is able to reach a read-throughput of 700k records per second and CPU core from a single storage server. The scan also benefits from this architecture allowing it to quickly transfer the data to clients.

Unfortunately the mixed-workload benchmark still has some issues that need to be fixed: While the system shows excellent OLTP performance, measuring system performance with a mixed workload turned out to be troublesome. The different designs of existing benchmarks for pure OLTP and pure OLAP workloads can not be easily combined to test systems in a mixed workload. The interplay between OLTP and OLAP workloads has a significant impact on each others performance making it difficult to measure e.g. OLAP performance in a system that is generally strong in OLTP workloads. This is still an open research problem which has not been solved.

6.1 Future Work

The system still has many parts requiring further work: There is no replication support and a crash in the storage node leads to data loss as it is only stored in DRAM. The query evaluation framework for processing the scans consumes a significant part of the total scan run time when evaluating multiple queries at once. The garbage collection algorithm needs improvements for tables that are not regularly scanned as the current algorithm produces a noticeable hit in OLTP performance when run in the background. The commit manager service needs an implementation for scaling out to multiple instances and the current snapshot descriptor data structure requires a compression mechanism when a long running transactions prevents the version bitmap to "roll forward". The Bd-Tree can further benefit from a deeper integration with the execution model to batch modifications from different transactions into one operation to decrease the required amount of node writes.

Bibliography

[1] Philip A. Bernstein and Nathan Goodman. “Concurrency Control in Distributed Database Systems”. In: *ACM Comput. Surv.* 13.2 (June 1981), pp. 185–221. ISSN: 0360-0300. DOI: 10.1145/356842.356846. URL: http://doi.acm.org/10.1145/356842.356846.

[2] Lucas Braun et al. “Analytics in Motion: High Performance Event-Processing AND Real-Time Analytics in the Same Database”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: ACM, 2015, pp. 251–264. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742783. URL: http://doi.acm.org/10.1145/2723372.2742783.

[3] Richard Cole et al. “The Mixed Workload CH-benCHmark”. In: *Proceedings of the Fourth International Workshop on Testing Database Systems*. DBTest ’11. Athens, Greece: ACM, 2011, 8:1–8:6. ISBN: 978-1-4503-0655-3. DOI: 10.1145/1988842.1988850. URL: http://doi.acm.org/10.1145/1988842.1988850.

[4] Keir Fraser. “Practical Lock-Freedom”. PhD thesis. University of Cambridge, 2004. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.html.

[5] H. Gao, J.F. Groote, and W.H. Hesselink. “Lock-free dynamic hash tables with open addressing”. English. In: *Distributed Computing* 18.1 (2005), pp. 21–42. ISSN: 0178-2770. DOI: 10.1007/s00446-004-0115-2. URL: http://dx.doi.org/10.1007/s00446-004-0115-2.

[6] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. “SharedDB: Killing One Thousand Queries with One Stone”. In: *Proc. VLDB Endow.* 5.6 (Feb. 2012), pp. 526–537. ISSN: 2150-8097. DOI: 10.14778/2168651.2168654. URL: http://dx.doi.org/10.14778/2168651.2168654.

[7] Timothy L. Harris. “A Pragmatic Implementation of Non-Blocking Linked-Lists”. In: *Proceedings of the 15th International Conference on Distributed Computing*. DISC ’01. London, UK, UK: Springer-Verlag, 2001, pp. 300–314. ISBN: 3-540-42605-1. URL: http://dl.acm.org/citation.cfm?id=645958.676105.

[8] A. Kemper and T. Neumann. “HyPer: A hybrid OLTP and OLAP main memory database system based on virtual memory snapshots”. In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. Apr. 2011, pp. 195–206. DOI: 10.1109/ICDE.2011.5767867.

[9] Justin Levandoski, David Lomet, and Sudipta Sengupta. “LLAMA: A Cache/Storage Subsystem for Modern Hardware”. In: *Proceedings of the International Conference on Very Large Databases, VLDB 2013*. VLDB – Very Large Data Bases, Aug. 2013. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=232415.

[10] Justin Levandoski et al. “High Performance Transactions in Deuteronomy”. In: Conference on Innovative Data Systems Research (CIDR 2015), Jan. 2015. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=232391.

[11] Simon Loesing et al. "On the Design and Scalability of Distributed Shared-Data Databases". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 663–676. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2751519. URL: http://doi.acm.org/10.1145/2723372.2751519.

[12] Maged M. Michael. "Hazard pointers: safe memory reclamation for lock-free objects". In: *Parallel and Distributed Systems, IEEE Transactions on* 15.6 (June 2004), pp. 491–504. ISSN: 1045-9219. DOI: 10.1109/TPDS.2004.8.

[13] Maged M. Michael. "High Performance Dynamic Lock-Free Hash Tables and List-Based Sets". In: *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '02. Winnipeg, Manitoba, Canada: ACM, 2002, pp. 73–82. ISBN: 1-58113-529-7. DOI: 10.1145/564870.564881. URL: http://doi.acm.org/10.1145/564870.564881.

[14] John Ousterhout et al. "The Case for RAMCloud". In: *Commun. ACM* 54.7 (July 2011), pp. 121–130. ISSN: 0001-0782. DOI: 10.1145/1965724.1965751. URL: http://doi.acm.org/10.1145/1965724.1965751.

[15] Chris Purcell and Tim Harris. "Non-Blocking Hashtables with Open Addressing". In: *DISC 2005: Proceedings of the 19th International Symposium on Distributed Computing*. A longer version appears as technical report UCAM-CL-TR-639. Sept. 2005. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=67422.

[16] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. "Log-structured Memory for DRAM-based Storage". In: *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. Santa Clara, CA: USENIX, 2014, pp. 1–16. ISBN: ISBN 978-1-931971-08-9. URL: https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble.

[17] Ori Shalev and Nir Shavit. "Split-Ordered Lists: Lock-free Extensible Hash Tables". In: *J. ACM* 53.3 (May 2006), pp. 379–405. ISSN: 0004-5411. DOI: 10.1145/1147954.1147958. URL: http://doi.acm.org/10.1145/1147954.1147958.

[18] Patrick Stuedi et al. "DaRPC: Data Center RPC". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. Seattle, WA, USA: ACM, 2014, 15:1–15:13. ISBN: 978-1-4503-3252-1. DOI: 10.1145/2670979.2670994. URL: http://doi.acm.org/10.1145/2670979.2670994.

[19] Transaction Processing Council (TPC). *TPC Benchmark C (V5.11)*. 2010. URL: http://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-C_V5-11.pdf.

[20] Daniel Widmer. "Real-Time Analytics in a High Volume Event Processing System". MA thesis. ETH Zurich, Sept. 2013. URL: http://systems.ethz.pubzone.org/pages/publications/showPublication.do?publicationId=2575781.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

EFFICIENT SCAN IN LOG-STRUCTURED MEMORY DATA STORES

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
BOCKSROCKER	KEVIN

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 09.09.2015	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.