Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 148 b

Systems Group, Department of Computer Science, ETH Zurich

Elasticity in Tell

by
Nikolas Göbel

Supervised by
Lucas Braun, Markus Pilman, Donald Kossmann

Feb. - Aug. 2016

inf | Informatik
Computer Science

Contents

1 Abstract

Tell is a novel, distributed database aiming to support fast analytical queries on rapidly changing datasets. To achieve this goal, Tell deconstructs the traditional database architecture into three parts: A key-value record store with support for fast, distributed scan-operations, an accompanying application library providing query and transaction interfaces and a commit-manager serializing transactions.

Before this work, the storage layer could only support a fixed and known number of storage nodes, due to the use of a simple, modulo-based partitioning scheme. Adding or removing nodes during cluster operation without a complete re-partitioning of all keys would have resulted in most of the clusters data becoming inaccessible. The goal of this thesis was to explore the design space of more flexible partitioning schemes, to extend Tell with a promising design and to evaluate the implementation.

We obtain a lock-free system based on the method of consistent hashing that distributes data well, while allowing us to add and remove storage nodes during normal operation. Node additions and removals have no impact on availability and the cluster returns to normal operation after a few seconds. Our design introduces marginal performance overhead in each storage node, but the current implementation still only performs at about 50-60% of non-elastic Tell's throughput in the TPC-C benchmark. We attribute this to a combination of missing optimizations (not yet implemented or disabled due to incompatibility), concurrency-related bugs and problems with distributed indexes and we expect to be able to fix these problems in the near future.

2 Overview

In chapter 3 we will establish a basic understanding of the current systems shortcomings and the exact problem that we will be trying to solve. There we will see, that our overall solution will consist of two main components: A flexible partitioning algorithm and a rebalancing protocol that will transfer keys across nodes, to enforce the most recent partitioning.

Chapter 4 is concerned with the design of such a flexible partitioning scheme, first identifying important criteria and evaluating possible algorithms against them. Consequently, in chapter 5 we will devise the protocols for adding and removing storage nodes. We will again proceed by first identifying critical properties and security guarantees that need to be provided, before evaluating our designs against them.

The sections on implementation (chapter 6) will cover the core operations of the partitioning algorithm and provide implementations in pseudo code. We will also discuss more advanced topics like caching, the use of virtual nodes and how we worked with Tell-specific subsystems like the LLVM code generator.

Finally in chapter 7, we will evaluate the overall system. For this we analyze the distribution of data across the cluster and compare it to the static, Modulo-based partitioning scheme. We provide results of running TPC-C with storage

node additions and removals and compare overall performance to non-elastic Tell.

3 Problem Statement

Tell's storage layer has to maintain a set of key-value pairs on n nodes in a cluster, where the size of the dataset generally exceeds any single machine's main storage. Therefore, the keys have to be distributed onto the available nodes, according to some partitioning scheme. All keys distributed onto a particular node are said to be *owned* by that node.

The simplest possible partitioning schemes are *fixed*, meaning that they work with a number of nodes that is assumed to be constant. With a fully fixed scheme, the partitioning can be precomputed and distributed to each node in a perfectly safe manner, as it will never change. Probably most common among the fixed partitioning schemes is the modulo operation: $owner(key) = key \mod n$. This scheme is easy to implement, extremely cheap to compute and the only information that has to be distributed is n. Like many nascent distributed systems, Tell made use of this scheme.

Though easy to implement and efficient, fixed schemes come with a set of drawbacks. A fixed-size cluster cannot scale to accommodate larger datasets or to relieve nodes under heavy load. If any node fails (*unplanned failure*) or is shut down by an operator (*planned failure*), its data in the cluster becomes inaccessible. We therefore require a system that can adapt well to changes in n.

In theory, every fixed scheme could of course be made flexible, by recomputing the full partitioning on every change of n and redistributing all keys accordingly. Moving keys between nodes, in order to match a partitioning is known as *rebalancing*. As we will see in section 4.2.1, modulo-based partitioning requires rebalancing most of the data in the cluster whenever n changes. Moving large amounts of data is expensive and will impact response times negatively. Additionally, maintaining consistency and availability during a rebalancing is a non-trivial problem (especially doing so in a lock-free way) and incurs performance overheads. Therefore minimizing the number of keys that require rebalancing after a node addition or removal is very desirable.

The ability to dynamically increase or reduce the size of the cluster during normal operation, without serious impact on neither availability nor response times and with no loss of consistency is known as *elasticity*. This work explores ways to make Tell elastic. Specifically, we aim to achieve two goals:

1. It should be possible to add a storage node during normal cluster operation. This should relieve some of the existing nodes from some of their load and increase the overall capacity of the cluster.

2. Planned removal of nodes should be possible without data loss. Removing an existing storage node from the cluster should move all its data onto remaining peers.

We will have to design and implement a flexible partitioning scheme, that allows for changes in n that are accompanied only by partial rebalancing operations. Once this is achieved, we will have to develop a safe and efficient rebalancing operation, that is tailored to the specifics of Tell.

The following section of this thesis explores the design decisions that have to be made, the choices comparable systems made and sensible choices for Tell itself.

4 Designing a Partitioning Scheme

4.1 Partitioning Scheme Properties

correctness
: The most important property of any partitioning scheme is of course *correctness*, i.e. the guarantee, that keys are distributed in a deterministic way. This is what allows us to determine the node that will be able to serve a request. A non-deterministic partitioning scheme is useless, as we cannot retrieve keys once they have been assigned to an owner.

stability
: Next is the expected number of keys that have to be rebalanced, whenever the cluster's composition changes. We will call this property *stability*.

uniformity
: Another interesting property is *uniformity* of the resulting key distribution. Here we cannot strictly argue for uniform distributions to be "better" than uneven ones, as we will see in our discussion of some popular schemes. Non-uniform distributions usually provide data locality advantages, but lead to so called *hot-spots*, nodes receiving a disproportionately high amount of the overall system load.

replication-readiness
: In order to accommodate replication efforts, a partitioning scheme should be able to determine any number of distinct replica nodes for a key.

statelessness
: *Statelessness* is useful to consider as well, as it will reduce the complexity of the implementation. A partitioning scheme is considered stateless, if the owner of a key is fully determined by the key.

awareness
: The extent to which a scheme can make use of environmental information we will call *awareness*. An algorithm with high awareness could aim to choose replicas according to rack-layout or to adapt its operation to the actual load distribution on the cluster. In high-availability environments it is, for example, crucial, to place replicas in separate data-centers, to secure data against catastrophic events.

control

By *control* we refer to the degree to which the algorithm can be tweaked by an operator. A common manual intervention is the alleviation of load from a specific node, by redistributing some of its data. This can be useful to optimize a cluster for expected load, short-termed load spikes or to assist in maintenance. It can also be used to make up for lacking awareness in the partitioning algorithm itself.

performance

Finally, the partitioning algorithm will have to be run for every single request, so we have to be very aware of its *performance* characteristics. These are both, the actual running times and the time complexity with increasing number of nodes.

4.2 How to determine the owner for a given key?

As we have established, the partitioning scheme assigns each key to a node in the cluster. We call this node the key's *owner*. The owner stores the key and its associated value and serves all read and write requests on this key. In strongly consistent environments, the owner will also be coordinating the replication of its keys. We have already seen the impracticality of modulo-based partitioning for our use-case. Nevertheless, we will evaluate it against the criteria identified in the previous section, in order to obtain a baseline.

Afterwards we will discuss the three methods commonly found in distributed systems comparable to Tell: *range-based partitioning*, *consistent hashing* and *highest-random-weight hashing*.

4.2.1 Modulo-based Partitioning

The algorithm $owner(key) = key \mod n$ is deterministic, the same key will always be assigned to the same node. *Correctness* is therefore guaranteed. Replica nodes for a key can be easily identified by a slight extension of the scheme $replica(key, i) = (key + i) \mod n$. If more than i nodes are available in the cluster, these replicas are guaranteed to be distinct.

One of the main, motivating factors behind this work is the instability of modulo-based partitioning. With every change in n, the modulo value for most keys changes, so they have to be rebalanced. This is the scheme's crucial weakness.

Modulo-based partitioning achieves a perfectly uniform distribution on both, continuous key ranges and keys chosen uniformly at random, as we will later see in figure 7. Still, there are insert patterns that can lead to hot-spots. E.g. the key series $key_i = (1 + i * n)$ will be exclusively owned by $node_1$. These kinds of patterns might rarely occur in practice, but as we will see, other schemes are completely immune to this problem.

The only state besides the key required for the scheme to work is n. When considering only the implementation complexity of the algorithm, this is very

desirable. Modulo-based partitioning makes absolutely no use of any environmental information. Operators have no control over the algorithms behavior, short of manipulating node ids (for example to guarantee replicas are placed across racks). Performance of the algorithm is very favorable, as both, time and space complexity are constant in the number of nodes in the cluster.

4.2.2 Range-based Partitioning

In range-based partitioning, the entire key-space is partitioned linearly into ranges, which are then assigned to different nodes. A key is owned by the node that owns the range containing the key. This technique is famously used by Google's BigTable system [8]. As long as the mapping of ranges to nodes is deterministic and consistent across the cluster, the whole scheme is deterministic, as a key will always belong to the same range. Determining replicas for a given range can't follow any simple algorithm, replicas have to be managed separately in the mapping from ranges to nodes. Lots of state might be required to maintain the mapping, especially for ranges of variable size.

Re-balancing happens on a per-range level, therefore the minimum size of a range determines the granularity at which rebalancing can happen and consequently the stability of the scheme.

Partitioning solely based on key ranges is highly susceptible to *hot-spots*, nodes receiving a disproportional amount of the overall system load. This is because for monotonic keys like timestamps, more recent ranges tend to be used much more frequently than older ones. A uniform load distribution can only be achieved with keys chosen uniformly at random.

Adding new nodes might do very little to alleviate existing nodes under load. Consider $node_a$, responsible for ranges R_1, R_2. Now assume there is a load hot-spot on range R_1. Adding a new node $node_b$ to the cluster will not affect the hot-spot, as $node_a$ will either give up R_1 entirely, thereby merely moving the hot-spot to $node_b$, or R_2, which was not under heavy load anyway.

BigTable itself tries to solve this problem by splitting ranges that have grown to big or are under too much load into sub-ranges. Those can then be distributed separately onto multiple nodes. While this reduces the impact of hot-spots, a range-split is not easy to coordinate in a distributed system, as the new responsibilities must be propagated atomically and consistently throughout the whole cluster.

Ranges do offer some advantages with regards to data locality. Since the ordering of keys is preserved, keys close to each other tend to be stored on the same machine, which can improve scan performance in some cases.

By default, range-based partitioning makes no use of environmental information, but operators retain a great amount of control over the algorithm's behavior. The uneven load distribution, for example, can be used to operational advantage in heterogeneous environments, by moving less popular ranges onto physically weaker nodes.

Performance of the scheme is heavily dependent on the data structure used to maintain the mapping from ranges to nodes. BigTable uses a multi-level

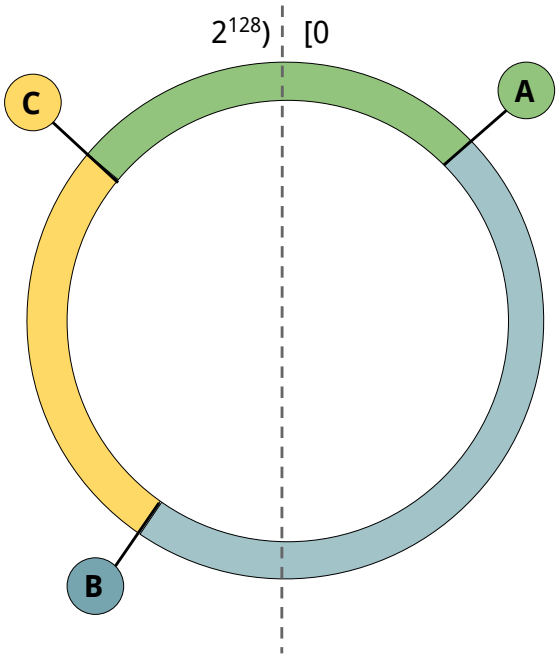


Figure 1: A wrapping range of 128-bit tokens

mapping and aggressively caches lower layers. Therefore a practical level of performance can definitely be achieved, even at massive scale. Most of the more recent architectures don't offer range-based partitioning, or only in combination with other techniques.

4.2.3 Consistent Hashing

The method of consistent hashing is due to Karger et al.[2] and was famously used in Amazon's DynamoDB [9]. At it's core, the idea is to hash both, nodes and keys, with the same hash function, such that they map onto the same, wrapping range of integers (the key-space), forming a hash ring. See figure 1 for an example. Nodes therefore require some kind of token that can be hashed. This token can either be generated randomly or assigned by an operator.

To find the owner for a given key, we compute it's hash value and find the node that maps closest to it in the key-space.

Consistent hashing guarantees, that on average, only $O(1/n)$ of all keys have to be rebalanced whenever a node is taken out of or added into the cluster. Intuitively, this can be seen in figure 1. A new node inserted between nodes A and B would only have to retrieve keys from B. If every node (in expectation) is responsible for $1/n$ of all keys and a new node will only retrieve keys from

one neighbor, then it can, at most, receive $1/n$ of all keys[1]. This is a vast improvement over the full rebalancing required by modulo-based partitioning.

Replicas can be easily assigned in a deterministic way as well, simply by walking the hash ring starting from a key's owner and returning the next i distinct nodes encountered. If again more than i nodes are available, it is highly likely that any (virtual-)node's neighbors will be distinct.

Depending on the hash function used, nodes will be mapped randomly to positions in the hash ring. This leads to a highly non-uniform distribution, since the size of each nodes key range is then essentially random as well. Similarly, removing a node from a well balanced cluster results in hot-spots on its neighbors in the hash ring. Without the use of an order-preserving hash function, consistent hashing will not preserve the order of keys. This is in stark contrast to e.g. range-based partitioning.

On the upside, consistent hashing with a well-mixing hash function is immune to degenerate insert patterns (like the one discussed in our analysis of modulo-based partitioning in section 4.2.1), because the partitioning key is effectively randomized.

The most common solution to smoothing the distribution of keys is to assign each physical node not just one, but many tokens in the hash ring. The number of tokens to assign can be fixed (typically a value around 200 is chosen[2]). These tokens are then called *virtual nodes*. In informal terms, this means that every physical node is responsible for multiple smaller partitions scattered throughout the key-space instead of a single, continuous one. Statistically, the total partition size for each physical node will even out. This is depicted in figure 2.

Consistent hashing requires a lot less state than range-based partitioning, as a complete view of the hash ring can be constructed locally from just the node tokens. Those still have to be propagated consistently, a problem that will be of primary concern to us in later sections.

A random mapping also means that consistent hashing, by default, is completely oblivious to heterogeneity in node performance, differences in interconnection speed or other environmental factors. But in situations were more direct control over load distribution is desired, e.g. in environments where the load on each node can be monitored and fed back into the consistent hashing algorithm, virtual nodes can be eschewed in favor of manual placement of nodes. Tokens of idling nodes can be chosen to map closer to busy ones in the hash ring, thereby relieving the latter from some of their responsibilities. Thus operators retain full control over the distribution and can manually intervene to fix hot-spots.

Finding the owner for a given key can be done in $O(\log n)$ time, by using a sorted map. n is again the number of physical nodes in the cluster. Inserting and removing a node requires an evaluation of the hash function for each virtual node, which is still constant in n.

Besides DynamoDB, many other popular systems like Riak[10] and Cassandra[5] make use of consistent hashing, the latter notably in combination with an order-

[1] A much more in-depth explanation can be found in [2].

[2] As recommended by e.g. Cassandra.

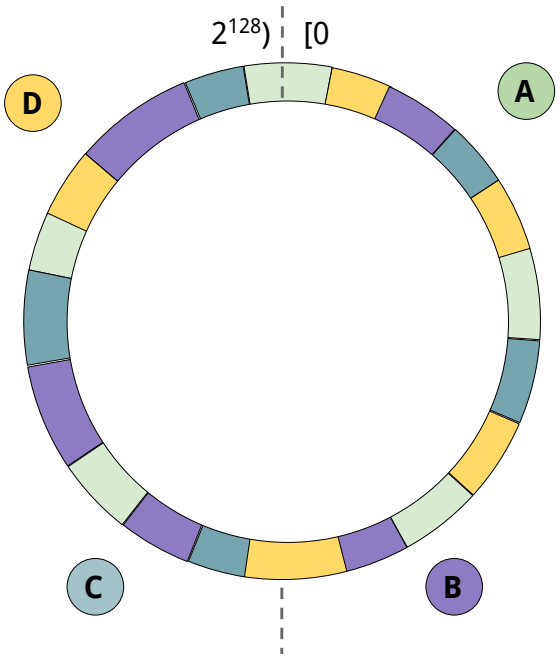


Figure 2: A hash ring with 5 virtual nodes per physical node

preserving hash function to regain some data locality. The Chord distributed hash table makes use of a modified version of the same principle [11].

4.2.4 Highest Random Weight Hashing

Highest Random Weight Hashing (HRW), introduced in [3] and also known as *Rendezvous Hashing*, provides a solution to a more general consensus problem for n sites. Its application to partitioning is merely a special case of this problem. A key's owner is determined by computing n hash values from n different hash functions $h_i(key)$, and picking the node id i, whose hash function produced the largest hash value.

HRW provides the same guarantee on the expected percentage of keys that require rebalancing, $O(1/n)$.

Replicas can be determined easily, by always choosing the next largest hash value produced by a distinct node.

Unlike consistent hashing, HRW distributes data uniformly by default. The uniform distribution is even maintained in the face of node removals. No additional measures like virtual nodes are required.

Again, HRW requires a lot less state than range-based partitioning, as all decisions can be made from just set of n hash functions used. As with consistent hashing, changes in those have to be communicated consistently.

Naively implemented, finding the owner for a given key has time complexity $O(n)$ and should therefore perform worse than consistent hashing. Since n is usually rather small in practice, the difference does not matter as much. A more involved implementation is described in [4], that has time complexity of $O(\log n)$.

4.2.5 Conclusion

The complexities associated with range-based partitioning have made it very unattractive in comparison to hashing techniques, as they provide most of the benefits of ranges, with less management overhead. Consistent hashing in particular allows operators to adapt to heterogeneous environments in much the same way that ranges do, simpler and with greater flexibility. Loss of fine-grained operational control was also the big drawback of HRW when compared to consistent hashing. With the latter, operators could still manually select node tokens to influence placement in the hash ring. No such influence is possible with HRW. For this reason, most real-world systems choose consistent hashing over HRW. We will do so as well.

Since Tell scans are designed to run across all nodes in parallel, with multiple queries batched together, loss of data locality should not affect scan performance. This drawback is even less significant when compared to the modulo-based partitioning Tell used before, as locality is also destroyed by the modulo operation. Distributed scans also favor a highly uniform distribution of keys onto nodes, such that each node can contribute equally to a scan and no node

becomes a bottleneck. Consistent hashing (when employing virtual nodes) can provide and maintain such a uniform distribution.

4.3 Where will partitioning be handled?

A Tell cluster is made up of three kinds of nodes: a storage layer, a processing layer and the commit-manager. We will have to decide which layer will be handling what part of the overall scheme. Tell's own implementation of Multi-version Concurrency Control (*MVCC*), for example, utilizes cooperation between all three layers to great effect. The commit-manager was the natural choice for maintaining the current partitioning. Storage nodes will coordinate with the commit-manager and request keys from each other. The processing nodes will be extended to provide crucial safety guarantees during a key transfer.

4.4 Bootstrapping and distributing partitioning state

Before this work, Tell had no existing component to maintain and distribute cluster-state. For example, storage node address information had to be provided manually to every processing node. We therefore decided to extend Tell's commit-manager with a node registry. Storage nodes register themselves with the commit-manager on start-up. Processing nodes can then query the commit-manager for address information. More importantly, since we are now working with an elastic cluster, processing nodes can re-fetch information from the registry, whenever the number of nodes in the cluster changes. We also allow processing nodes to register themselves with a tag, indicating the benchmark or application they are running. Tell clients can then query the commit-manager for address information of the processing nodes they are interested in.

We considered implementing the node registry as a Tell table hosted by the commit-manager. This way, the registry information would have benefited directly from any future replication solution. This would have required extensive changes to the commit-manager, as it was not designed to act as a storage node. Since in Tell's current architecture, the commit-manager is a single point of failure anyway, we decided that neither durability nor availability of the registry were more important than ease and simplicity of implementation for the time being. This option can nevertheless be explored, should we move to a replicated commit-manager in the future.

5 Protocol Design

As stated in chapter 2, we will now design protocols for adding and removing storage nodes, that will govern the rebalancing operations involved. Again, we will first consider properties that any such protocol must provide.

First and foremost, increasing or reducing the size of the cluster must be *safe*. This means, that no data may be lost or corrupted during a rebalanc-

ing operation. Although some increase in response times during a rebalancing operation is to be expected, we would prefer for the protocols to not interrupt cluster availability beyond that and to quickly restore normal operation. Reads and writes on all keys must continue to be served throughout the protocols execution. To keep with the rest of Tell, we would also like all protocols to be *lock-free*, meaning that at least one node is guaranteed to make progress at any given time (although other nodes might get starved).

The design process was a team effort. All designs were influenced and refined by many long discussions with my advisors Lucas Braun and Markus Pilman.

5.1 Error Scenarios

We start by considering some variations on common error scenarios.

write after transfer
: Say $node_{src}$ is transferring partition $P := \{..., (key^*, x), ...\}$ to $node_{dest}$. Assume the transfer is still in progress and the tuple (key^*, x) has already been transferred to $node_{dest}$. Now processing node *proc* sends $write(key^*, x')$ to $node_{src}$. The new owner of P, $node_{dest}$ has not seen this write and it is therefore lost to all processing nodes reading from $node_{dest}$ in the future.

read before transfer
: Assume again, that $node_{src}$ has started the transfer of P to $node_{dest}$ and the transfer is still in progress. The tuple (key^*, x) has *not yet* been transferred. If now processing node *proc* sends $read(key^*)$ to the new owner of P, $node_{dest}$, the request will fail, as key^* is not yet available at $node_{dest}$.

phantom write / write before transfer
: Special care has to be taken during the transfer, to never overwrite local data at $node_{dest}$. Assume that the tuple (key^*, x) has not yet been transferred. Now processing node *proc* sends $write(key^*, x')$ to the new owner of P, $node_{dest}$. After the write has finished, the transfer of (key^*, x) completes and the local value, x', is overwritten.

lost update
: If there is a window, during which transactions are still executing on the previous and the new owner concurrently, and two writes to the same key are routed to $node_{src}$ and $node_{dest}$ respectively, the update at $node_{src}$ will be lost. This scenario is somewhat involved, thus warranting an in-depth exploration in section 5.1.1.

starved transfer
: Assume $node_{src}$ will only begin transferring P once local write activity on P has ceased, for example to prevent the *write after transfer* scenario. But this might never be the case, so P might actually remain at $node_{src}$ indefinitely.

perpetual transfer

Consider the case where $node_{src}$ starts the transfer of P immediately and resends any tuples that changed due to local write activity (again as a measure to prevent *write after transfer*). Now the transfer might never end, if write activity at $node_{src}$ doesn't end.

starved client

All of the above could be trivially avoided, by delaying or blocking all reads or writes on P until the transfer has finished. This would of course lead to starvation in the processing nodes.

5.1.1 The Lost Update Problem

For this problem, the actual key transfer is irrelevant. As we will see in the following section, the transfer will only start, once all transactions on transient partitions at $node_{src}$ have finished. Nevertheless, there exists a window after adding a new node $node_{dest}$, during which some transactions are still executing reads and writes on P (the partition being transferred) at $node_{src}$. Transactions started after $node_{dest}$ joined will be aware of its existence and read and write at $node_{dest}$ instead. The scenario depicted in figure 3 becomes possible.

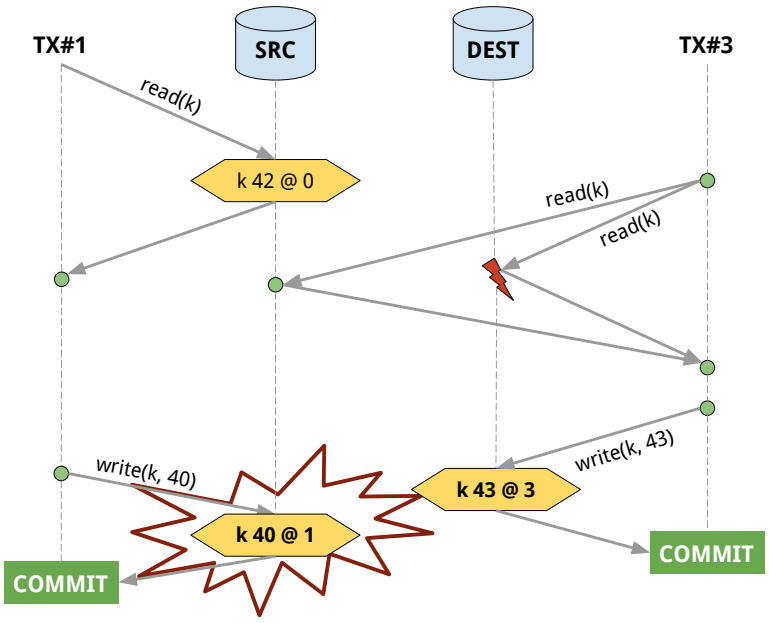


Figure 3: The write(k, 40) by transaction 1 is lost

Transaction 1 is not aware of $node_{dest}$, while transaction 3 is, because it

was started after $node_{dest}$ registered itself with the commit-manager. Therefore transaction 3 will be routing writes on key k to its new owner, $node_{dest}$. As $node_{src}$ is unaware of the more recent $write(k, 43)$ by transaction 3, it will still accept $write(k, 40)$ by transaction 1. Either transaction 1 or transaction 3 should have been aborted.

One possible solution would be to block all writes at $node_{src}$, as soon as $node_{dest}$ registers itself. This would require a mutex in the storage nodes and thus violate the requirement, that our protocol should be lock-free. It would also be a very coarse-grained solution, as all transactions still writing at $node_{src}$ would have to abort, even though they might not be conflicting with any writes at $node_{dest}$ by newer transactions. Equally problematic would be blocking writes at $node_{dest}$ until all outstanding transactions on $node_{src}$ have finished. Instead, we will need to replicate writes at both nodes. [3] How to do so efficiently and without incurring overhead during regular operation will be covered in chapter 5.4. Replicated writes will prevent the problem, as is shown in figure 4.

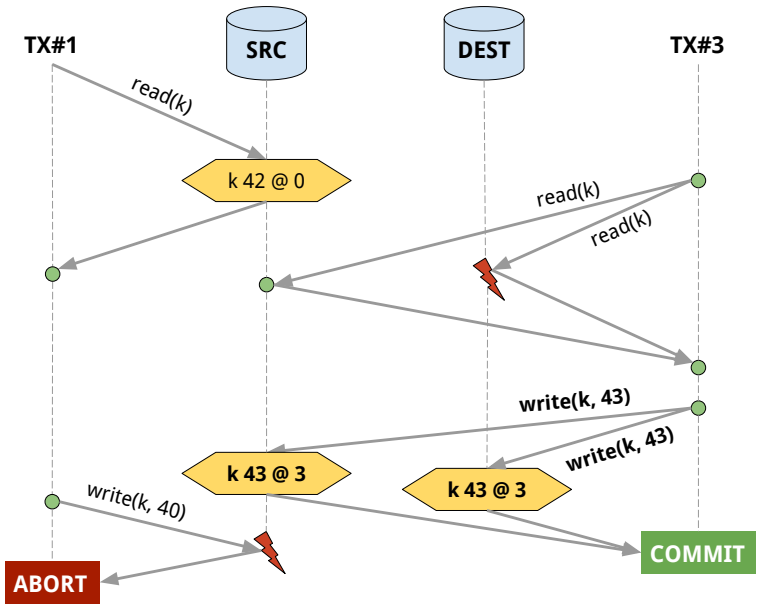


Figure 4: The write(k, 40) by transaction 1 is rejected

Now $node_{src}$ is aware of the $write(k, 43)$ by transaction 3 and will reject the later $write(k, 40)$ by transaction 1. Conversely, if the write by transaction 3 would have reached $node_{src}$ only after transaction 1 had already committed, it

[3] A similar solution will be necessary for reads, as is evident in figure 3.

would have been equally rejected and transaction 3 would have been aborted instead.

5.2 Safety Guarantees

Based on the considerations in the previous section, we will now determine the guarantees, that our final design must provide.

Theorem 5.1. *Let the current owner of a partition P be called $node_{src}$ and the node to which P should be moved called $node_{dest}$. Then the transfer of P may only start, once write activity on P ceases at $node_{src}$.*

Theorem 5.2. *Write activity on a partition P that is waiting to be transferred must eventually cease at its previous owner, $node_{src}$.*

Theorem 5.3. *All reads on a partition P that is being transferred must be tried at both, the previous owner $node_{src}$ and the new owner $node_{dest}$, until all tuples are available at $node_{dest}$.*

Theorem 5.4. *All writes on a partition P that is being transferred must be replicated at both, the previous owner $node_{src}$ and the new owner $node_{dest}$, until all writes on P at $node_{src}$ have been processed.*

Theorem 5.5. *When, during a key transfer, $node_{dest}$ receives a tuple whose key is already present locally with a higher version, the incoming tuple must be rejected.*

Theorem 5.6. *If $node_{src}$ and $node_{dest}$ are both available, then reads and writes on a partition P that is being transferred must be served, even while the transfer is in progress.*

Theorem 5.1 implies that all writes still actively being processed at $node_{src}$ must be handled before the transfer starts. This, by definition, prevents the *write after transfer* scenario. Still, this theorem alone would be prone to the *lost update*, *starved transfer* and *perpetual transfer* scenarios.

Theorem 5.2 implies that once ownership of P changes, subsequent writes must go to the new owner, $node_{dest}$. This ensures that write activity at $node_{src}$ will eventually cease, once all active writes have been processed. Once this is guaranteed, starvation scenarios *starved transfer* and *perpetual transfer* can never occur.

Theorem 5.3 circumvents the *read before transfer* scenario. The read at $node_{dest}$ might still fail, but every key in P is guaranteed to be available at either $node_{dest}$ or $node_{src}$. We pay a small performance penalty, by sending two requests instead of a single one.

Theorem 5.4 directly prevents the *lost update* scenario, as discussed in section 5.1.1 and pictured in figure 4. It also protects against a premature shutdown of $node_{src}$ at the cost of potentially unjustified aborts: If a transaction can't validate it's write on a partition (of which it thinks that it's still bootstrapping) at the old owner, it will abort, as there could have been an earlier transaction

that performed a conflicting write at $node_{src}$ and committed immediately before $node_{src}$ was shut down. Again we incur a small performance penalty for the duplicated request.

Theorem 5.5 prevents the *write before transfer* scenario. Usually, local tuples at the new owner will have a newer version than those at the old owner, as write activity is routed to the new owner, as per theorem 5.2. The additional version check in theorem 5.5 is nevertheless necessary, as there might be situation in which P used to be owned by $node_{dest}$, was then transferred to $node_{src}$ and never garbage collected. If now P is moved back to $node_{dest}$, local tuples in P are still present, that might have lower versions than those from $node_{src}$.

Finally, theorem 5.6 forbids the trivial solution leading to the *starved client* scenario, as all of the above guarantees could be achieved by blocking requests on P until the transfer has been completed. Our challenge is to come up with protocols that not only continue to serve requests, but are completely lock-free.

Taken together, these theorems prevent all of the identified error scenarios. By our use of temporal relations ("in-flight", "subsequent") in these theorems, it already becomes clear, that we want rebalancing to be an atomic operation. Only then can we serialize all requests into "before start of the transfer", "during the transfer" and "after the transfer has started". We will utilize Tell's existing MVCC subsystem to ensure atomicity: All transactions are issued a unique, incrementing version number. Snapshots issued by the commit-manager when starting a new transaction are sent with all requests, containing information about the lowest transaction version that is still being processed. This version is called the *lowest active version* and will play an important role in our protocols.

In the following sections, we will describe the protocols for adding and removing storage nodes, as well as an extended protocol for performing transactions. We will evaluate each protocol against the theorems identified in this section.

5.3 A Protocol For Adding a New Node

With exception of the very first node in a cluster, all further nodes join in an environment, where the key-space is already partitioned among some existing nodes in the cluster. Therefore adding a node is essentially a re-partitioning operation followed by a transfer of keys from their previous owners to the new one. We refer to the joining node as $node_{dest}$.

1. Upon joining, $node_{dest}$ registers with the commit-manager inside of a transaction. This is important, as it establishes the version x, from which point on the new partitioning will be respected by all processing nodes.

2. The commit-manager updates its hash ring and uses it to determine the partitions $P_1, ..., P_v$ that $node_{dest}$ will be responsible for. Here v is equal to the number of tokens (or virtual nodes) per physical node. We will call the nodes currently owning these partitions the *sources*. The commit-manager marks $P_1, ..., P_v$ as *bootstrapping*.

3. $node_{dest}$ contacts all sources and requests schema information from them. It will create a copy of the schema locally.

4. $node_{dest}$ registers each necessary transfer locally and starts serving reads and writes on $P_1, ..., P_v$. For each request, $node_{dest}$ reads the accompanying snapshot information and compares the lowest active version to x. Once it's greater than x, the transfers are initiated. Incoming tuples already available locally with a higher version will simply be discarded by $node_{dest}$.

5. Once all tuples have been inserted locally at $node_{dest}$, it will notify the commit-manager.

6. Upon receiving the acknowledgement, the commit-manager removes the *bootstrapping* tag from $P_1, ..., P_v$.

The first step of this protocol is crucial. In it, we determine the transaction version x, after which we assume all processing nodes have heard of the new node (we will of course have to ensure this later in section 5.4). In other words, all writes in transactions after x will go to the new node, because all processing nodes will be aware of it and will be routing requests accordingly. This is enough to satisfy theorem 5.2. In fact, this guarantee is so crucial, that it warrants highlighting:

Lemma 5.7. *All writes in transactions with transaction id greater than x will go to $node_{dest}$.*

By definition, once the lowest active version is greater than x, no transaction with version smaller than x is still running. In step 4 of the protocol, we therefore use this as the condition for starting the key transfer. This ensures that the key transfer will only start once the last transaction with version smaller than x has completed. Since all writes in transactions after x will go to the new owner (by lemma 5.7), no more writes will be active or sent to $node_{src}$. This is equivalent to the requirement established in theorem 5.1 and therefore satisfies it. As locally available tuples are preferred over transferred ones, 5.5 is satisfied as well.

Marking the partitions as *bootstrapping* in step 2 is important information for the processing nodes, as they will treat such partitions differently. This will become apparent in section 5.4.

5.4 An Extended Protocol For Performing Transactions

As we've seen in the previous section, some cooperation from the processing layer is necessary to obtain up-to-date routing information and to uphold the guarantees provided by theorems 5.1, 5.2, 5.3 and 5.4. Achieving this will be the goal of the following protocol. We consider a processing node *proc* that has to be initialized with the address of the commit-manager, no other information apart from that is required for the protocol to work.

1. *proc* contacts the commit-manager to start a new transaction at version number t.

2. The commit-manager responds with snapshot information and the current version of its hash ring.

3. *proc* stores a copy of the commit-manager's hash ring and proceeds with executing reads and writes inside the transaction. For each request on a key k, *proc* will run the consistent hashing algorithm on its local version of the hash ring, in order to determine the storage node that owns k and can therefore serve the request.

4. For reads on keys inside a partition that has been marked as *bootstrapping*, i.e. a partition that is currently being transferred from $node_{src}$ to $node_{dest}$, *proc* will attempt the read on both storage nodes. If the read succeeds at $node_{dest}$, *proc* will mark the keys as *transferred*.

5. Writes on keys inside a partition that has been marked as *bootstrapping* that have not been read successfully from $node_{dest}$ yet (i.e. that are not marked as *safe*), will be replicated at $node_{src}$. The transaction will abort, if either of the writes fails.

By updating each processing nodes routing information on the start of each new transaction, we can guarantee that a processing node executing transaction t has heard of all changes to the commit-manager's hash ring, up to that point in time. Therefore if $t > x$, *proc* must have heard of the new node $node_{dest}$ and will route all writes to it. This validates the assumption made in section 5.3.

Since reads on bootstrapping partitions are retried at the previous owner, theorem 5.3 is now also satisfied. The same holds for 5.4, as writes are replicated as well. By only replicating writes on *unsafe* keys (keys that have not been read successfully from $node_{dest}$ yet) we achieve a very fine-grained solution, that avoids unnecessary replication as much as possible and has close to no overhead on other writes. Additionally, the implementation of this simple read and write replication can be extended in the future, to handle replication to replica nodes as well.

In practice we will have to differentiate between updates and inserts for the replication, because updates on keys that are not yet available at $node_{dest}$ should fail, even though the keys simply haven't been transferred yet. Inserts don't have this problem, as inserts on already existing keys will fail at $node_{src}$. We will therefore first try every update as an insert at $node_{dest}$. If that fails, the key has arrived at $node_{dest}$ in the meantime, so we can retry the original update. This solution is sub-optimal and a cleaner solution is outside the scope of this thesis, but will be discussed briefly in chapter 8.1.

It is always safe to treat a partition as bootstrapping when the transfer has already finished, as sending redundant reads or writes to the old owner will not harm consistency. A slight problem exists when $node_{src}$ is shut down after the transfer, as a failed, replicated write will cause an abort. For now, this will

problem will have to be solved by operator diligence. Treating a partition as non-bootstrapping when a transfer is still in progress leads to serious consistency problems, as the safety mechanisms provided by read and write replication will not be performed. Therefore we have to be careful to only remove the bootstrapping tag once the transfer has finished *and* all transferred tuples have been inserted at $node_{dest}$. Only once this is the case will $node_{dest}$ acknowledge the transfer with the commit-manager. From this point on, *read before transfer* can never occur, so the commit-manager may safely remove the bootstrapping tag. With the next transaction, *proc* will learn that the bootstrapping tag has been removed and will stop sending redundant read retries.

5.5 A Protocol For Removing a Node

From a conceptual point of view, removing a node is very similar to adding one. Again, keys have to be transferred, but this time from the leaving node to one of the remaining ones. Once the transfer is acknowledged, the leaving node can be safely removed, while all data remains available at the new node. No change in the actual transfer is required, therefore all safety considerations will still hold. To further underline the similarities between the two protocols, lets refer to the leaving node as $node_{src}$.

1. Upon deciding to leave, $node_{src}$ unregisters with the commit-manager inside of a transaction. This establishes the version x, from which point on the new partitioning information will have been seen by all processing nodes.

2. The commit-manager updates its hash ring and uses it to determine the nodes that will take over responsibility for $node_{src}$'s partitions $P_1, ..., P_v$. $P_1, ..., P_v$ are marked as *bootstrapping*. We will call the new owners *destinations*.

3. $node_{src}$ contacts all destinations and orders them to request a key transfer at version x. Each destination node registers the respective transfer locally and proceeds as in step 4 of the node addition protocol, using x to determine when to start the actual transfer.

Since the actual transfer proceeds as in protocol 5.3, theorems 5.1 to 5.5 still hold.

5.6 Observations

Taken together, protocols 5.3 to 5.5 satisfy the safety and correctness guarantees we identified in section 5.

The check in step 4. of the addition protocol in section 5.3 might just as well be done by the $node_{src}$ node, as the commit-manager ensures that the lowest active version is consistent throughout the cluster. But doing it in the $node_{dest}$ node has multiple, non safety-related benefits: It is entirely possible, that no

request will ever reach a leaving $node_{src}$, once it has unregistered. This happens when no transactions are still in-flight at that point in time. In that case, $node_{src}$ would never perform the lowest active version check and the transfer would never be started. Our implementation would require an additional mechanism for $node_{src}$ to regularly check-in with the commit-manager and kick-off any queued transfers. By doing the check in $node_{dest}$ instead, our protocol has no need for such a mechanism.

Additionally, all state relevant to the transfer is kept only on the joining / leaving node respectively and on the commit-manager. This alleviates the need for special cleanup requests to remove queued transfers. Should anything go wrong during a node addition / removal, the protocol can simply be canceled and restarted. In fact, a node giving up a partition is completely unaware of the new node, apart from regular request-response interactions.

6 Implementation

6.1 Choosing a Hash Function

Choosing a good hash function for our purposes could have warranted a thesis in and on itself. Important properties of hash functions are the time it takes to compute them, how well they distribute their input data and how frequently they produce collisions (two distinct inputs mapping to the same hash value).

Typical contenders in distributed databases are MD5 and the Murmur family of hash functions, of which Murmur3 is the most recent one. In contrast to MD5, Murmur3 is non-cryptographic, around four times faster and maintains great collision and avalanche behavior.

We settled on Murmur3 for its speed and popularity in real-world systems [12]. Murmur3 produces a 128-bit hash value. We use the reference implementation by its inventor Austin Appleby, available in the public domain at [6].

6.2 Choosing a Key

Once the hash function has been decided on, we need to chose what part of a tuple will be hashed to determine its partition. This is whats known as the *partition key*. As many comparable systems provide a pure key-value interface, the partition key is usually just the tuple's key. Since Tell provides a Table abstraction, tuple keys are only guaranteed to be unique inside a single table. Our partition key therefore had to be composite, made up of the tuple key and the id of the table it is contained in.

6.3 Hash Ring Operations

The hash ring is the data-structure used to implement consistent hashing. It models a wrapping range of 128-bit tokens. As we recall from our discussion of consistent hashing, nodes and partition keys are mapped onto this range, by

use of a hash function (Murmur3 in our case). This allows us to order nodes and partition keys by their hashes and to establish a definition of ownership:

Definition 6.1. Let $token_1, token_2, ..., token_n$ be the ordered sequence of node hashes. Then node i owns all keys in the range $(token_{i-1}, token_i]$. In addition, the lowest node ($token_1$) owns the range $(token_n, max]$. Where max is the largest possible token.

Note that we will actually be assigning tokens to each *virtual* node. The hash ring has to support operations for inserting and removing new nodes, retrieving the owner of a given key and computing all ranges owned by a given node.

We based the implementation on the std::map implementation of an ordered hash map. Besides operations to access the first and last elements in the map, **First** and **Last** respectively, the sorted map also provides us with an efficient **LowerBound** operation, that can retrieve the first token in the map, that is greater or equal to a given one in logarithmic time. If no such node exists, **LowerBound** will return a special pointer indicating the end of the hash ring (**End**). Similarly, given a node in the map, we can efficiently access it's predecessor using the **Predecessor** operation. Using this, we can give rather concise, pseudo-code descriptions of the core operations:

6.3.1 Adding a Node

Generally, the new node will be inserted between two others in the ring. This case is depicted in figure 5. There the new node N is inserted between nodes A and B. Following the consistent hashing algorithm, N will be responsible for the range $[A + 1, N]$. The two special cases, inserting a node with the highest token and inserting a node with the lowest token in the ring require no special treatment during the insert operation, but have to be considered in section 6.3.4.

Algorithm 1 insertNode :: HashRing → Node → HashRing

```
function InsertNode(ring, nodeId)
    for all vnodeId ∈ [0, numVirtualNodes] do
        partitionToken ← Murmur3(nodeId, vnodeId)
        ring[partitionToken] ← nodeId
    end for
end function
```

6.3.2 Removing a Node

Again, removing a node will generally occur in a setting as depicted in figure 6. Node B leaves the cluster, offloading it's partition onto node C. This also illustrates the hot-spots that can occur when removing nodes without the use of virtual nodes: C is now responsible for the majority of the key-space. As in the insertion operation, we don't have to treat the highest and lowest nodes specially during removal.

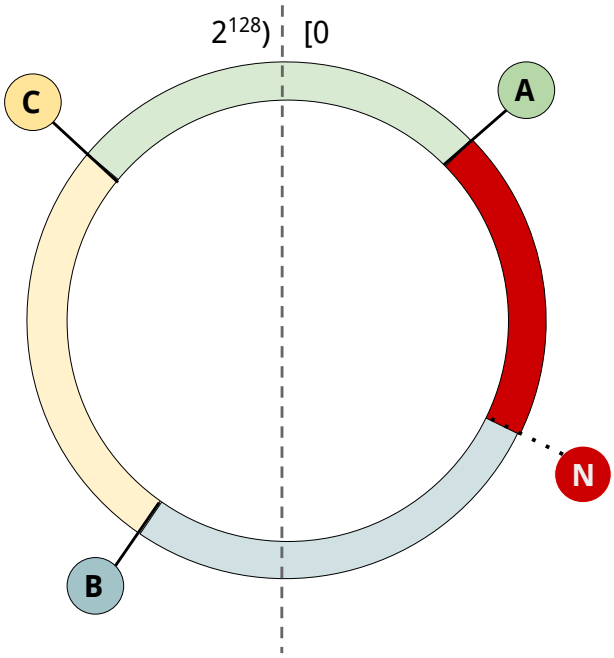


Figure 5: Inserting a new node into the ring

Algorithm 2 removeNode :: HashRing → Node → HashRing

function REMOVENODE($ring, nodeId$)
  for all $vnodeId \in [0, numVirtualNodes]$ **do**
    $partitionToken \leftarrow$ MURMUR3($nodeId, vnodeId$)
    delete $ring[partitionToken]$
  end for
end function

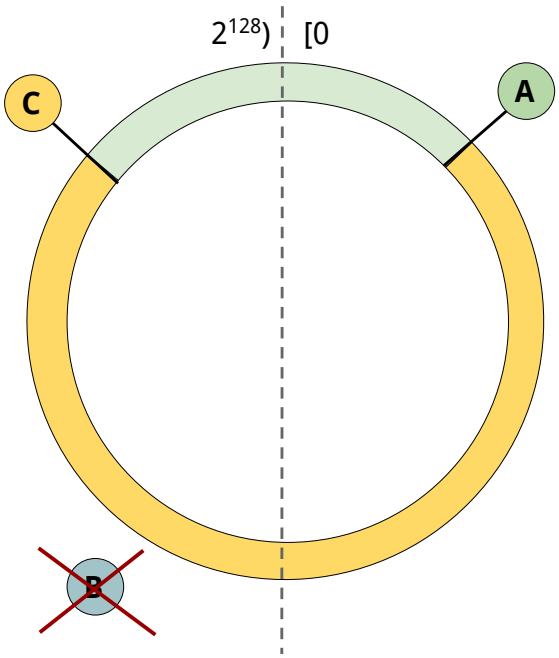


Figure 6: Removing a new node from the ring

6.3.3 Finding The Owner Of A Given Key

This operation will be used much more frequently than the others. It is the basic computation that makes consistent hashing work. Processing nodes will use this operation to route requests. To find the owner for a given key, we compute its partition token and walk the ring in clockwise direction. The first node encountered will be the owner for this key. In figure 1 for example, node A would be responsible for key 10. If no node is encountered, **LowerBound** will return **End**, indicating that no node was found. In that case we have to wrap-around and return the first node in the ring.

Algorithm 3 getOwner :: HashRing → TableId → Key → Node

```
function GetOwner(ring, tableId, key)
    token ← Murmur3(tableId, key)
    owner ← LowerBound(ring, token)
    if owner = End(ring) then
        owner ← First(ring)
    end if
    return owner
end function
```

6.3.4 Computing Partitions For a Node

The last important operation, finding all partitions a given node is responsible for, is more complex, because we will have to consider a number of special cases. Many publicly available descriptions of the consistent hashing scheme also gloss over this important part of the algorithm.

Remark. *It is helpful to observe, that*

$$LowerBound(ring, token_i) = \begin{cases} node_i & \text{if } node_i \in ring \\ node_{i+1} & \text{if } node_i \notin ring \end{cases}$$

meaning that we can use the same operation to determine ranges a node will be responsible for, before and after it has been inserted into the ring.

There are two special cases. One is the node with the lowest token, which must be responsible for the very last partition in the key space as well (as we are dealing with a wrapping range). Recall figure 1, there node A is the node with the lowest token and therefore responsible for the ranges $[0, A]$ and $[C+1, 2^{128})$. The other special case follows from the first one. When a new node with a token higher than that of any other node in the ring is inserted, the currently highest token will mark the beginning of this node's partition. It has to contact the first node in the ring, as that is the node currently owning the last partition in the ring.

Algorithm 4 getPartitions :: HashRing → Node → [Partition]

```
function GetPartitions(ring, nodeId)
    partitions ← ∅
    for all vnodeId ∈ [0, numVirtualNodes] do
        token ← Murmur3(nodeId, vnodeId)
        owner ← LowerBound(ring, token)
        if owner = First(ring) then
            (lastNode, ·, lastToken) ← Last(ring)
            partitions ← partitions ∪ (owner, lastToken + 1, maxToken − 1)
            partitions ← partitions ∪ (owner, minToken, token)
        else if owner = End(ring) then
            (lastNode, ·, lastToken) ← Last(ring)
            currentOwner ← First(ring)
            partitions ← partitions ∪ (currentOwner, lastToken + 1, token)
        else
            (neighbour, ·, neighbourToken) ← Predecessor(ring, owner)
            partitions ← partitions ∪ (owner, neighbourToken + 1, token)
        end if
    end for
    return partitions
end function
```

Processing nodes will also need to determine the previous owner of a partition. This could be inferred from the information contained in the hash ring, simply by ignoring the current owner and continuing to walk the ring until the next higher node is encountered. Unfortunately this approach can break when more than one node join concurrently, as the next higher node in the ring might have just joined as well. Information on which partitions are bootstrapping is contained in the ring, so one could probably modify this algorithm to continue walking the ring, until a non-bootstrapping node is found.

Maintaining the previous owner explicitly is cheap enough and not hard to do, so we went with that option, instead of proving the above algorithm correct. This could be an area of future work.

6.4 Caching Partitioning Information

Lots of transactions in combination with the very low frequency of storage node additions or removals means that a lot of time and bandwidth is wasted, re-transmitting the same hash ring information over and over again. It is therefore a natural target for some kind of caching scheme. We re-utilize the transaction versions for this. Whenever the commit-manager updates its hash ring (this must always happen inside of a transaction), it will note the transaction's version number t. Each processing node will also keep track of the last transaction id t_{cached}, at which it received new partitioning information. t_{cached} will be sent whenever the processing node is starting a new transaction. From it, the

commit-manager can easily decide on whether it needs to include the current version of the hash ring with the response.

Keeping a cached copy of the hash ring per thread instead of per-node is relatively cheap and avoids contention, which is important to maintain the lock-free operation of Tell. Transactions on a single processor will of course still be concurrent, so the hash ring might change during execution of a transaction. This is still safe, as partitioning information can only get newer, so no bootstrapping partitions will suddenly lose their tag. With multiple processors it becomes important to pin each transaction to the processor it was started on, as it might otherwise receive outdated partitioning information.

6.5 Key Transfer

As for the key transfer itself, the natural choice was to utilize Tell's fast scan capabilities, as the two operations are quite similar: In principle a key transfer of a partition $P = [k_{lower}, k_{upper}]$ is equivalent to a scan with two predicates $P_1(key) = key >= k_{lower}$ and $P_2(key) = key <= k_{upper}$. We had to extend Tell's scan operation to include the MVCC timestamps for each tuple. This can still lead to unnecessary aborts:

Consider a situation in which transaction t_1 has already completed. Transaction t_5 is running. Between the completion of t_1 and the start of t_5, a record r, accessed by both, is being transferred to a new node. Now t_5 could write r at its new owner before the transfer has completed. This write will therefore overwrite the value of r written by t_1. Another transaction t_4, started before t_5 now tries to read r, but fails, because r is not in its read set. This is because t_4 is unaware of t_5 and the original value of r (written by t_1) is not available any more. In this case, t_4 would have to abort. To avoid this problem, the key transfer must include all active versions of each tuple. We did not yet implement this.

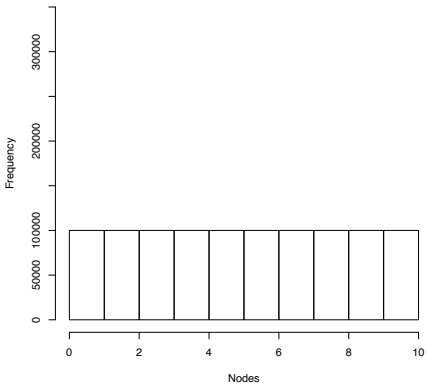
To achieve the best possible performance on all platforms, Tell uses LLVM to compile each scan request into highly optimized machine code. While this has many benefits, it of course requires us to support each type of field and each predicate in the LLVM code generator. We are also restricted to relatively simple computations, as anything more complex would have to be supported in the code generator as well. We remind ourselves that a tuple's partition key is determined by the Murmur3 hash of it's internal, 64-bit Tell key and the 64-bit id of the table it is contained in. A scan operation on partition keys would therefore have to compute these hashes on the fly and entirely in LLVM. Storing an additional 128 bits of partitioning key per tuple was an acceptable trade-off in favor of execution speed and code generator simplicity. Nevertheless, implementing Murmur3 in LLVM to remove the need for explicitly stored partition tokens could be an interesting area for future work.

Before this work, Tell supported fixed size fields of up to 64 bit and variable sized string and binary blob fields. But our partition keys are 128 bit in size. We therefore decided to extend the LLVM code generator with a new field type for 128-bit values and the accompanying predicates.

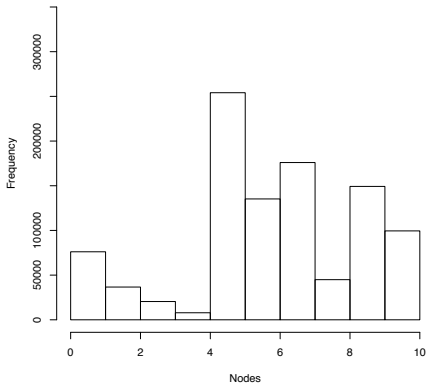
7 Experiments

7.1 Key Distribution

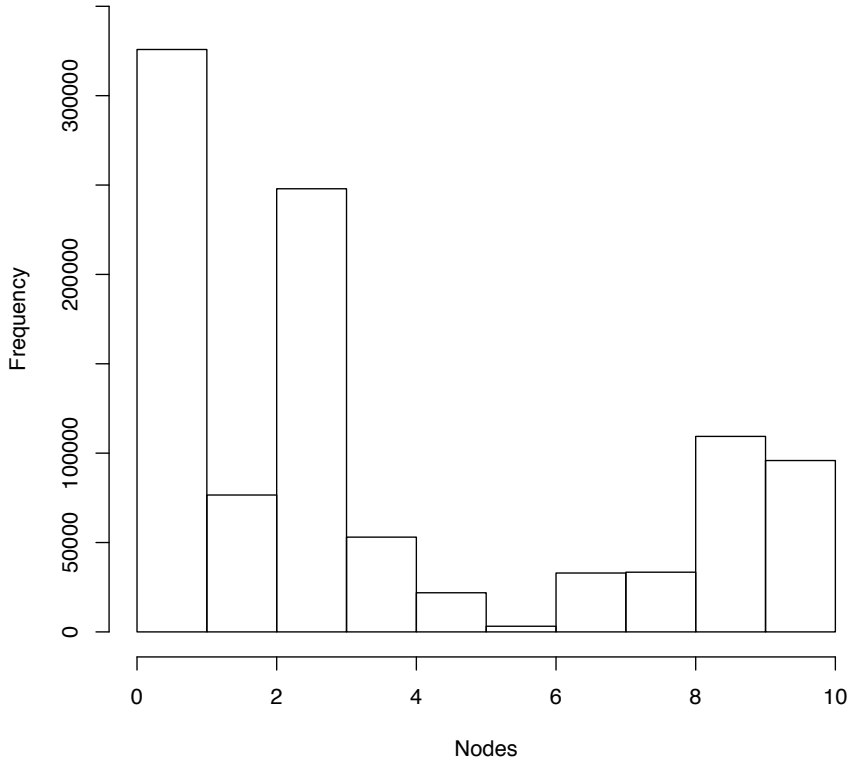
We distributed a continuous range of 1 million integer keys onto ten physical nodes. Figure 7 shows the resulting distribution using a simple modulo operation as a baseline, the std::hash default hash function and finally the Murmur3 implementation now used in Tell. Each physical node is only assigned a single token (one virtual node per physical node). In line with our discussions in section 4.2.3, a single token per node leads to a very uneven distribution of keys throughout the cluster. As expected, modulo-based partitioning achieves a perfectly uniform distribution. Figure 8 shows the resulting distribution when re-running the experiment with 50 tokens per node. Already, uniformity of the distribution has increased significantly. The even higher number of 200 tokens per node is still necessary, to maintain the distribution after node removals.



(a) Modulo baseline

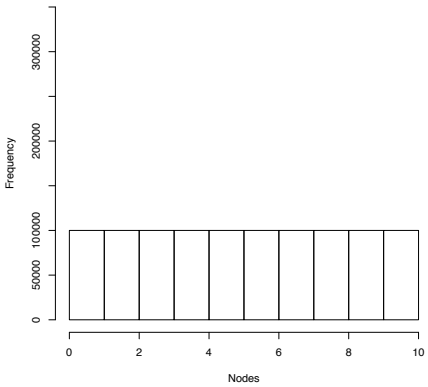


(b) std::hash

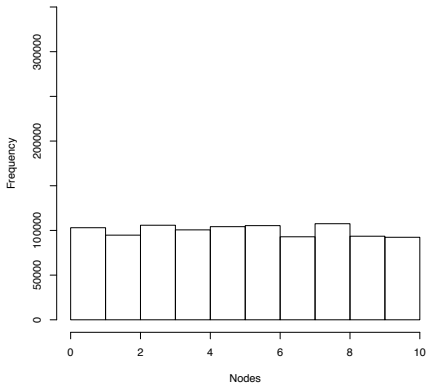


(c) Murmur3

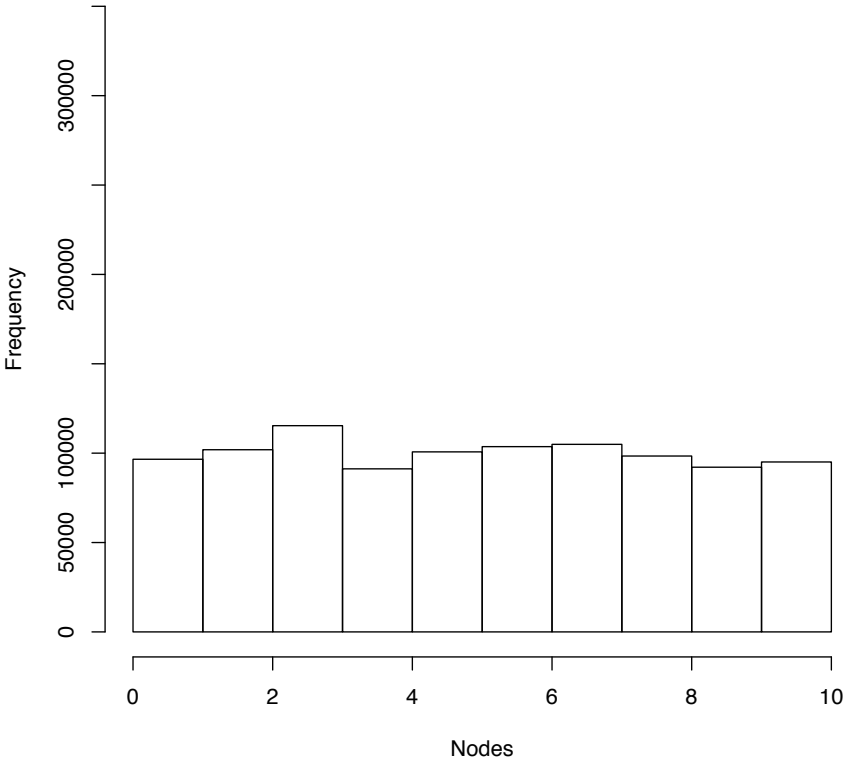
Figure 7: Key distribution with a single token per node



(a) Modulo baseline



(b) std::hash



(c) Murmur3

Figure 8: Key distribution with 50 virtual nodes per node

7.2 TPC-C

In order to measure the overheads introduced by our design, we compared a TPC-C run of the old, non-elastic Tell to our implementation. The experiment was performed on a single physical server running a commit-manager instance, two Tell storage nodes, the TPC-C server application and a single TPC-C client. The resulting transaction latencies are shown in figure 9. We calculated latency quantiles in milliseconds, these are shown in table 1.

Quantile	99%	95%	75%	5%
Ms	4	2	1	0

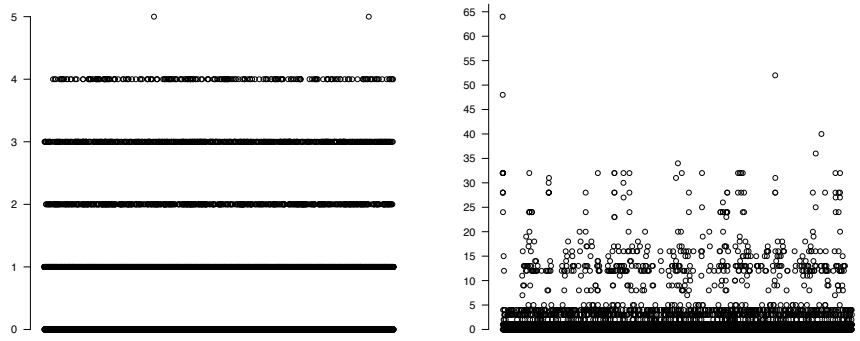
(a) Non-Elastic Tell

Quantile	99%	95%	75%	5%
Ms	24	8	1	0

(b) Elastic Tell

Table 1: Latency Quantiles in Milliseconds

Transaction latency got less predictable overall, and about 25% of transactions completed significantly slower. A very small number of transactions completed about an order of magnitude slower than the norm, some of these, early on during the benchmark execution, can be traced back to step 4 in protocol 5.3, as they cause the initiation of an initial key transfer between the two cold storage nodes. Further analysis is necessary to explain spikes later on in the benchmark.



(a) Non-Elastic Tell

(b) Elastic Tell

Figure 9: TPC-C latencies in milliseconds

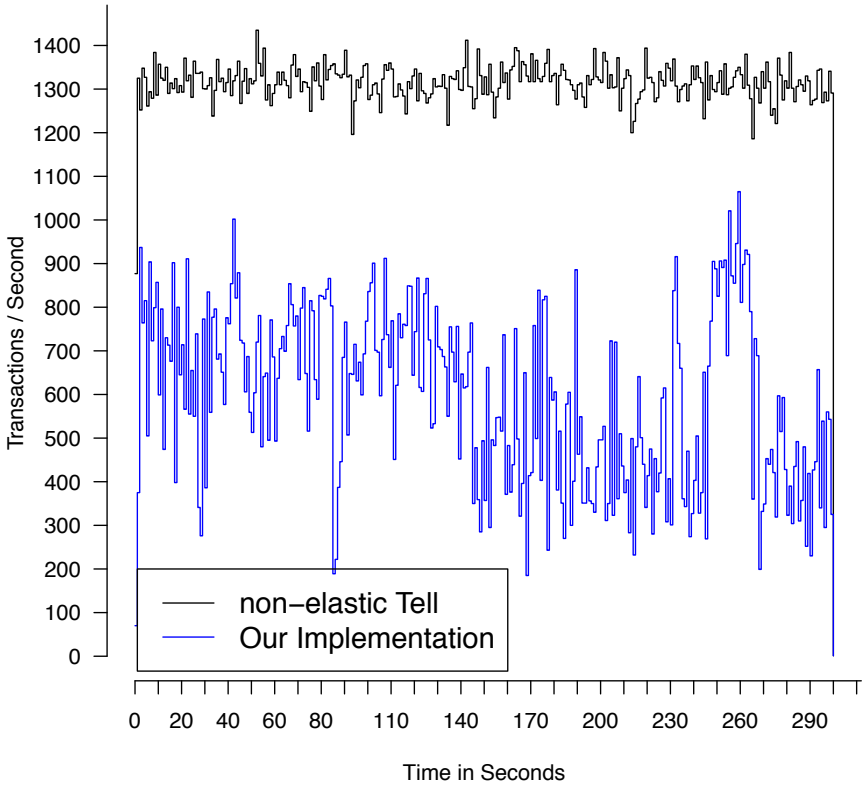
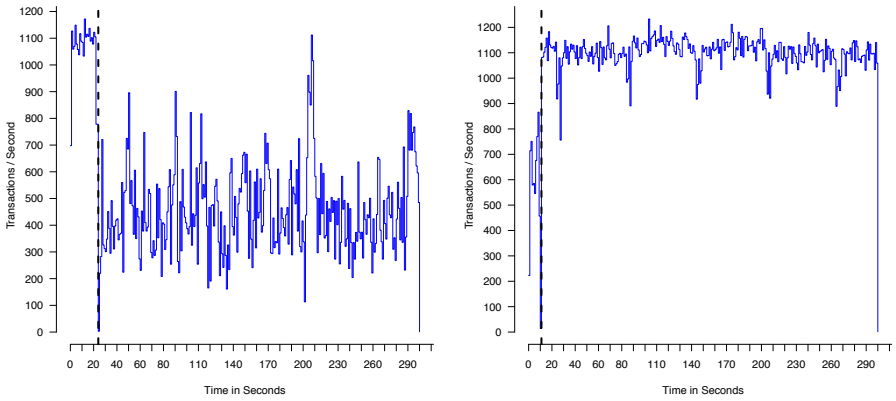


Figure 10: Throughput during regular operation

The same setup was used to measure throughput during regular TPC-C operation (no storage additions or removals). We plotted the number of completed transactions per seconds in figure 10. Overall, our implementation performs at only 50-60% of non-elastic Tell and throughput varies greatly, making predictable operation very hard. Still we expect to eventually match non-elastic Tell's performance, as the current results can be attributed to a number of factors: Processing nodes have to compute a Murmur3 hash for every request, and look up its owner in their local hash ring. This takes an essentially fixed number ($O(\log n)$) of comparisons but is still significantly more expensive. Storage nodes have to compare a new snapshot against all their registered transfers on every incoming request (step 4 of protocol 5.3). Processing nodes will also replicate reads and writes on bootstrapping partitions, causing additional overhead. Our current implementation does not yet mark individual keys as *safe* (see protocol 5.4), meaning that many writes might be unnecessarily replicated



(a) Recovery after addition

(b) Recovery after removal

Figure 11: Recovery times for non-elastic Tell

(as keys are usually read before written). Non-elastic Tell was compiled with advanced compiler flags like link-time optimization enabled. At the time of this writing, elastic Tell was not yet compatible with these, and was thus compiled without any advanced optimizations enabled. Additionally, our current implementation contains a concurrency-related bug, forcing us to run the TPC-C server in single-threaded mode, whereas for non-elastic Tell, the benchmark was run on two threads. Another unexplored area is the effect of distributing Tell's internal index tables across multiple nodes. Many smaller performance optimizations were also left out, as we focused on safety and availability first. Lastly, as will be discussed in 8.1, our current implementation may unjustifiably abort transactions in certain situations.

Finally, we ran TPC-C again, adding and removing a storage node respectively, to measure the impact of these operations and the time it takes the cluster to recover normal operations (under the limitations discussed above) afterwards. The resulting graphs are presented in figure 11. The dashed lines indicate the time at which the re-sizing operation was performed. Immediately after the operation throughput levels drop dramatically, but normal operation is restored after just a few seconds.

8 Future Work

We identified a couple of targets for future efforts, building atop the results presented in this work. As mentioned in section 6.5, an implementation of Murmur3 in LLVM would allow us to compute partition tokens on the fly. If this proves efficient enough, explicit storage of partition tokens would not be necessary anymore.

Storage nodes leaving the cluster should keep track of their outgoing transfers and print a log message, once it is absolutely safe to take them offline. Currently an operator might shut down the node while a transfer is still in progress. Similarly, Tell's garbage collector should be extended to efficiently remove partitions that are no longer owned by a node.

Additional measures have to be taken in order to prevent a potential race condition when adding a new storage node, while a new schema is still being deployed across the cluster. In such cases, the new node might end up with only a partial view of the schema.

The performance of our consistent hashing implementation might be improved by pre-computing and caching each node's tokens.

8.1 Storage Extensions

Recall chapter 5.4. There we hinted at a problem with organizing writes at the new owner $node_{dest}$, while a key transfer is in progress, as updates are handled differently from inserts. Specifically, inserts of keys that already exist and updates on keys that have not been inserted yet are prohibited. This becomes problematic, when part of a key's history has not yet been transferred to its new owner. By replicating writes and turning updates into inserts, we found a solution that maintains consistency but involved unnecessary retries and a lot of cases to handle in the implementation.

A cleaner solution would be to properly merge tuples from the key transfer with new writes at $node_{dest}$. Such a solution would have to be implemented for every storage layout provided by Tell. We will only consider the *delta-main* storage layout now. In this layout, new writes are appended to a buffer D (called the *delta*) that is periodically merged into structured storage (called *main*). $node_{dest}$ could block the merge while a key transfer is still in progress and store all transferred data in a separate delta $D_{transfer}$. Regular writes at $node_{dest}$ would go into D as per usual, but without checking the existence of the key for updates. Once the transfer completes, $node_{dest}$ would first merge $D_{transfer}$ into the (still empty) *main* before unblocking the regular merge process.

The replication employed in protocol 5.4 guarantees that all updates in D would still be valid, even though key existence is not enforced, so no transactions could have erroneously committed. With this setup, updates don't have to be first tried as inserts by transactions anymore.

9 Conclusion

We revisit the goals of this thesis, established in chapter 3:

1. It is now possible to add a storage node during normal cluster operation. Consistent hashing guarantees us, that, on average, only $1/n$th of all keys have to be rebalanced. By assigning 200 tokens to each physical node, we achieve an almost uniform distribution (compare figure 8). Due to

the performance problems discussed in section 7.2, adding storage nodes currently only increases the clusters storage capacity, not it's throughput. Still, adding storage nodes does relieve existing nodes under heavy load.

2. Planned removal of nodes is now possible without data loss. Removing an existing storage node from the cluster moves all its data onto remaining peers. Because of the virtual nodes, an almost uniform distribution is still maintained.

All protocols involved in our design operate entirely lock-free and the cluster stays available, as we saw in section 7.2. As expected, some performance overhead was introduced. The current implementation, as presented in this thesis, provides safety and availability, but further work and optimizations are necessary to achieve performance levels in the same range as non-elastic Tell.

The implementation is available at *github.com/comnik/tell*, all experimental data and the scripts used to produce the presented figures and graphs are available at *github.com/comnik/consistent-hashing-results*.

References

[1] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquin, D. Kossmann, *Fast Scans on Key-Value Stores* (under submission)

[2] D. Karger et al., *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654-663.

[3] D. Thaler, C. Ravishankar, *A Name-Based Mapping Scheme for Rendezvous*.

[4] Y. Zizhen, C. Ravishankar, et al., *Hash-Based Virtual Hierarchies for Caching in Hybrid Content-Delivery Networks*. static.cs.ucr.edu/store/techreports/UCR-CS-2001-05062.pdf, as accessed on 2016-07-28.

[5] A. Lakshman, P. Malik, *Cassandra: a decentralized structured storage system*. In *ACM SIGOPS Operating Systems Review archive*, Volume 44 Issue 2, April 2010, pages 35-40.

[6] A. Appleby, github.com/aappleby/smhasher. As accessed on 2016-07-28.

[7] T. Lipcon et al., *Kudu: Storage for Fast Analytics on Fast Data* kudu.apache.org/kudu.pdf

[8] F. Chang et al., *Bigtable: A Distributed Storage System for Structured Data*. In *ACM Transactions on Computer Systems (TOCS)*, Volume 26 Issue 2, June 2008, Article No. 4

[9] G. DeCanadia et al., *Dynamo: amazon's highly available key-value store*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205-220.

[10] A. Sicular, *Why Riak Just Works*, basho.com/posts/technical/why-riak-just-works/, as accessed on 2016-07-25.

[11] I. Stoica, R. Morris, D. Karger, et al., *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149-160.

[12] *Choosing a Good Hash Function* research.neustar.biz/2012/02/02/choosing-a-good-hash-function-part-3/, as accessed on 2016-07-28.

[13] M. Meyer, *The Simple Magic of Consistent Hashing* paperplanes.de/2011/12/9/the-magic-of-consistent-hashing.html, as accessed on 2016-08-06.

[14] M. Perham, *Consistent Hashing in memcache-client* mikeperham.com/2009/01/14/consistent-hashing-in-memcache-client/, as accessed on 2016-08-06.

[15] T. White, *Consistent Hashing* tom-e-white.com/2007/11/consistent-hashing.html, as accessed on 2016-08-06.

List of Figures

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

ELASTICITY IN TELL

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Göbel	Nikolas

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Zürich, 12. August 2016	NGöbel

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.