DISS. ETH NO. 24147

# Tell: An Elastic Database System for Mixed Workloads

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

MARKUS PILMAN

Master of Science ETH in Computer Science, ETH Zurich

born on 18. September 1983

citizen of Amriswil, Thurgau

accepted on the recommendation of

Prof. Dr. Donald Kossmann (ETH Zürich), examiner
Dr. Philip A. Bernstein (Microsoft Research), co-examiner
Prof. Dr. Peter Boncz (Vrije Universiteit Amsterdam), co-examiner
Prof. Dr. Timothy Roscoe (ETH Zürich), co-examiner

2017

# Abstract

It is an exciting time to do database research. Two movements dominated the field for the last few years: Big Data and NoSQL. Both movements arose out of necessity, as cloud computing imposes new requirements on database systems.

Cloud computing makes scalability and elasticity more important than ever. A user does not want to pay for computing and storage resources she does not use, but she expects to be able to get these resources as soon as they are needed. Traditional database management systems, however, are not able to meet these requirements.

Early NoSQL systems provided elasticity and scalability by massively simplifying the provided consistency guarantees and the underlying data model. Most notably key value stores can scale to thousands of machines and allow resizing their cluster at runtime. However, their simplicity is also their greatest weakness: The lack of transactions makes it difficult to reason about concurrency, and the simple data model makes them difficult to use. Key value stores push most of their complexity into the application. As a result, more recent solutions try not only to add transactions, but they also implement complex operations in a layer above the underlying NoSQL storage. This layering is often referred to as *SQL over NoSQL*.

Big Data, on the other hand, is about the analytical processing of massive amounts of data in the cloud. The Hadoop ecosystem and, more recently, Spark are the most prominent systems that play in this field. These systems allow for massive parallelization of complex analytical queries and are elastic and scalable. They achieve this by implementing a shared data architecture which decouples computing resources from storage resources.

However, these Big Data platforms still have a problem: bringing the data from the online NoSQL (or SQL) database into Hadoop is a complex issue. Traditionally, this is solved like traditional data warehousing which is a heavy weight solution.

A system like Spark also can not simply use a key value store for its underlying storage, because current key value stores perform poorly when they have to deliver high volumes of data.

This thesis introduces Tell, a distributed shared-data database management system that fills the gap between NoSQL and Big Data. Tell implements the SQL over NoSQL design principle: it performs transaction processing on top of a high-performance key-value store. At the same time, its key value store is heavily optimized for scan queries, allowing data processing engines to fetch their data directly from the online database.

# Zusammenfassung

Big Data und NoSQL waren und sind Bewegungen die die Datenbankwelt in den letzten Jahren stark geprägt haben. Beide Bewegungen entstanden aus der Notwendigkeit neuen Anforderungen, die vor allem Cloud Computing an moderne Datenbanksysteme stellt, zu genügen.

Die wichtigste Eigenschaft die ein System innerhalb der Cloud haben muss ist Elastizität. Ein Kunde will nicht für Ressourcen bezahlen die er nicht nutzt. Der Anbieter muss jedoch in der Lage sein, zusätzliche Ressourcen zur Verfügung zu stellen sobald sich die Anforderungen der Kunden ändern.

Elastizität und Skalierbarkeit wurden von NoSQL Systemen zu beginn erreicht indem die Konsistenzgarantien und das unterstützte Datenmodel stark vereinfacht wurden. Dank diesen Vereinfachungen können, zum Beispiel, Key-Value Stores auf tausende von Maschinen skalieren und zur Laufzeit Machinen aus dem Cluster entfernen oder zum Cluster hinzufügen.

Leider ist jedoch diese Vereinfachung sowohl eine Stärke als auch eine Schwäche: Die fehlenden Transaktionen und das stark vereinfachte Datenmodel erschweren die Benutzung dieser Systeme, da die Komplexität von der Datenbank in die Applikation verlagert wird. Aus diesen Gründen versuchen neuere Systeme Transaktionen und komplexe Operationen eine Schicht oberhalb eines NoSQL storages zu implementieren. Dieser Ansatz wird häufig als *SQL over NoSQL* bezeichnet.

Big Data ist ein anderer Trend der letzten Jahre. Wir produzieren immer mehr Daten und wollen mit analytischen Abfragen Informationen aus diesen Daten gewinnen. Hadoop, und seit ein paar wenigen Jahren Spark, sind die bekanntesten Produkte die aus diesem Trend hervor gingen. Diese Systeme erlauben mit massiver Parallelisierung komplexe Anfragen über grosse Datenmengen zu beantworten. Beide Systeme skalieren und sind elastisch. Dies wird erreicht indem die Recheneinheiten vom Storage getrennt werden.

Leider haben diese Systeme einen grossen Nachteil: die Daten die verarbeitet werden sollen befinden sich meistens in einem Datenbanksystem, entweder einer NoSQL oder einer traditionellen SQL Datenbank. Die Daten müssen dann kopiert werden damit sie von Spark oder Hadoop gelesen werden können. Leider können Spark und Hadoop die Daten nicht direkt aus dem Online-System lesen, da Key-Value Stores nicht optimiert sind um grosse Datenmengen zu lesen.

Diese Dissertation stellt Tell vor, eine verteilte shared-data Datenbanklösung die die Kluft zwischen NoSQL und Big Data füllt. Tell arbeitet nach dem SQL over NoSQL Prinzip: es implementiert Transaktionen eine Schicht oberhalb eines schnellen Key-Value Stores. Der Key-Value Store ist zusätzlich in der Lage grosse Datenmengen direkt an analytische Systeme zu liefern. Dies erlaubt sowohl Echtzeit-Transaktionsverarbeitung und analytische Anfragen auf den selben Daten.

# Acknowledgments

> Dicebat Bernardus Carnotensis nos esse quasi nanos gigantum umeris insidentes, ut possimus plura eis et remotiora videre, non utique proprii visus acumine, aut eminentia corporis, sed quia in altum subvehimur et extollimur magnitudine gigantea
>
> (John of Salisbury: Metalogicon 3,4,46-50)

Working on this thesis was a tremendous learning experience and this work would not have been possible without those who worked in the field before me and with me. I want to express my deepest gratitude towards Donald, who advised me not only during my doctorate but virtually throughout all my years at ETH. Donald granted me a lot of freedom, but he was always there when he was needed.

I also want to thank Mothy for his support. Discussions with him were always insightful. Mothy's knowledge within and outside of his field is impressive. I also want to thank him for his life-advice to my wife and me.

Throughout my doctoral studies, I had the chance to meet and work together with brilliant people. I want to thank all the people in the Systems Group for interesting discussions and new insights. While working on a particular problem, it was always helpful to have people who presented me the problem from a different angle. At this point, a special thank goes to my office mates, Simon, Lucas, and Jonas. Sharing the workplace with them was always pleasant.

During my studies, I did an internship at Microsoft Research under the supervision of Phil Bernstein. I want to thank Phil who was an awesome adviser. Even though I sometimes felt dumb while talking to him, he was always patient and never condescending. During my short time at Microsoft, I also worked a lot with Sudipto Das, from whom I also learned a lot and who was a very pleasant person to work with. Microsoft Research is filled with brilliant people, and I am glad I had the chance to talk to many of them.

Tell was quite a big project and I would not have been able to do it alone. A first thanks goes to Simon Loesing, who worked closely together with me on Tell for nearly three years. I had the rare luck to supervise only excellent master students during my doctorate. Thomas Etter and Kevin Bocksrocker did an enormous amount of the implementation work - during their master thesis and while working for the Systems Group as Software Engineers. Thanks goes to Daniel Bucher who did some important work on the Bd-Tree and Florian Köhl who did excellent work in the field of temporal query processing (which was an important side-project during my doctoral studies but is not part of the work presented in this thesis). I also want to thank Lucas and Renato who worked on Tell and TellStore. Lucas' work on AIM (Analytics in Motion) influenced my work a lot, and he helped during the implementation and the experimentation.

Last but not least, my biggest gratitude goes to my wife, Sonja. Her support was essential for my work as well as for my private life.

# Contents

# I TellStore - A Storage for mixed Workloads

# 1

# Introduction

Der Starke ist am mächtigsten allein.[1]

(Friedrich Schiller, Willhelm Tell)

Verbunden werden auch die Schwachen mächtig.[2]

(Friedrich Schiller, Willhelm Tell)

## 1.1 Background and Motivation

For a lot of emerging applications, data warehousing is reaching its limits. With the rise of cloud computing and the drop of storage prices, it gets easier to collect data. However, analyzing this data is still a huge challenge.

Data warehousing brings high costs for hardware and maintenance. Within a cloud, one would like to have an elastic system so that these costs only need to be paid for while the warehouse is used. This, however, is not achievable without an elastic system.

Another problem arises when real-time analytics becomes a requirement. The data within a classical warehouse does not hold the current data, but it gets updated periodically.

Updating the data warehouse is usually done with *ETL* (Extract, Transform, Load): the data is extracted from a parallel running online transaction system,

[1] The strong man is strongest when alone.
[2] Even the weak men are strong when united.

transformed in a read-optimized schema, and then loaded into the data warehouse. This process, however, is non-trivial and requires a lot of engineering and maintenance to work correctly.

All in all, data warehousing is a very heavy-weight solution.

To solve some of these problems, new database systems emerged from industry and research that can run mixed workloads on a single machine. Examples are HyPer [KN11] and AIM [Bra+15]. Single machine solutions, however, are not the answer to all these problems.

For a cloud solution, elasticity is a hard requirement. Without elasticity, multi-tenancy is virtually impossible to achieve. Furthermore, as scaling the system while it is running is not possible, a user has to overprovision resources to get the desired response time and throughput. This increases the cost per operation significantly compared to a system that can scale down during times when the load is low.

Another problem with single machines solutions is that scaling a system up is expensive. Also, with new hardware trends, single machine systems face similar challenges than distributed systems: CPU speed is not improving significantly. Instead, hardware vendors put more CPUs into a single box. As the number of CPUs within a system increases, memory communication and inter-CPU communication becomes challenging. A design that works around these problems is *NUMA* (Non-uniform memory access): the memory is split between the processors and access time depends on the memory location relative to the processor. These systems can still be programmed as if it would be a multi-core system with uniform memory. This will, however, result in reduced performance compared to a system that is NUMA-aware. As a consequence, a single machine has to be programmed the same way as a distributed system: each CPU has its own memory region and communication between these regions is more expensive than local memory access. As the CPU count increases, this effect can be expected to get even higher. At the same time, network latencies are still declining. Current Infiniband networks provide network latencies which are only around one order of magnitude slower than memory latencies between two NUMA units.

Distributed solutions, however, mostly optimize for a particular workload. Systems like VoltDB/H-Store [Kal+08] or MySQL Cluster optimize for online transactions and systems like Hadoop, Spark, and Presto optimize for analytical workloads.

## 1.2 Problem statement

We want to have a system that can handle mixed workloads on *one* logical copy of the data. We only want to copy data for durability and availability. As this system is designed to run in a cloud environment, it has to be elastic and scalable.

We want to be able to separate storage resources from computing resources. In such a system, it has to be possible to add computing resources at run time for lower response times of analytical queries, to improve the transaction throughput for online transactions or both. At the same time this has to work on the other layer as well: if the data shrinks, it has to be possible to remove storage resources from the storage layer or add more storage if the database grows in size. Optimally this should be doable without adding cost for additional computing resources if these resources are not needed. A system with these properties is what we call elastic.

Optimally, on a scalable system, the cost per transaction stays constant independent of the load. The throughput has to increase linearly on a scalable system. For analytical queries, the response time should decrease linearly as more computing resources are added to the system.

## 1.3 Contributions

Analytical and transactional workloads impose different requirements on an underlying storage system and on the processing engine. For online transactions, we want to optimize for a high transaction throughput, many point queries, small range queries, and many small updates. However, analytical workloads usually require a high data throughput.

The first and largest contribution of this thesis is TellStore, a key value store specifically designed to support mixed workloads. As part of the work on TellStore we explored the design space for a storage engine which can sustain a high get and put load while providing low response times for concurrent full table scans. We were able to show that it is possible to build a storage engine that can achieve high scan performance with making only minor sacrifices to its get and put performance.

To support secondary indexes, we designed and built a distributed, lock-free B-tree, called Bd-tree. This data structure is based on the Bw-tree presented in [LLS13].

We adopted the distributed shared-data architecture for mixed workloads on a single copy of data and provide a comprehensive performance evaluation over the explored design space.

We invested a lot of engineering effort and published a working system called

Tell as Open Source on github[3]. Tell is an elastic and scalable database management system that can run mixed workloads.

Tell is designed for RDMA. We designed an implemented Infinio which is a threading runtime and network layer designed to work over RDMA networks with a minimal computing overhead. Infinio reduces the CPU overhead significantly compared to all other comparable libraries which are known to us. We achieved that without making compromises for ease of use: a user can program with Infinio as if it is a blocking socket interface. Infinio achieves its high throughput by implementing aggressive batching of requests between user-level threads.

## 1.4 Overview of the thesis

The first chapter of this thesis defines the architecture and give a high level overview of the system. The rest of the thesis is divided into three parts:

1. The first part of this thesis describes the storage. We specify the requirements of a storage engine for a storage that can be used to implement transactions and can handle scans for analytical transactions. Furthermore, the design space for such a storage is explored and all design decisions explained. We implemented three different storage engines, each covering a different part of the design space. The last chapter of this part shows experimental results where we compared the different approaches with each other.

2. The second part of this thesis explains how indexing and transactions are implemented on top of the storage engines described in the first part.

3. The last part brings together the components from the first two parts and demonstrates how they can be used to run mixed workloads on Tell.

The thesis finishes with an overview of possible future work and conclusion.

[3] https://github.com/tellproject/tell

# 2

# The Shared Data Architecture

## 2.1 Introduction

### 2.1.1 Finding the Right Architecture for the Right Job

As for any system, choosing the right architecture is not only the first step, it might be one of the most important ones. For a distributed database, it is inherently hard to get right and each architecture seen implemented so far seems to have distinct benefits and drawbacks.

Who should have ownership of which part of the data? Where should computation happen? How is consistency guaranteed? How are machine failures handled? How can the system scale? How can it be made elastic? While the architecture by itself might not answer these questions, it restricts the set of possible solutions.

### 2.1.2 Requirements

A database management system designed for the cloud has to scale and has to be elastic. A database system designed for mixed workloads has to be able to handle complex analytical queries as well as simple, short-running, localized queries and updates. Furthermore, we want to achieve these goals without relying on copies of the data. So whatever architecture we chose for such a system needs to provide us with enough flexibility so that we can find a solution to all problems dictated by these requirements.

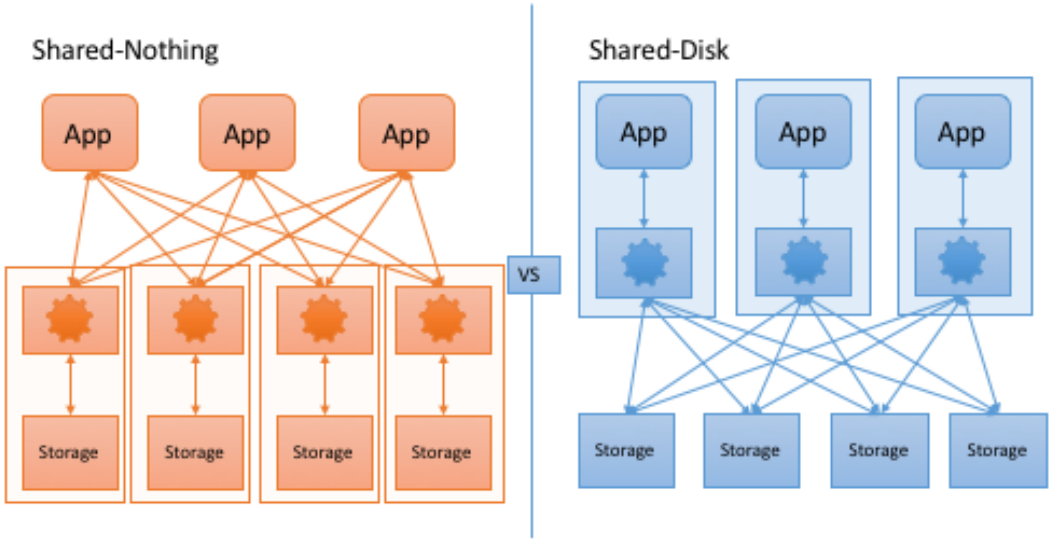



Figure 2.1: Shared-nothing (left) and shared-disk (right) architecture

### 2.1.3 Contributions

This chapter introduces the shared-data architecture which is a mix of two well-established architectures used in many database management systems. I will argue that by choosing the right heuristics for storage and compute distribution, it is possible to design a system that can satisfy the requirements outlined in section 2.1.2. This architecture simplifies the design and implementation of a distributed database system for mixed workloads and will guide all work presented in the following chapters.

## 2.2 Shared Nothing vs Shared Disk

The two most common architectures for distributed database systems are illustrated in fig. 2.1. In a shared-nothing database, each machine owns a partition of the data which is stored locally (on a disk or in-memory). A system implementing the shared-disk architecture stores all data in a shared disk which is then accessible to all machines that are executing transactions. The following paragraphs give a brief overview how such systems are usually implemented and discuss their advantages and disadvantages. The differences between these two architectures are summarized in table 2.1.

### 2.2.1 Shared Nothing

The shared-nothing architecture was most famously introduced by Micheal Stonebraker in [Sto86] and is probably the most popular architecture for OLTP systems.

The main idea is to use one full-featured database management system per machine or, as in [Kal+08], even one per core. Each instance has ownership over one partition of the data. The partitioning of the data is chosen in a way that most (or ideally all) transactions only need to read and write from one partition. If that is the case, a transaction can run in complete isolation on one partition - i.e. one machine or even one core.

In other words, there is no separation between compute and storage, or most complexity is pushed to the storage layer. Shared nothing databases are usually optimized for a high transaction throughput and low latency. Furthermore, shared-nothing database systems often allow a high write load, as single-partition read-write transactions can be executed without inter-partition communication.

Shared nothing works great as long as it is possible to find a good partitioning for a given workload. But as soon as some transactions need to write data from several partitions, some form of inter-machine communication needs to be done. Usually, this is solved by introducing an additional federation layer. This layer will send the queries of an inter-partition transaction to each machine involved in the execution of the transaction and aggregate the results. In order to serialize such a transaction, a consensus protocol is needed (e.g. two phase commit [LS79] or Paxos [Lam78]).

This architecture is mostly used for Online Transaction Processing (OLTP), as these workloads require low latency and high transaction throughout. Furthermore, OLTP workloads often allow fine grained partitioning of data. But this partitioning requirement is also the biggest weakness of the architecture. Sometimes it is not possible to find a suitable partitioning for a given workload which will result in a performance degradation. Especially analytical queries usually need to read more than one partition of the data.

The next big drawback of the shared nothing architecture is the lack of elasticity. Adding and removing storage nodes is non-trivial since the data needs to be repartitioned. Repartitioning might not even be possible at a certain point since the workload might not allow a fine-grained enough partitioning. Even if it does, it might not be possible to partition the data evenly over the storage nodes.

Comparing these strengths and weaknesses of the architecture with the requirements listed in section 2.1.2 shows that it will either impossible or very hard to design a shared-nothing system that can fulfill all requirements. It can run OLTP-style workloads and it can scale out well. However, elasticity will be hard to implement and concurrent long-running queries with a high data throughput might be problematic for such a design.

### 2.2.2 Shared Disk

On a shared disk (or sometimes called shared everything) system, every database node has a global view of all data. The data is then stored on a (potentially distributed) disk that is accessible over the network. Examples of such systems are Oracle RAC [Smi03] (which uses a shared SCSI, SAN, or NAS device to store the data), Googles MapReduce [DG08] (which uses the Google File System [GGL03] as a storage), Hadoop [Shv+10] (which is essentially an open source implementation of Googles MapReduce and the Google File System), and Apache Spark [Zah+10].

While shared nothing systems push all complexity to the storage layer, shared disk systems try to keep the storage as simple as possible. A storage node does not execute transactions, but it can just be used to read and write simple blocks of data. The transaction processing is then implemented in a higher layer which we call the processing layer. It can consist of several machines and each transaction is executed by one or many machines. Each machine within the processing layer has a global view of all data; this means that each processing node can potentially read and modify all data in the system, and no machine has explicit ownership over a partition of data.

These systems are usually optimized for high data throughput. An arbitrary number of processing nodes can process all data concurrently. This makes this architecture attractive for analytical workloads: several machines can execute one query and work on a partition of the data. But the partitioning of the data does not need to be made beforehand. Each processing node can just read its partition from the disk and produce a result and send this back to some master node for the final aggregation.

Shared disk databases are, unlike shared nothing systems, elastic. Adding and removing processing nodes is trivial, as they typically are mostly stateless. Adding and removing storage nodes gets much easier as well: since the processing layer is oblivious to how data is partitioned across storage instances, repartitioning can always be done without any user interaction and independent of the workload. For the same reason, scaling becomes easier as well. These features make a shared disk database system attractive for analytical workloads.

In practice, shared disk systems suffer from higher latencies than shared nothing systems. This is mostly because the data is physically further away from the CPU that is processing a query. Concurrency control gets harder as well, as locking needs to be done either in the storage (by taking locks in the file system) or a central locking service needs to be in place. The next problem is caching:

| **Shared Nothing** | **Shared Disk** |
|---|---|
| Complexity in storage layer | Complexity in processing layer |
| Optimized for partitioned workload | Optimized for worst-case |
| Low latency | Higher latency |
| High transaction throughput | High data throughput |
| Most popular for OLTP | Most popular for OLAP |

Table 2.1: Comparison of shared nothing and shared disk

while a processor in a shared nothing database can cache virtually the whole database without problems (the only bottleneck is the physical size of the cache), a shared disk system can never be sure that its cache was not invalidated by another processing node. Therefore it often needs to reread data from disk to ensure data freshness. These drawbacks make the shared disk unattractive for OLTP workloads.

To conclude, a shared-disk system can easily satisfy some requirements of the ones listed in section 2.1.2: it can scale, it is elastic and it can handle complex analytical workloads. But most shared-disk systems run into issues when it comes to OLTP-style workloads.

### 2.2.3 Copying data

As argued in [SÇ05], it is hard to build systems that perform well for different kind of workloads. Because of this, companies today are often using two database management systems: one optimized for OLTP and another one optimized to run analytical workloads. The OLTP system contains the live data which is then copied to the OLAP system. This copying can be done either with every write operation or in one batch at certain time intervals. In that case, for example, every day or every week, all data is extracted from the OLTP system, then potentially transformed into a read-optimized schema and finally loaded into the OLAP system. This process is, therefore, usually called ”Extract, Transform, Load” (ETL).

As stated in section 1.2, the goal of this work is to provide a system that can run mixed workloads without relying on copies of data. ETL has its drawbacks; the most important one is data freshness. Analytical queries will only see data that is as old as the snapshot that was copied into the analytical database system. One solution to this could be to copy the data of each transaction at commit time. While this approach is valid, it still has the drawback that one needs twice as much

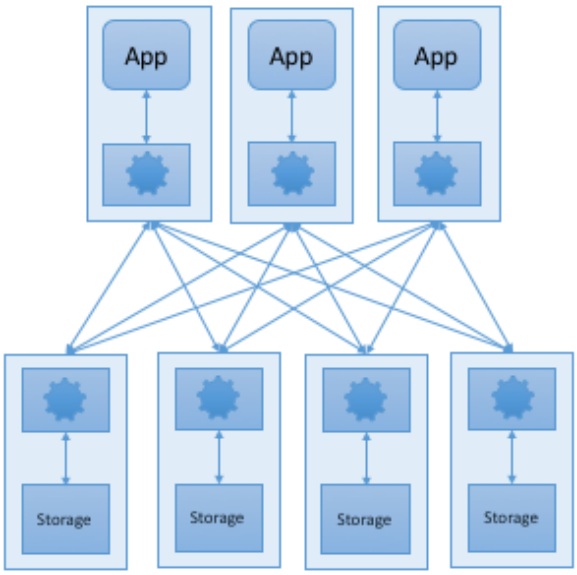



Figure 2.2: The Shared Data architecture

memory to run mixed workloads and making a copy of the data during each commit is hard by itself. Therefore, running all queries on the same data is not just a design decision, it is a feature.

One might argue that we usually want to replicate data anyway (for availability and higher read throughout). Therefore, we could just replicate the database state into another storage format which is better suited to answer analytical queries than the OLTP system. But it is not clear how to achieve that. First of all, having a copy in different data formats is not equivalent to copies in the same data format: if an OLTP copy crashes, the analytical system would not be able to handle OLTP load and vice versa. Another observation is that copying data in real time while bringing it into a different format is by no means trivial and still needs research. Some open questions are: how do we handle analytical transactions that are not read-only - or do we simply not support this? How do we make sure that all OLAP replicas are in a consistent state so that analytical transactions are executed in serial with OLTP transactions?

## 2.3 Shared Data

This thesis argues that shared data is the right architecture to handle mixed workloads. But before doing that, shared data needs to be defined. The architecture is illustrated in fig. 2.2. On first sight, this architecture looks very similar to the shared disk architecture. The main difference between shared data and shared disk

| | **Shared disk** | **Shared data** | **Shared nothing** |
|---|---|---|---|
| Time complexity | $O(1)$ | *For get/put:*<br>• $O(1)$ average (hash)<br>• $O(log(n))$ (tree)<br>*For scan*:<br>• $O(n)$ | Unbounded |
| Space complexity | $O(1)$ | $O(1)$ | Unbounded |

Table 2.2: Complexity comparison of storage layers

is that some functionality is implemented in the storage layer. Instead of serving files (or disk blocks), the storage is serving tuples indexed by a key. Shared data is a compromise between shared disk and shared nothing.

This idea is not new and is often also referred to as "SQL over NoSQL". NoSQL systems often implement a shared-nothing architecture. SQL over NoSQL does, however, imply that the SQL-process has a global view on data. Usually, the underlying storage system is a key value store, but according to our definition, it can be arbitrarily more complex than a disk. E.g. Hyder [BRD11] uses a shared log instead of a key value store to store data. One has to be careful though: if too many features are pushed into the storage layer, we might end up with a shared nothing architecture again (where the processing nodes have the role of a simple federation layer). So the boundary between shared data and shared nothing needs to be defined. Throughout this thesis, shared data is defined as stated in definition 2.1.

**Definition 2.1** A distributed database system implements the shared data architecture if all operations supported by the storage back end can be executed in $O(n)$, where n is the number of tuples stored in the database, while the space overhead of each operation is constant.

Therefore, shared data is just a generalization of the shared disk architecture. On a disk, each operation can be executed in constant or, in case of a seek, in linear time. A shared nothing architecture is not a shared data architecture since the storage is a full-featured database system by itself where each operation can be unbounded in complexity. As an example, sorting cannot be executed in linear time. But the storage of a shared data system can be more feature rich than a simple disk: it can evaluate predicates at scan time, do simple aggregations, and projections. To illustrate the differences between shared data, shared disk, and

| | Disk | Shared-Data Store | Relational Database |
|---|---|---|---|
| Transaction support | ✗ | ✗ | ✓ |
| Selection Push Down | ✗ | ✓ | ✓ |
| Projection Push Down | ✗ | ✓ | ✓ |
| Aggregation Push Down | ✗ | ✓ | ✓ |
| Join & Group By | ✗ | ✗ | ✓ |
| Secondary Indexes | ✗ | ✗ | ✓ |

Table 2.3: Feature comparison of storage layers

shared nothing table 2.3 lists the features of a disk, a relational database system, and a shared data storage system, an implementation of which will be presented later in this thesis.

Definition 2.1 means that any shared-nothing system could be used as the underlying storage as long as only features that execute within linear time are used by the processing layer. The crucial aspect of shared-data is decoupling of storage and compute as only cheap computational operations are executed within the storage.

This limitation simplifies scalability and elasticity on both layers. On the compute layer because each machine has tiny state. On the storage layer because dependencies between data and operations are limited and the overall architecture is simplified.

## 2.4 Potential Bottlenecks in a Shared Data System

No matter how well a system is designed and implemented, at one point, it will run out of resources. In the best case, this will result in reaching peak throughput and increasing response time. In the worst case, the system throughput will stall, as all processes will fight for the available resources.

Systems do not run out of all resources simultaneously, but one or few resources will limit the system's ability to handle the load it is experiencing. This resource is called the bottleneck. As a countermeasure, the system administrator can add resources (in the form of additional or faster hardware) to the system. To do so, one needs first to understand the bottleneck - it will not be helpful to add more machines if a network switch can not keep up with the network load and replacing the CPUs with faster models will not bring improvements if the memory controller

reached its limits.

In this section, I will outline the bottlenecks a shared data system might hit:

- Network
- CPU
- Memory controller

Another reason why the system might stall is resource contention: If a lot of transactions try to read or write the same data, the system will not be able to use concurrency to answer requests. Read operations will all hit the same machines, degrading the performance while the other machines have plenty free resources. This can be solved by replicating the data or by using a caching layer, for example, memcached.

If a lot of concurrent transactions try to update the same data, the system has to serialize these updates. If the system implements a form of optimistic concurrency, it will abort a lot of these transactions which will result in low throughput. If the system implements a form of pessimistic concurrency control (for example latches), the system will serialize these transactions resulting in a higher response time.

Write contention can not generally be solved. But the problem can be reduced if we can reduce the response time for transactions. It could be done by using CPUs with a higher single core performance (but uni-core speed will at best increase modestly in the foreseeable future) or with intra-transaction concurrency. Short transactions, however, do not experience speed-up with more concurrency, they typically become even slower because of the concurrency overhead.

### 2.4.1 Network bottleneck

Tell was designed for RDMA and built to use Infiniband which is a popular network technology that implements RDMA. On Infiniband, there are mainly two limiting quantities: the throughput and the message rate.

The throughput is usually not a problem for an OLTP workload: e.g. with TPC-C, Tell generates only a few hundred megabytes of network throughout when it reaches 1 million TpmC. Of course, this story will change for analytical workloads, therefore we try to minimize the utilized throughput as well as possible. The best way to reduce throughput is to make sure that only data that needs to be sent over the network gets sent. As most other record-management layers do, TellStore supports projections and selections. Therefore we can execute full table scans mostly on the storage layer, and the storage will only send data over the network

that is needed to execute the query. This is the main feature that distinguishes TellStore from most other key value stores.

Another popular way to reduce network throughput is caching. Caching often also has the desired side-effect that it decreases latency since data can be read from local memory. TellDB does caching on the transaction level: whenever a transaction reads a tuple from the key value store, it saves it locally. The updates are then only applied locally and written back at commit time. Therefore updates are only written once, and tuples are only read once per transaction. However, as shown in [Loe15], inter-transaction caching hurts the overall performance. Therefore, each transaction maintains its own cache, even if other transactions are executed on the same machine.

The message rate is a much more prevalent bottleneck for OLTP workloads. Infiniband can handle a few million messages per second (the exact number depends on the Infiniband hardware and the network stack implementation). We spent a lot of time in optimizing this with Tell 2.0 (this will be explained in more detail in chapter 8). Our main counter measure is to pack messages together so that we can send as big messages as possible (message batching).

We also only use lock-free data structures, since global locks will introduce new messages that need to be sent through the network. In contrast to locks, lock-free data structures help us to reduce the message rate to a minimum. If no conflict occurs, a message needs to be sent only once. Only if a conflict is detected, more messages are needed as the algorithm needs to either retry or roll back. The problem here is, however, that with a high load, the number of atomic operations that fail increases. For transactions, we will just abort and roll back as soon as a conflict is detected, for other operations, we have no other choice than to retry until the operation succeeds.

Our only countermeasure for retries of atomic operations is to try to engineer the implementation of the algorithms as efficient as possible. We spent a lot of time to optimize the code paths between successful atomic operations.

### 2.4.2 CPU

To discuss CPUs as a potential bottleneck we need to differentiate between CPUs within storage machines and CPUs within processing machines.

Adding CPU power to a system can be done by either using faster CPUs or adding more CPUs. The first is not possible, as CPUs do not get (much) faster anymore. Adding more CPUs is only helpful if the work can be done in parallel. Tell is designed to be highly scalable, so adding CPUs should bring a

linear performance benefit up to a certain point. This is also an important reason why distributed systems are becoming so important: the more CPUs we add, the more the system starts to look like a distributed system. It does not matter whether the CPU cores are within the same box. Eventually, communication between CPUs slows down due to the increasing physical distance between them. Todays NUMA technology started this trend, as communication between NUMA units is significantly more expensive than between CPU cores within the same unit. A low latency network like Infiniband is just another layer of indirection to communicate between CPUs within the system. Tell is built to scale to rack size (or may be a few racks - but we do not have the resources to test the scaling limits of Tell).

On the processing level, TellDB runs each transaction in its own thread. One could, therefore, allocate one CPU core per transaction. For short running transactions which do not involve a lot of computation (as it is usually the case for OLTP workloads), this degree of parallelism is more than enough. Furthermore, OLTP transactions are mostly IO bound. For analytical queries, this assumption does not hold. To use inter-query parallelism, Tell provides a connector to Spark and a Presto plugin. Tell does not provide its own distributed execution engine and this is considered to be out of scope for this thesis.

The storage machines usually do not need to execute complex computations for OLTP workloads as in that case, the storage is only answering to simple get and put requests. In the processing layer, one can just add more CPUs or machines.

The story changes however for analytical queries. Analytical queries are often very CPU intensive. Tell implements two counter measures: The heuristic shown in table 2.2 help control the CPU load on the storage. Furthermore, we try to do as many computations as possible with each CPU cycle. To do that, TellStore compiles every scan query to highly optimized code. In the processing layer, every query can be run on several machines which allows scaling out CPUs. Furthermore, one can add more CPUs to the storage to speed up scans, as TellStore can scale linearly with the number of CPU cores it can use for scans.

### 2.4.3 Memory Controller

For OLTP workloads, the memory controller is usually not a bottleneck, as only small amounts of data are read for each transaction.

For analytical workloads, however, scans will put a high load on the memory controller. While evaluating a scan, we need to ship a lot of data from main memory to the CPU which will then evaluate predicates and materializes the result. If the evaluation of the predicates is expensive, the CPU will become the main bottleneck.

To minimize the load on the memory controller, TellStore implements a columnar layout. This means that it does not need read all data but only the columns a query is interested in. As another countermeasure, TellStore implements shared scans. Shared scans trade response time for better caching behavior: one column might be needed to calculate several query predicates. If that is the case, we need to fetch the data only once into the CPU and can evaluate all of them at once.

# TellStore - A Storage for mixed Workloads

# 3

# Designing a Storage for Mixed Workloads

## 3.1 Introduction

In this chapter, I discuss the general requirements for storage for a shared-data database. These requirements did dictate the implementation of TellStore. But any storage layer to be used for mixed workloads in the cloud has to fulfill these requirements.

Formulating requirements has to be done cautiously. If the chosen requirements are too weak, the system might not work correctly. If the requirements are too strong, the implementation might become too complicated and slow.

This chapter does not by itself present new ideas. The function of this chapter is to outline the characteristics that any storage layer for a shared data database should have. This analysis guided us in the creation of TellStore - which will be presented in detail in later chapters. The content of this chapter is currently under review for publication.

## 3.2 Requirements

The idea to use a distributed NoSQL database as a storage for a SQL database is not new and often referred to as SQL over NoSQL. Examples of such systems are Presto [Fac16] and the SQL layer for FoundationDB [Fou13] (which was discontinued after the company got bought by Apple).

However, most of these systems got developed bottom-up: a NoSQL store

was designed independently and a SQL layer was added later. As a result, these systems are optimized for a particular set of workloads and the SQL layer have to be adopted to the storage layer.

In contrast, we designed the system top-down: we started with a processing-layer (which is described in [Loe+15]) and with that knowledge designed a storage that could execute all workloads well. We learned that it is important to design both layers of the system simultaneously.

This section describes the requirements for such a NoSQL-layer. While the underlying layer can be used without the SQL layer, it is designed to perform well if used with a complex processing layer.

#### GET/PUT

This is probably the most obvious feature and the main functionality of any key value store as well. The storage must provide the possibility that the client can define some form of immutable key which points to a chunk of data (in the traditional relational database world, we would define a table with some primary key and a BLOB). This means that the storage must provide an index on the primary key. The *get* operation is then used to fetch the associated value of a given primary key value, and *put* is used to write a new value for a given key.

Most existing key value store can use arbitrary keys (e.g. strings or byte arrays). This is, however, not a requirement. The only strong requirement is, that the key is wide enough to allow storage of enough data. Therefore a mapping from integer to a byte array is enough to implement a SQL store above it, but the integer should be eight bytes wide to allow storing more than four billion key-value pairs.

#### Atomic Operations

While some key-value stores provide support for ACID-transactions, there are benefits in implementing these on a higher level. Implementing transactions in the processing layer has some advantages: it simplifies the storage layer drastically and non-transactional operations do not need to be implemented separately. For example, a secondary index implemented within the processing layer does not need transactional operations.

However, the *get/put* operations must be atomic. This is a much more relaxed requirement than transactional support, but it is still stricter than other consistency guarantees seen in some key-value stores (e.g. eventual consistency is too weak).

Whenever a write request arrives at a storage node, it will either fail or succeed (all or nothing), if it succeeds, all subsequent read requests see the updated version of the tuple. The ordering of the requests is done at the storage, no ordering of

requests from separate clients needs to be enforced. This means that if two writers issue a write request for the same tuple simultaneously, the storage can decide on the ordering. This has implications for replication. If replication is used, the system needs to make sure that all clients always read the newest version of a tuple.

Additionally, the processing layer also needs to be able to write data under the condition that the tuple did not change after it was read. In this work, I will refer to this feature as compare and swap (CAS).

However, compare-and-swap is just the softest requirement for a storage to be used within a transactional system. Another, stronger form, of an atomic operation, is load-link/store-conditional (LL/SC). LL/SC gives stronger guarantees, as it prevents the ABA problem: an atomic compare and swap might successfully execute if the value changed several times in between.

In a distributed environment it is easier and more efficient to implement LL/SC than CAS: for CAS the old value has to be shipped to the storage together with the write request so that the storage can make a comparison. For LL/SC, something potentially smaller can be used, like a sequence number to identify the previous state of the value.

#### Scan Operator

Unlike a typical OLTP workload, analytical queries usually need to fetch a lot of data from the storage. It is possible to use a secondary index to fetch whole tables. However, this is inefficient for two reasons: the storage node has to issue one random read operation per tuple and applying a predicate while scanning on the server side is not possible - this further increases the load on the network.

Therefore, an efficient system needs a scan operator on the storage level. As described in section 2.3, such a scan operator can execute selections and simple aggregations.

#### Durability and Availability

The D in ACID stands for durability. This means that all write operations within a successfully committed transactions will be readable after a crash or power failure. Usually, this is done by writing to a disk.

A lot of key-value stores write to disk asynchronously to speed up write operations. An ACID system, however, needs to execute all write operations of a transaction before it commits.

#### Scalability and Elasticity

Due to its simplicity, almost every key value store is scalable and elastic. With scalability we mean that adding more hardware resources will translate to more performance (for example higher throughput or lower response time). Elasticity means that adding and removing hardware resources during operations is possible.

As mentioned in section 2.1.2, a system for the Cloud has to be designed to scale and for elasticity. This requirement is the main motivation why we want to eliminate as much complexity from the storage layer as possible.

#### Versioning

As shown in [Loe15], snapshot isolation is the most efficient algorithm for transaction execution within a shared-data architecture. Furthermore, it has the advantage that write-operations do not block long-running read-operations. Because of these two reasons, a system designed for mixed workloads has to implement snapshot isolation for concurrency control.

Multi-version concurrency control (MVCC) is a popular way to implement snapshot isolation. While it is possible to implement MVCC in the processing layer (as shown in [Loe+15]) it is preferable to push some of this functionality to the storage.

Otherwise, all versions of each tuple need to be sent to the client on each request, which will increase load on the network. Our storage has to implement MVCC and the processing layer will implement the other parts needed for transaction execution. Or in other words: the storage implements versioning to speed up snapshot isolation.

Versioning is a broad term. What we mean by versioning is that the storage organizes key-value pairs as follows:

- For each key, the storage keeps a set of version-value pairs.
- Whenever a client wants to read data, it needs to send the set of versions (called read set) it wants to read from. The storage will then send back the value with the highest version that is in the read set.
- For write operations, the client sends a key-value pair it wants to write, the version the new tuple should have, and a read set. The storage will then look for the key and only apply the update, if for the given key, there either does not exist such a tuple or the newest version of the tuple is also contained in the read set delivered by the client.
- There is a garbage collector that cleans old versions from the storage. How

the storage decides which versions to clean can be configured (this is described in more detail below).

Versioning can also be used to implement a compare and swap feature within the storage. Therefore, if the system implements versioning, we can drop the requirement for atomic operations. One example of a key-value store that uses versioning to implement atomic operations is RAMCloud. However, the garbage collection mechanism has to know which versions it can erase, which would not be the case if only atomic operations were to be supported.

## 3.3 Why is this difficult?

The big problem of these requirements is that they are in conflict. That is why most key value stores today (with the notable exception of Kudu) have been designed to support *get/put* requests only (e.g., Cassandra and HBase), possibly with versioning (e.g., RAMCloud) and sometimes with asynchronous communication. All these features are best supported with sparse data structures: To read a specific version of a record as part of a *get* operation, it is not important that this record is clustered and stored compactly with other records. Scans, however, require a high degree of data locality and a compact representation of all data so that each storage access returns as many relevant records as possible. Specifically, adding scans creates the following locality conflicts:

- *scan vs. get/put:* Most analytical systems use a columnar storage layout to increase locality. KVS, in contrast, typically favor a row-oriented layout in order to process *get/put* requests without the need to materialize records [Sto+05].
- *scan vs. versioning:* Irrelevant versions of records slow down scans as they reduce locality. Furthermore, checking the relevance of a version of a record as part of a scan is prohibitively expensive.
- *scan vs. batching:* It is not advantageous to batch *scans* with *get/put* requests. OLTP workloads require constant and predictable response times for *get/put* requests. In contrast, scans can incur highly variable *latencies* depending on selectivities of predicates and the number of columns that are needed to process a complex query.

Fortunately, as we will see, these conflicts are not fundamental and can be resolved with reasonable compromises. The goal of the next chapter is to study the design space of key-value store and lay out compromises.

# 4

# TellStore

Tell uses a key value store for storing and retrieving data. Key-value stores (KV stores) are getting more and more popular. Unlike traditional database systems, they promise elasticity, scalability, and are easy to set up and maintain. Furthermore, the performance characteristics of a KV store are easy to understand which facilitates reasoning about and understanding performance numbers of systems that rely on a KV store. In essence, the first and foremost advantage of KV stores is their simplicity.

The goal of Tell was always to run mixed workloads on the same data. Currently, systems usually achieve this by copying data from an OLTP optimized database systems to an analytical system doing Extract, Load, Transform (ELT) - copying the data. Tell, however, can run analytical workloads on OLTP data without making a copy of it.

| **Key Value Store** | **Scan Time** |
|---|---:|
| Kudu | 27 s |
| RAMCloud | 196 s |
| HBase + Spark | 316 s |
| HBase | 23 min |
| Cassandra | > 2 hours (time out) |
| *TellStore - log* | *391 ms* |
| *TellStore - row layout* | *466 ms* |
| *TellStore - columnar layout* | *82 ms* |

Table 4.1: Simple scan on popular key value stores

To illustrate that current state-of-the-art KV stores are not properly optimized for scans, we developed a simple micro-benchmark: We first populated a KV store with 120 million tuples. Each tuple was 1kB in size and contained a small random number. We then took three state-of-the-art KV stores that provided a scan-like interface and let them calculate the sum of all these random numbers. We used four machines to run the KV store. The results are shown in table 4.1.

While Cassandra was not able to answer this query within two hours, HBase took 23 minutes to process it. Since HBase stores its data in HDFS, we then took Spark to speed up the scan. This brought the response time down to 316s which is still prohibitively high. RAMCloud was the fastest of the tested (pure) KV stores with a response time of less than 200 seconds.

Kudu is a special case here: it tries to solve the same problem as TellStore: providing high scan performance while retaining high get/put throughput and low response time for point queries and updates.

Having fast scans is essential to provide support for fast analytical queries. As shown in table 4.1 it does not seem to be trivial to implement a system that provides fast get and put access and fast scans.

In this chapter, I will explain how we developed three key value stores that provide a scan feature. We analyzed the design space of key value stores and came up with these three implementations to better understand the dichotomy of point queries/updates and scans. As it turned out: this dichotomy is much smaller then we expected. The main reason current key value stores perform so badly with scan queries is probably that they were designed without taking scans into consideration. This indicates that a lot of key value stores either added scans late during the development or don't provide a scan feature at all. From our three implementations the clear winner was a key value store that used a columnar data layout to store the data. It is not surprising that a column layout is beneficial for scan performance, but we were able to show that such storage can also provide a competitive get/put performance.

The work presented in this chapter is currently under submission for publication.

## 4.1 Features

Tell's architecture imposes certain requirements on our storage layer. To process the indexes, we need singe-tuple compare-and-swap operations. As latency is a crucial factor, TellStore is specifically designed for Infiniband networks and RDMA. The other requirements are outlined in the following paragraphs.

### 4.1.1 In-Memory

TellStore is an in-memory storage system. In-memory systems deliver higher throughput and lower latency at the cost of more expensive memory hardware and higher energy costs. For durability and availability, the data is then replicated and stored in non-volatile memory. How to do this is out of the scope of this work, but very well covered in existing work, for e.g. [Geo11; LM10; RKO14].

### 4.1.2 Shared Scans

As described in[Unt+09], shared scans can help to keep our operations behaving in a predictable manner. Without a shared scan, the number of threads executing scan requests can get out of hand - slowing down concurrent get/put operations. The main drawback of sharing is that slow queries will slow down everybody. By implementing shared scans, we are trading throughput and predictability for a higher response time.

### 4.1.3 Predictability

What makes KV stores so useful is that they provide predictable response time. Each request, get or put, finishes in constant time and consumes a small, constant amount of resources - i.e. main memory on the storage machine, CPU cycles, and network bandwidth.

This is also true for more complex systems, like relational database management systems, if only get and put requests are executed. However, the simplicity of KV stores allow to better optimize only this functionality. More complex systems might introduce additional overhead to these operations (like SQL parsing, transaction execution etc). This often makes them less efficient for simple get/put workloads and in general harder to handle.

In TellStore, get/put requests run in constant time and consume constant space on the storage side. Scans, however, must run in linear time and constant space. This implies that we can do simple aggregations, selections, and projections directly in TellStore, but more complex operators, join and group by, have to be implemented in the processing layer.

### 4.1.4 Lock- And Latch-Freeness

To allow efficient concurrent accesses, it is prohibitive to use locking in any place. Neither the processing nodes nor the storage nodes should take any locks, and no locking must take place between machines. This requirement makes the implementation of some potential KV store designs very difficult or virtually impossible.

The absence of locks is important to guarantee scalability on multicore machines. The existence of a scan makes lock-freeness even more important as we need to reduce the fight for resources between get/put operations and concurrent scans to a minimum.

### 4.1.5 Consistency

For a shared-data database, one has to carefully select which features run in the storage layer and which are implemented in the processing layer. Secondary indexes and transactional processing are implemented in the processing layer (this is described in detail in [Loe+15] and later in this thesis). The storage layer, however, must provide some basic consistency and synchronization features to allow for an efficient implementation of these features in the processing layer. Tell is a complete ACID database system and TellStore is one important component of this system. To allow transaction processing, TellStore must provide strong consistency guarantees. For synchronization, the storage needs a feature to allow for conflict detection when writing back data.

### 4.1.6 Durability

Tell currently does not support durability. We recognize that this is a major drawback of the current implementation. Instead, section 4.7 will sketch one possible implementation of this feature. The reason for not providing this feature is that we see this as an engineering task.

### 4.1.7 Versioning

Scans might slow down *get/put* requests. We take several countermeasures to minimize these effects. One is shared scans, and the other is versioning. It is difficult to synchronize put requests with running scans. Furthermore, we want to be able to scan a consistent snapshot of the data. Since our processing layer implements snapshot isolation, we can push the versioning feature into the scan. This makes the implementation of the processing layer simpler while allowing lock-free scans in the storage layer.

### 4.1.8 Resource Provisioning

A big difficulty with mixed workloads is that a high OLTP load might slow down analytical queries or analytical queries could lower the OLTP throughput. We, therefore, try to provision the available resources for certain workloads. This can

not be done perfectly. We don't have direct control over how system resources like L3 cache and memory bus are shared between processes.

The overall architecture is illustrated in fig. 4.1. Each thread is pinned to one execution thread. There are three different kinds of threads: *get/put* threads, scan threads, and one garbage collection thread. Whenever a client opens a connection, one of the *get/put* threads accepts the connection. This thread is from that point on the only thread listening to that particular connection. Whenever it receives a *get* or a *put* request, it will execute the operation on the storage and send the answer to the client. Scan queries are simply put on a queue (scan queue).

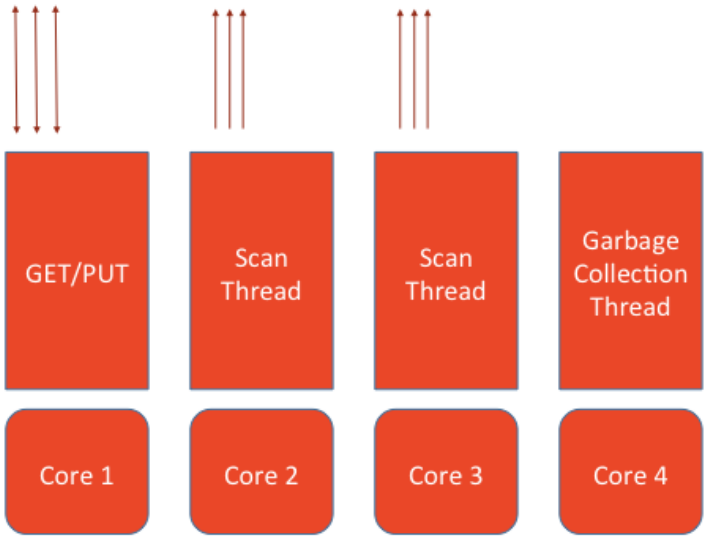



Figure 4.1: General Overview of the key value store architecture

One of the scan threads has the role of the scan coordinator. This scan thread polls the scan queue. If this queue is non-empty it batches all scan queries up to a configurable threshold into one query and notifies all the other scan threads. Then all scan threads, including the coordinator, partition the storage into equal parts and scan their parts.

This threading model allows for independent provisioning of resources for *get/put* and scans. The threading model for scans is illustrated in fig. 4.2.

### 4.1.9 Operation Push Down

To speed up analytical queries, our scan interface allows defining selections, projections, and simple aggregations. The scan threads will execute batches of scan queries during each cycle. We require all selections to be in the conjunctive

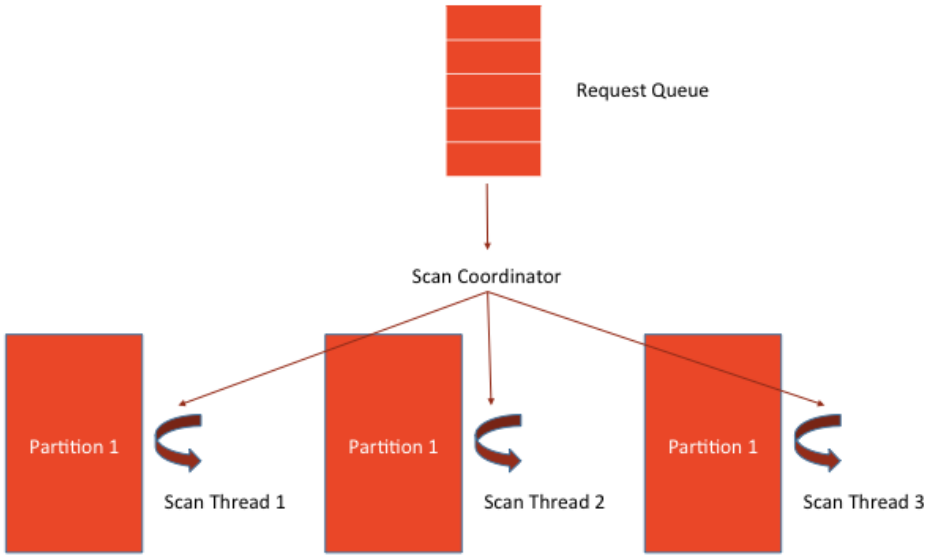



Figure 4.2: Shared Scans on multiple partitions

normal form (CNF). All the queries are then compiled into one big query plan (we were inspired by this [GAK12] work). To speed up the scan, we generate a function with LLVM that evaluates all the queries. Compiling queries into LLVM is not a new idea and is described for example in [Neu11] and [Klo+14]. To our knowledge, however, this was not done before for shared scans. LLVM gives us some optimizations for free. For example, if two queries share a predicate $x < 4$ we generate code that executes this predicate twice. LLVM will remove this duplicate for us. The generated LLVM code produces a byte matrix, where each row corresponds to a tuple in the page and each column to a conjunct. This matrix is initialized with all values set to $0$. Whenever a predicate evaluates to true, it will set the corresponding entry in the byte matrix to $1$. When we are done, we test for each query on each row whether all its conjuncts are set to $1$. For each query and each tuple where this is the case we materialize the result and write it into a local buffer. This buffer will be sent to the client as soon as it is full (or the scan finished) and a new buffer will be acquired. The scan will be discussed in detail in section 4.6.

## 4.2 Design Space

There are a lot of ways how one can build a key value store with a scan. We visualized the most important decisions in fig. 4.3. The red edges mark the approaches we implemented and benchmarked. In the following part of the section

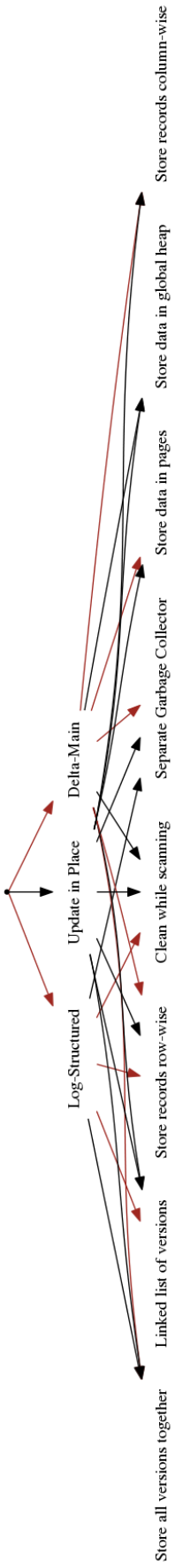



Figure 4.3: Decision-Tree

we highlight the most important design decisions and discuss their implications. For most design decisions there is a potential trade off between *get/put* and scan performance.

### 4.2.1 Where do we put Updates?

This is one of the most important design questions. The three most popular answers to this question are: update in place, delta-main, and log-structured. We sketch all three approaches shortly.

The most obvious way to store data is to update wherever it is stored in memory. This has the main benefit that updates do not fragment the data, if the update does not change the size of the tuple too much. This advantage will diminish drastically in a system that supports versioning: every update will increase the size of the tuple, since the tuple cannot be rewritten but a new version needs to added. Therefore, update in place is good for scans but bad for *get/put*. The next problem of update in place is that it is difficult to implement it without using locks: updating a tuple can't be made with a simple atomic operation. Current hardware only support atomic operations on a few bytes - typically 16. Traditionally, one would take a lock on the page, when the tuple is being updated.

Organizing data in a log was introduced in [RO92] as a way to implement a write-optimized file system. This idea is also used by key value stores (e.g. [Ous+11]). A variation sometimes used are log-structured merge-trees [ONe+96] (for example LevelDB and RocksDB). Using a log brings several benefits: a log can sustain high update loads as new data just needs to be appended to the log and, as shown in [RKO14], a log imposes low memory overhead. This all makes it very attractive to use for a key value store. The drawback of a log is that scanning a log is not straightforward. One cannot simply iterate through the log, as invalid records don't get invalidated or deleted. By just looking at a record it is usually not clear whether an entry is still valid and, as a result, the main index needs to be consulted. The next potential problem is that if the system experiences a high write load, there might be a lot of garbage that needs to be scanned. As a consequence the memory controller might become a bottleneck. Therefore, log-structured storage good for *get/put* but bad for scans.

Instead of updating data in place, one can write into a write-optimized data structure and merge it from time to time into a read-optimized one. We call this approach *delta-main*. The good thing about this is that it allows to store data very densely and still provide high write throughput. This idea was presented in [Sto+05]. SAP HANA [Fär+12] is a commercial implementation that uses the

delta-main approach.

### 4.2.2 Record Layout

This is probably the hardest question to answer. We can either store each tuple as one block of data or we can break it down into fields and store several fields of different tuples continuously in blocks. The second approach, a columnar data-layout or column-store, has proven to work well for analytical workloads. The first, a row-store, is very popular for storage systems that specialize for OLTP workloads. Also, most key value stores store their data row-wise. The intuitive conclusion is that a column layout hurts *get/put* performance and is good for scans while a row layout favors *get/put* requests at the cost of scan performance.

Column versus row layout has been studies extensively before (e.g. [Hal+06], [AMH08], [ABH09], [Sto+05], [BMK99], and [Bon+06]).

The main advantage of a columnar layout is scan speed. If a query asks for only a few columns, we do not need to scan all data but only the columns the query is interested in. Furthermore, we can also process data more efficiently, e.g. by using vectorization (described in [ZR02] and [Wil+09]).

The main advantage of a row-based layout is that *get* and *put* requests are simple to process. While we need to collect every column from a different memory location in a columnar layout, we just have to copy one memory block if the row is stored. Some previous work (e.g. [Ail+01]) proposes to store data column-wise only within a page. This improves *get* performance while it still preserves the advantages of column-stores.

### 4.2.3 How should we pack Tuples together?

This is an important question, as the answer will highly influence the locality of the data. The denser data is packed together, the faster the scan will be while the *put* request will potentially become slower as it needs to make sure that the data stays compact. Therefore, a dense data layout is good for scans but bad for *get/put*. In an extreme case, one would just put all data into a single block of memory. While this makes scanning very easy, maintaining such a memory layout is virtually impossible. The other extreme is to just use the global heap to allocate space for each tuple. But this will hurt scans, put requests, and garbage collection as well, scans because it will need to scan the hash table and do pointer chasing, put because each request needs to do an allocation on the global heap, and garbage collection because it needs to free a lot of small memory blocks. Furthermore, such a system would not behave predictably because neither malloc

nor free execute in constant time. The traditional compromise (see [GR93]) is to pack data into fixed-sized memory blocks usually called pages. With paging a scan operation can load whole blocks of data into cache instead of single tuples and therefore can improve scan times drastically. Furthermore, it is much easier to write a fast allocator/deallocator for fixed size pages. Paging is usually done to trade memory for disk accesses [GP87]. Our main concern, however, is how we can trade *get/put* performance for scan speed. The bigger the page, the higher the scan performance, but the more work needs to be done to manage a single page during a *put* and in the garbage collector.

### 4.2.4 How do we handle versions?

As described in section 4.1.7, we want to support versioning of tuples. This means that we need to potentially store several versions per key. There are mostly two ways to do that: store them at different locations and link them together or try to keep all versions for one key in one memory block. Building a linked list of versions has the advantage that writing a new version is much cheaper, as we never need to move the old versions to another place in storage. As a drawback, we will need to iterate through this linked-list for each *get* (and potentially *put*) request in order to find the version we are interested in.

### 4.2.5 When should we do Garbage Collection?

As we support versioning, having some form of garbage collection is inevitable. Versions are only readable as long as there are transactions which don't have a newer version in its read-set. Here we can chose between two strategies: either we run a dedicated garbage collector from time to time, or we collect garbage whenever we execute a scan. If the scan threads have to do garbage collection, scan time will increase. But this strategy has still a strong advantage: cleaned data is cheaper to scan (since scanning garbage is not free) and doing garbage collection while scanning makes sure that tables are scanned often are garbage collected often as well. But even more importantly: a dedicated garbage collection thread will fight for resources with the scan threads.

### 4.2.6 Conclusions

There are more decisions to make than the ones we listed in this section, but we tried to focus on the fundamental ones. It is not the case that we can combine any strategy with any other. For example, it does not make much sense to design a log-structured column store. Furthermore, there are some dichotomies between

particular design decisions and the features we want to implement. Most importantly, update in place is very hard to implement in a lock-free manner. Therefore we did not implement a storage that uses this strategy. Storing the tuples in the global heap will not allow to get a predictable response time for scans and probably not even for *get/put* requests.

We therefore decided to implement three combinations of the strategies outlined above. The first is log-structured storage. There we link versions together, since packing them together would mean that we would have to copy old versions to the head of the log on each update. Since *get/put* operations do not profit at all from a cleaned log, we run the garbage collection at scan time. This ensures tables that are scanned often are also cleaned more often while tables are never scanned will only be cleaned when we start to run out of memory. The second and third implementations both use a delta-main strategy to process updates. For delta-main, garbage collection is more expensive as we need to rewrite a lot of pages under high write load. We use log-structured storage to store deltas. The main difference between the second and third implementation is the record-layout: the second implementation stores tuples row-wise, the third uses a columnar representation. We call these three implementations ”TellStore log”, ”TellStore row”, and ”TellStore column”.

## 4.3 Log Structured

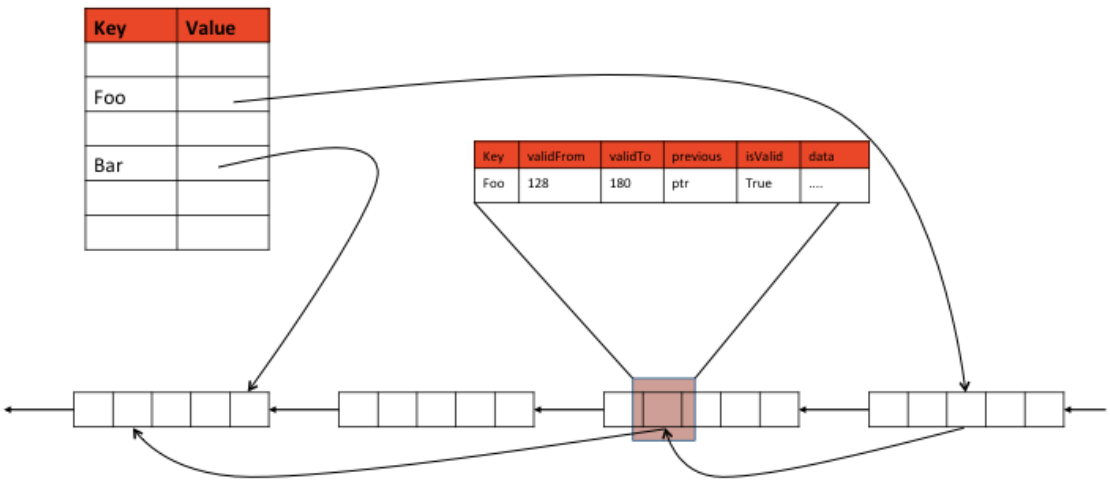



Figure 4.4: Log-Structured storage

Our first approach was to take a fast key value store and add a scan feature to it. Tell 1.0 used RAMCloud which proves to be very efficient for get and put operations. However, as shown in table 4.1, RAMCloud does not perform well for scan queries. The basic idea of the log-structured storage is to have a hash table serving as the primary index where the value of the hash table points to an entry inside the log. For each update and delete operation, a new entry is appended to the log and the hash index is updated. Simply replacing an existing entry in the log is not an option because, by definition, the contents of a log is immutable. The hash table serves as point of synchronization: concurrent updates append an entry to the log and try to atomically update the hash table. Appending to the log will never fail unless we run out of memory. The atomic update on the hash table, however, will fail if another update on the same key executed first. In this case the storage will report a conflict error to the client.

There are two obvious ways to scan through a log-structured storage: scan through the index or scan through the log directly. The second way of scanning looks favorable at first since it does not introduce a random memory lookup for each tuple. However, implementing such a scan is not as straightforward as it may seem: as a new entry is appended to the log for each update or delete, only log entries that are referred by a pointer in the hash table are valid. As a consequence, a log scan needs to perform a hash table lookup for each log entry to verify that the entry is still valid. If we assume that the hash table is larger than only a few megabytes (which is the case for all the workloads in our experiments), it does not fit into cache. Therefore, hash table lookups are essentially also random memory lookups which means that we cannot efficiently scan the log unless we change its structure.

A log normally supports only two operations: read and append. We could get rid of this restriction and allow for write operations. However, this would make replication and durability either very expensive or virtually impossible as log-shipping cannot be achieved in a consistent manner if the log segments keeps changing. Nevertheless, we can soften this restriction, if we make sure that changes in the log do not need to be replicated but can be recovered by replaying the log.

In our implementation every entry in the log corresponds to one version of a tuple. The versions are linked together. Each version entry stores its key, the tuple itself, and two version numbers. The first version number, *valid-from* is equal to the version of the transaction that inserted the tuple, the second version, *valid-to*, is initially set to infinity, but is adapted once a new version of this tuple is appended.

### 4.3.1 Log-Append

Appending to the log can be done atomically. The log maintains three pointers: a *head*, a *tail* and a *sealed-pointer*. A call to append will atomically increase the head pointer by a given number of bytes and return the old pointer to the caller. This region is now private to the caller: this means that the calling thread can write to it. After the caller is done writing to the log, it will set a bit at the beginning of the entry that marks the entry as either valid or invalid (this allows to abort an append if it runs into a conflict - as described later in section 4.3.3). Then it will mark the entry as sealed by setting another byte in the meta-data section of the entry. At this point the entry is read-only. Finally, we try to update the sealed-pointer by incrementing it over log-entries until either we reached the head of the log or a non-sealed log-entry. The sealed-pointer is used for scans: without it a scan might read a log segment while another thread is writing it. Our implementation guarantees that all segments before the sealed-pointer have been successfully written.

### 4.3.2 Get

Get operations are straight-forward: we first check the hash table and if the key exists there, we follow the pointer to the corresponding entry. If the *valid-from* field of the entry is in the read set of our transaction, we know we found the tuple and can read its data. Otherwise, we have to follow its previous pointer (potentially recursively) until we find a version where *valid-from* is in our read-set. If we encounter previous pointer set to null, this indicates that there is no readable tuple for this key and we can stop and report an error to the client.

### 4.3.3 Put

For a put request, we again first try to find an existing tuple in the hash table. The hash table is our point of synchronization therefore we remember the pointer, if it already exists. We then check whether the newest version is in the read-set of the caller - if it is not we can return and report a conflict to the client. Otherwise we append a new entry to the log with the updated version. The previous pointer is set to $null$ if this put does not update an existing entry, otherwise we store a pointer to the previous entry. The *valid-to* field is set to infinity. We finally try to replace the old value in the hash table or insert a new one if it does not exist anymore. The result of this compare and swap will be reported to the client, as failing this last step means that we got a conflict. If this put updates an existing tuple, we write the version of the caller into the *valid-to* meta data field of the previous log entry.

After that the appended block is set to valid/invalid (depending on the result) and then sealed. While setting the *valid-to* field of the previous log-entry is not atomic, it is nonetheless correct: while put is running, we know that the new entry will not be readable by any other transaction.

### 4.3.4 Scan and Garbage Collection

Our log-structured storage will do garbage collection whenever a scan gets executed. One drawback of this technique is that a table might never get scanned and as a result garbage collection never gets executed on that table. To prevent this from happening, we use a timeout that will trigger a scan execution on tables that did not get scanned for a long time. While this works it might not be optimal: a table that does not get scanned does not need to be garbage collected as long as there is enough memory. One could definitely come up with better heuristics how to call garbage collection in these cases.

Each page contains a 4-byte integer at the beginning storing how many bytes within the page are garbage. For scanning we walk through the log from head to tail page by page. If the garbage counter is set to a value bigger than some user-configurable threshold (by default half the size of a page), we scan the page in collection mode: each scan process has a reference to a non-empty page and fills up this page with valid entries from the page we are collecting and scanning. For each entry we copy into the new page, we check whether *valid-to* is set to infinity. If not, we look up the key in the hash table and follow the linked-list of updates until we reach an entry that has a pointer to our current entry. Such a pointer has to exist, otherwise the entry we are copying would not exist. This pointer is then set to the new address in the new page. As soon as the page is full, it will be appended in one operation to the log. For each valid entry we also need to execute the scan operation: we simply evaluate all scan predicates from all queries against the entry and send it to all clients that are interested in the result.

One drawback of this technique is that it takes two full iterations until any garbage is collected. The alternative would be to scan each page twice. This would, however, slow down the scan.

## 4.4 Row Store

The main idea behind our delta-main implementation is to keep the main mostly read-only while writing changes into a delta. Each table holds four data structures: a list of pages hold the data in the main, a hash index for the main, a log that

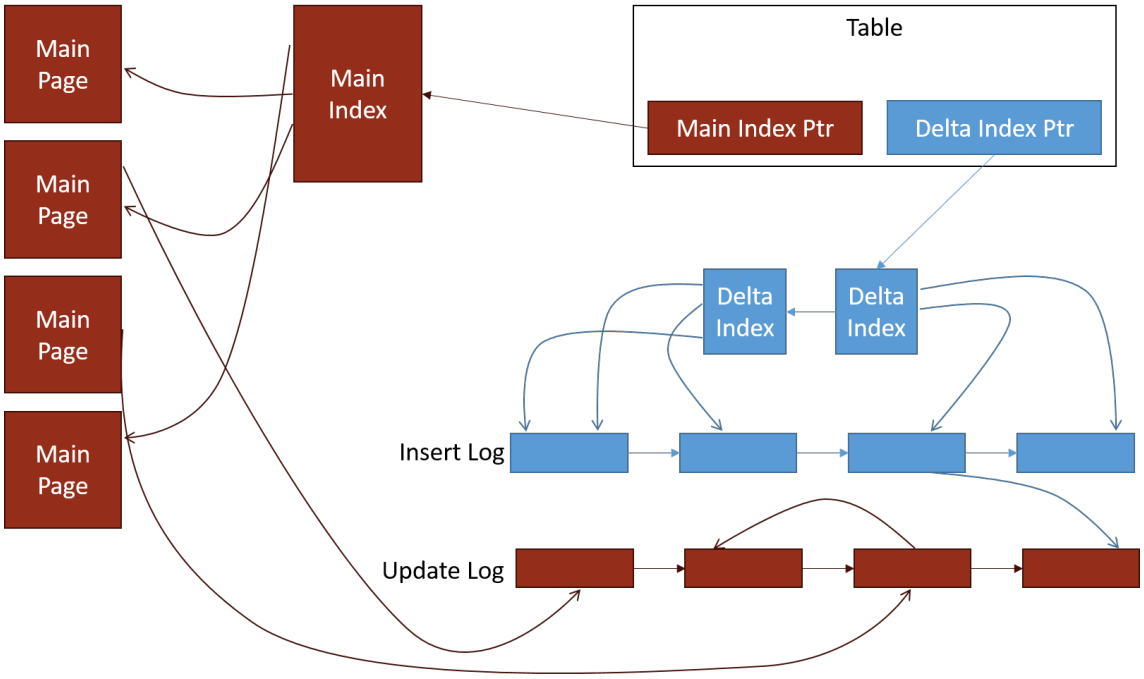



Figure 4.5: Data structures used in delta-main

stores the delta, and a list of hash tables index the delta - the overall architecture is visualized in fig. 4.5.

The data within the main is compactly organized as a collection of fixed-sized pages. The delta consists of two logs and a linked list of hash tables. The first, called *insert log* is used to store newly arrived inserts while the second log stores updates and deletes. Entries in the main keep pointers to the newest version of a record in the update log if such a record exists. So do entries in the insert log. Update log entries, on the other hand, keep backward-pointers to the previous version in the update log. However, in order to prevent cycles, there are no such *previous-pointers* back into the main or insert log. This design facilitates the construction of the delta index because it is sufficient to index the insert log. Update log entries are always reachable by following the *newest-pointers* in the main or delta index.

### 4.4.1 Main Index

We use cuckoo hashing [PR04] to index the main. Cuckoo hashing has the benefit of guaranteeing $O(1)$ lookup time. As the main is only read or constructed our cuckoo hash table implementation does not need to be thread safe. The garbage collection thread will not modify the main hash table but rewrite it.

It is important to realize that a record key always appears either in the main or in the insert log, but never in both. If two concurrent updates hit the same key, they

are both redirected to the corresponding entry, either in a main page or in the insert log which then serves as the point of synchronization. This reduces contention as conflicts can only occur at record and not at page level.

The drawback of cuckoo hashing is that each look-up can involve several cache misses. The reason for this is, that the data is stored in several tables (usually two or three) and in a worst case each table has to be checked. The cuckoo hash index does not perform as well as the large single-table the log structured storage uses. We suspect that a more efficient variant to cuckoo hashing exist but we did not explore this area as the cuckoo hashing implementation we used proved to be fast enough for our benchmarks.

#### Insert Log Index

Unlike the main index, the data structure that indexes the insert log needs to allow for concurrent write access because several threads could try to insert records at the same time. This is why we use a simple open addressing hash table with linear probing and atomic insertion operations. As resizing a hash table is an expensive operation, we never resize, but instead create a new hash table whenever the current one is full. After creation, we link it to its predecessor and finally atomically register it as the new *delta index pointer* of the table. This hash table is simple, as it does not need to support deletions. The garbage collector will simply append a new empty hash table when it starts and remove the old hash tables atomically when garbage collection is done.

#### Get

A get operation first performs a key lookup in the main index and then, if it cannot find the key, in the delta. If the key does not exist in both of them, it reports this to the client. The only difference between a block in the main table and one in the delta is that the one in the delta only contains one version, while the one in the main might contain several versions (the delta itself can, of course, contain several versions which will be linked together. But each log-segment within the delta will contain exactly one version of a tuple). Both of them might point to a newer version in the delta. It then first checks whether the newest version in the block is readable. If it is, it needs to follow the newest pointer if the pointer is not null, otherwise it searches for the newest readable version within that block or reports an error to the client if it is reading from the insert log. This works because all versions reachable from the newest pointer are at most as old as the newest version in the main entry. If it needs to follow the *newest-pointer*, it reads the entry in the delta it is pointing to - the newest version in the storage. If this version is in

the callers read set, it reads it, otherwise it follows the *previous-pointer*. It does this recursively until it either finds a readable entry, reaches a version that is also contained in the main entry, or until it reaches the end of this linked list. At this point, we know where the newest readable version is and return it to the client.

#### Put

Put operations are slightly more involved as they also have to deal with concurrency. Like a get operation, a put operation first checks for the existence of the key in the main and then, if necessary, in the delta index. Then we first check for conflict. A conflict occurs if the newest version of a tuple is not in the readers read set. The *newest-pointer* (be it in a main or in a insert log entry) serves as a point of synchronization, we remember its value before appending an initially invalid entry in the update log. Once this entry is created, we compare and swap its address with the remembered value. If we fail, we can simply abort the operation and report a conflict. This means that a concurrent update succeeded and we know for sure that the version of a concurrently running transaction is not in the read-set of the caller. If the compare and swap succeeds, we can set the *valid-bit* to true and return. In case of an insert, we also have to check whether the *delta index pointer* has changed in the meantime and if so, add a reference to our log record to this new delta index hash map.

#### Scan

Before a scan starts the coordination thread retrieves the list of all pages in the main and the current head and tail of the insert log. No inserts done after a scan starts are readable for any client that issued a scan request. The main pages and the pages from the insert log are then distributed to all scan threads - this helps us to nearly linearly speed up scans. The scan threads will then simply iterate through all tuples stored in the pages. For each entry they check the *newest-pointer* (i.e. it is not null). If it is set they also need to follow this pointer and check the referenced updates. For each tuple we then check all predicates from all queries. Then we check the version against the read set in the matching queries. If all matches, we copy the result into a result page. As soon as a result page is full, it is sent to the client.

#### Garbage Collection

A natural question is when and how to merge the data in the delta (update and insert log) into the main. In our case, we do this as part of the garbage collection which is also in charge of reclaiming space of no-longer used and invalid tuple

data. The strategy used in our case is *delta-main rewrite* which means that instead of updating main entries in place, we rewrite them completely in each garbage collection step. Obviously, this makes garbage collection an expensive operation. On the other hand, it allows packing tuple versions tightly together which speeds up get, put, and scan operations.

At the start of a garbage collection pass, the collector retrieves the head pointers of the logs, and the *page list* which is simply a array of all main pages. It then rewrites main pages lazily: It first checks whether a main page needs any modifications at all and if not, leaves it unchanged. It creates new main pages by merging existing records with their never versions in the update log and removes old tuple versions no longer used. A new *main index* is created as we proceed. This index is divided into several pages as well, and only affected pages are rewritten.

To prevent losing updates, we change the *newest-pointer* of each tuple in a garbage collected page. This pointer will show up in the main table but not in the update log. Therefore the new point of synchronization for concurrent updates will be inside the new main page. We use the same strategy to collect the insert log. During a garbage collection phase, we therefore introduce a new level of indirection for *get* and *put* operations: if such an operation sees the *newest-pointer* set to another main page, it will follow this pointer and continue from there the same way as described above.

### 4.4.2 Row-Layout

As the name already says, the row storage organizes tuples row-wise. We organize the data within two levels of granularity: each page contains a list of multi-version records, each multi-version record contains all versions of a tuple with a given key plus its key. The multi-version record's data layout is illustrated in fig. 4.6. The row-layout was highly influenced by PostgreSQL.

The multi-version record stores its size, the key of the tuple(s), an array of versions, an array of offsets, and the tuples themselves. The size field contains a compound number: the size in bytes of the multi-version record itself and the number of versions it stores. The first number is used by the scan so that it can skip the whole multi-value record if it can decide by looking at the key that no query is interested in the tuple. The newest-pointer is the only field in the multi-version record that is not read-only by *get/put* threads. It points, as described previously, into the delta, or is $NULL$ if the newest version is in the main. The version-array stores the versions of the tuples: this is the version of the transaction that wrote the tuple. These versions are ordered in descending order. This allows to easily

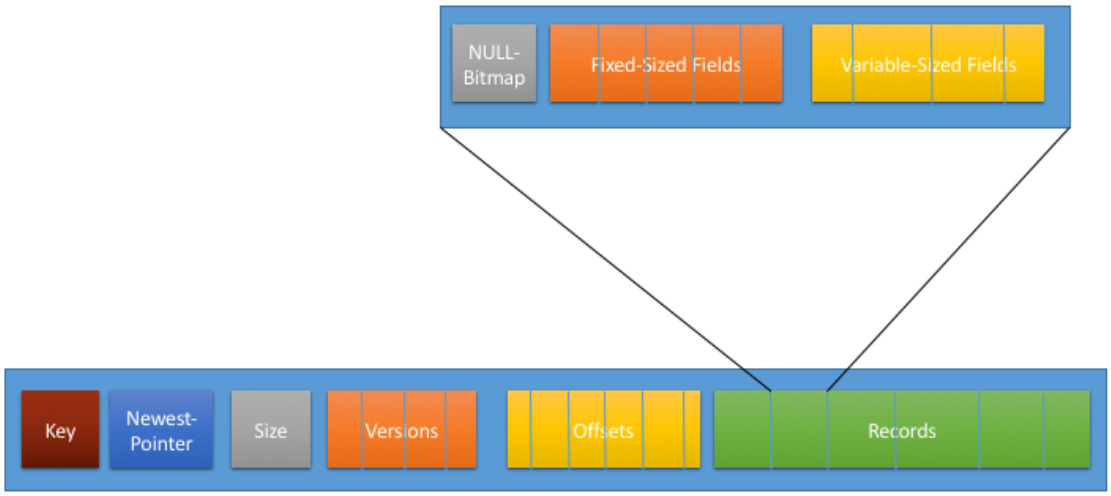



Figure 4.6: Organization of data within a multi-version record

calculate the *valid-from* and *valid-to* values of each tuple. *valid-from* is equal to the version at this index position, *valid-to* is equal to the previous entry or infinity if it is the first entry in this array and the *newest-pointer* is $NULL$. Note that the previous version might not be in the multi-version record but is reachable via the newest-pointer. The offset array stores offsets from the beginning of the multi-version record to the begin of the tuple (in bytes). Thanks to this data layout we can get a specific version of a tuple and its size without scanning through the data of all versions.

The layout of the tuple itself is quite straight-forward: we first store a bitmap which contains one bit per column. This bit is set to $1$ if the value is $NULL$, $0$ otherwise. If all columns have a *NOT NULL* integrity constraint, this bitmap is omitted. After that, we store all fixed-sized fields sorted by size, so that we only lose minimal space to the alignment. The last part is the variable-sized fields: each field is first a 4-byte integer which contains the size of the field followed by a byte array.

## 4.5 Column Map Layout

*ColumnMap* is a data structure first described in [Bra+15]. The idea, following the *Partition Attributes Across Paradigm* [Ail+01], is to first group records into contiguous chunks of memory (pages) and within such a chunk organize them in such a way that values corresponding to the same column are grouped together

which is also often referred to as *c*olumn major order in the literature. Columnar designs favor analytical scans because such scans are typically interested in only a small fraction of all columns, so only these have to be processed. Moreover, using vector instructions several values belonging to the same column can be processed at once. Typically, this good scan performance comes at the cost of decreased *get* performance as a tuple has to be gathered from several different locations in the page. In order to fetch a tuple, one must know the page offset of the first tuple value and then compute the offsets of the remaining values from the length (record count) and width (data type size) of each column.

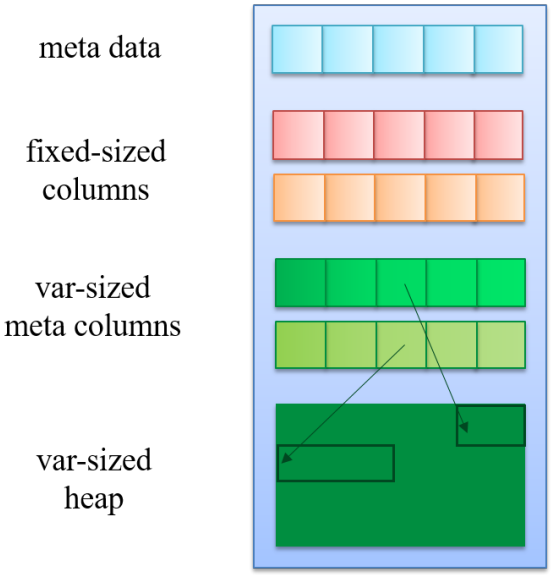



Figure 4.7: Column Map Page Layout

However, if values can have arbitrary sizes (as for example var chars), this simple computation falls apart and get/put operations become horribly slow. This is why state-of-the-art systems avoid variable-sized values, either by simply disallowing them (as in [Bra+15]), allocating them on a global heap and storing pointers (as in MonetDB [Bon+06] and HyPer [Neu11]), or using a dictionary and store fixed-sized dictionary code words (as e.g. in SAP/HANA [Fär12] and DB2/BLU [Ram+13]). For our design, global allocation does not work because it makes lock-free garbage collection virtually impossible and dictionary encoding would slow down *get* operations dramatically. Therefore, we settled for a novel columnar storage layout that improves ColumnMap as shown in fig. 4.7. In addition to variable-sized values, our *Column Map Page Layout* also supports versioning.

### 4.5.1 Column Map Page Layout

Each page starts with a field that stores the *tuple count*. Next, there is a column that stores key-version-newest triples. Versions belonging to the same key are grouped together in descending key order and *newest* points to an update log entry if newer versions of that tuple exist. Thus scanning this first column is sufficient to check whether a page needs garbage collection. A page does not contain garbage if one of the following is true:

- Each key within this triple is unique.
- The oldest version of each non-unique key within the triple is still active. This means that the oldest in-flight transaction is newer than this version.

The next column keeps the serialized length for each tuple. These numbers could be computed from other meta information, but are very handy for get or scan operations because they provide a shortcut to find out how much space tuples will need in a result buffer.

These meta data columns are followed by the fixed sized columns and the var-sized meta columns which, for every value on the var-sized heap, store a (4-byte) page offset to point to that value as well as its (4-byte) prefix. This has two advantages: first, by computing the difference of two consequent offsets, we can also compute the size of a heap value, and second, we get a speedup on prefix scan queries because we can identify a set of candidate tuples without having to look at the heap.

Last, but not least, there is a heap for variable-sized values. We make sure that entries belonging to the same tuple are sequentially stored which speeds up get operations because delivering the var-sized fields is basically a lookup to the tuple size and the first meta column, followed by a memory copy. With this design, column map offers good scan performance while still providing reasonably fast get operations.

### 4.5.2 Materialization

According to common wisdom, the main drawback of a columnar data layout is that the materialization of single record is expensive. The reason is that the offset to each column of the record needs to be calculated and fetched. For every copied field there will be up to one cache miss.

As an illustration, fig. 4.8 shows pseudo-code for a naive implementation of the materialize function. As soon as a record gets located within a page, it needs to

```
void materialize(char* result,
                 const char* page,
                 int tupleCount,
                 int idx)
{
    // make sure offset points to first column within page
    int offset = pageMetadataSize;
    for (auto& field : schema.fields()) {
        // copy field from this column to result buffer
        memcpy(result, page + fields.size() * idx, fields.size());
        // calculate offset to begin of next column
        offset += field.size() * tupleCount;
    }
}
```

Figure 4.8: Naive materialization function

be copied into a buffer in row-layout. Copying the tuple is done by a materialize function. The materialize function will simply loop over all columns within the table and copy them into a buffer. The result will then contain the tuple in row-format.

This function is, as explained before, expensive. The first obvious cost is the iteration over the fields within the schema - but we can optimize this by inlining the function. The next problem is the memcopy: if this data has to be read from memory, we have to wait here for hundreds of CPU cycles. This will potentially happen for every field. It will be faster if the CPU loads the whole page into a cache - but it will still cost around 10 cycles if it can be found in L2 and more than 70 cycles if it has to be fetched from L3. A last problem with the function is the calculation of the next offset: this offset is recalculated within the loop, making it difficult for the CPU to recompute.

Luckily, we can get rid of these problems with a very simple, yet effective trick. We know the schema already at compile time and we write the materialize-function as in fig. 4.9. Of course, we want to be able to create and modify table schemas at runtime, but the expectation is that this does not happen very often. Therefore we can afford to pay 100 milliseconds to recompile this function. Whenever a table is created or its schema changes, we take the schema and generate LLVM-code for the materialize function and register it on the table. Therefore each table will have its own materialize function which will simply execute a number of memory copy operations.

This will speed up the materialization of tuples drastically. On first sight, one might think that we did nothing more than loop unrolling. But this is only part of the story. Since we got rid of the loop and we compiled the schema information into the materialize function, the CPU can perfectly pipeline the offset computations

```
void materialize(char* result,
                 const char* page,
                 int tupleCount,
                 int idx)
{
    // pageMetadataSize is known at compile time
    memcpy(result, page + pageMetadataSize + 8*idx, 8);
    memcpy(result, page + pageMetadataSize + 8*tupleCount
                                           + 8*idx, 8);
    memcpy(result, page + pageMetadataSize + 8*tupleCount
                                           + 8*tupleCount
                                           + 4*idx, 4);
    memcpy(result, page + pageMetadataSize + 8*tupleCount
                                           + 8*tupleCount
                                           + 4*tupleCount
                                           + 2*idx, 2);
}
```

Figure 4.9: Compiled materialization function - Example with a schema with 4 columns

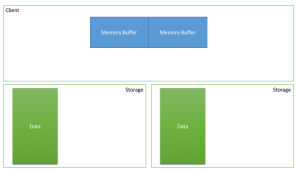
(there are no dependencies between offsets). Furthermore, the memory prefetcher will request all fields from memory at once. For the row based layout, we will to pay hundreds of CPU cycles to get the row as well, with the columnar layout, we will need to pay for the cache miss but we will wait for all cache misses only once in best case. This materialize function can therefore make efficient use of single core parallelism.

## 4.6 High-Performance Scans

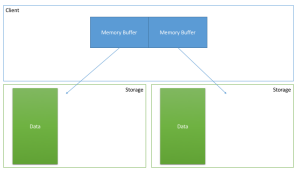
### 4.6.1 Using one-sided RDMA to write the results

As stated in section 4.1.9, we are compiling batches of scan queries into one big query to LLVM. This becomes even more efficient when using a columnar layout as we only need to scan columns of interest. Furthermore, LLVM helps us vectorizing the code to use even more parallelism.

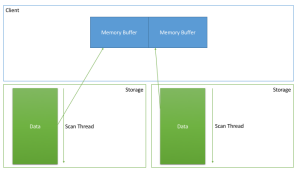
The scan within TellStore uses one-sided RDMA to ship the data and two-sided RDMA to initiate and finish the scan. The way the scan works is illustrated in fig. 4.10. Before the client initiates a scan, it needs to allocate a large enough block of memory to hold the result (fig. 4.10a). It will then send the query to all storage nodes (fig. 4.10b). Each storage will then execute the scan independently: they will first enqueue the query and as soon as a new scan phase start, it will be dequeued and all waiting queries will be compiled to LLVM (described below). During the scan, the scan threads will evaluate all selection predicates and for



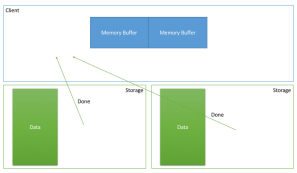

(a) First, the client allocates a buffer of memory for the result




(b) The client sends the query to all storage instances




(c) While scanning, the storage nodes write the result directly into the main memory of the client




(d) When done, the storage nodes sends an event to the client

Figure 4.10: Overview how the scan works - viewed from the client side

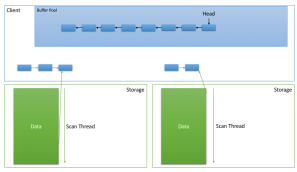
each matching tuple, they will materialize the tuple (potentially only partly, if the query contains a projection) and write it, using one-sided RDMA, right into the main memory of the corresponding client machine (fig. 4.10c). As soon as the scan finishes, the storage node will send a notification message using two-sided RDMA to all clients, informing them that the result of their scan is in their memory.

Using one-sided RDMA to deliver the query result to the client has the big advantage that the client CPU is not used at all during the scan phase. While a scan is running, a client can therefore do other work. Alternatively, it can also periodically poll its memory, to check whether new data was written by a storage node and start computation on the result.

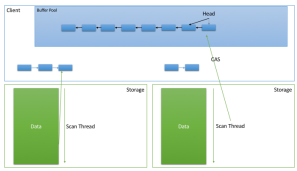
The main drawback of the current implementation is that the client needs to allocate a big enough block to hold the whole result within one block of memory before it starts the scan. This, however, is often very difficult or even impossible in practice. Therefore the clients have to over provision the size of the scan memory which limits the number of concurrent scan requests they can issue, as the main memory becomes a limiting factor very quickly.

This, however, is not a problem of our design but a mere limitation of the implementation. Instead of having big single blocks of main memory the storage writes the result to, a better solution would be to do the memory management within the storage.

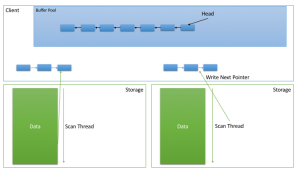
A sketch for how to solve this problem is presented in fig. 4.11. The algorithm for doing memory management on the client is quite simple and makes use of the



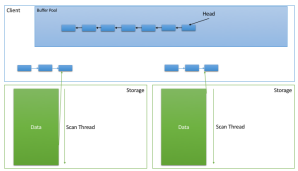
(a) One storage reaches the end of the current memory block



(b) It gets a new block from the pool using RDMA



(c) It appends the new block to its private linked list



(d) It continues writing results into the new appended block

Figure 4.11: Remote memory management

fact that Infiniband cards can execute atomic operations within remote memory. Since these operations are passive (they do not involve the client CPU), a slow client won't slow down the execution of the scan - except for a slow NIC. Instead of allocating a big block of main memory for each storage node it sends a scan request to, the client would allocate a large memory pool of small blocks of memory (probably a few megabytes each). It would register all blocks at the Infiniband card. Out of these blocks, the client would build a large, singly linked list.

Each storage node would now build a singly linked list of memory blocks which will form the result. Whenever a memory block is full (fig. 4.11a), it will execute a compare and swap operation within the client pool to set the head of this linked list to the next block (fig. 4.11b). If this compare and swap operation fails, it will retry. On success, the old head is privately held by the current storage node. It will then append the new block to its private linked list (fig. 4.11c) and continue working (fig. 4.11d).

The client on the other hand, while consuming, could just remove consumed blocks from this linked list and append it to the linked list within the pool. It just needs to make sure that it does not remove the last block from the list until it gets notified by the server that the scan finished.

This server-side memory management would, of course, not solve all problems. It might still happen that the client runs out of memory. It does, however, take away from the user (or query engine) the duty to know an upper bound of memory for each scan used.

### 4.6.2 Query Compilation

Compiling queries to machine code is an important and popular way to speed up query execution. Within the storage nodes, we do that to speed up the scan times. TellStore, however, does not compile each query separately but it compiles all queries at once used for the shared scan. The query compilation works differently on each underlying data storage (row, log, or column map). I will only explain how query compilation works on TellStore using the column map. As the column map comes out as clear winner, the other data storages will get deprecated in the near future. Furthermore, compiling the query plan for the column map is the most complex one - the other just in time compilers just have less functionality.

Compiling query batches instead of single queries has one important draw-back: since the mix of queries arriving at the cluster will usually change for each scan phase, the compiled queries can not be cached. Instead, the query plan needs to be built, optimized and compiled on every run. This leads to a optimization trade-off which is often seen in systems that do some form of ad-hoc execution: if we spend too much time optimizing the query plan, the whole execution time will go down as the optimization might become a bottleneck, if we don't do enough optimizations, scans will be too slow. In this section I will first explain how the optimization and compilation works in general. Our first implementation used often more time doing query optimization than scanning. Then, I will explain how we dealt with this trade-off.

Whenever a scan phase starts, the main scan thread will fetch all waiting queries from the scan queue. Each query is written in the conjunctive normal form (this is a hard requirement). A TellStore query is defined as a triple $q_i =< \pi_i, \sigma_i, \alpha_i >$. $\pi_i$ is a set of columns - the projection of the query. The selection, $\sigma_i$, is required to be in the conjunctive normal form - therefore we can write:

$$\sigma_i = (p_{i,0,0} \vee p_{i,0,1} \vee ... \vee p_{i,0,l}) \wedge (p_{i,1,0} \vee p_{i,1,1} \vee ... \vee p_{i,1,m}) \wedge ... \wedge (p_{i,k,0} \vee p_{i,k,1} \vee ... \vee p_{i,k,n})$$

In this notation, $p_{i,j,k}$ is the $k$th predicate within the $j$th conjunct of the $i$th query. The last element of the triple, $\alpha_i$, is a set of pairs of an aggregation function identifier and a column. The currently supported aggregation functions are:

- sum
- min
- max

- count

In theory, support can be added for all linear aggregation functions. A function $f$ is linear iff $f(\lambda \cdot x + \mu \cdot y) = \lambda \cdot f(x) + \mu \cdot f(y)$. One important function that can not be pushed down is average. In order to calculate an average, the user needs to push down two aggregations (sum and count) and calculate the final average on the client side. The reason only linear functions are supported is that each storage node will do the requested aggregation only on the data it holds. Therefore the client will need to aggregate all results from all storage nodes in the end.

If $\alpha_i \neq \varnothing$, the storage node will assume that $\pi_i = \varnothing$ is true. Here, it is important to remember that the storage does not support grouping which means that an aggregation on one column and a projection on another is not defined. TellStore will always ignore the projection and ship back all aggregation results.

TellStore will generate two new functions on each scan phase: a selection function and a materialization function. The selection function takes a page as input and returns a matrix $R$. Within $R$, $r_{ij} = 1$ iff the selection of query $i$ evaluated to true on record $j$ within the page. The materialization function will then take the page and the matrix $R$ as input and materializes the result. It will, for each record within the page and each query, either write all requested fields via RDMA to the client (if $\alpha_i = \varnothing$) or it will update an internal query state (otherwise). This means that we have to iterate over each page twice, once to evaluate all predicates and then again to materialize the result. Due to the columnar layout of the page, this does not mean, however, that we always iterate over all data within a page.

Figure 4.12 shows pseudo-code of such a generated function. Of course it does not show all details: We do not really make two heap allocations for each call to that function, we don't generate C++ code - instead we generate an intermediate representation for LLVM, we do not show here how the version handling is done or the lookup in the delta... But it should illustrate the main idea. The function basically generates an array of booleans where each boolean represents a conjunct for one record. For each column that appears in one predicate, we generate a loop that iterates over the column and evaluates all the predicates for that specific column. It then sets the conjuncts of the predicates to true iff the predicate evaluated to true. In a last step, the result is generated in another loop that sets all entries to true for which all predicates evaluated to true.

LLVM can optimize this code surprisingly well. It will unroll the loops and add vectorization code for the predicate evaluation. The materialize function will not be explained in detail here - as this function is much less interesting.

```
// An example of a generated selection function for two queries:
// - column1 < 8 && column1 >= 0
// - (column1 = 4 || column4 == 7) && column1 > 0
bool** selection(const char* page) {
    // toSizeOffset is a constant known at compile time
    int numRecords = * ((const int*) page + toSizeOffset);
    // we have 4 conjuncts in total
    auto conjuncts = new bool[numRecords][4];
    // metadataOffset is a constant known at compile time
    const char* column1 = page + metadataOffset;
    for (int i = 0; i < numRecords; ++i) {
        conjuncts[i][0] ||= *((const int64_t*) column1 + i*8) < 8;
        conjuncts[i][1] ||= *((const int64_t*) column1 + i*8) >= 0;
        conjuncts[i][2] ||= *((const int64_t*) column1 + i*8) == 4;
        conjuncts[i][3] ||= *((const int64_t*) column1 + i*8) > 0;
    }
    // column1Size, column2Size, etc. are known at compile time
    const char* column4 = page + numRecords*column1Size
                               + numRecords*column2Size
                               + numRecords*column3Size;
    for (int i = 0; i < numRecords; ++i) {
        conjuncts[i][2] ||= *((const int32_t*) column4 + i*4) == 7;
    }
    // The last step is to and together all conjuncts
    auto result = new bool[numRecords][2];
    for (int i = 0; i < numRecords; ++i) {
        result[i][0] = conjuncts[i][0] && conjunct[i][1];
        result[i][1] = conjuncts[i][2] && conjunct[i][3];
    }
    return result;
}
```

Figure 4.12: Example of a generated selection function

### 4.6.3 Compile Time Optimizations

The columnar version of TellStore ran pretty early into the problem that compile time overhead was dominating run time. Compilation could easily take twice as long as the actual scan - even for large tables. Compilation would take several hundred milliseconds while the scan itself would finish in less than 100 milliseconds. To make matters even worse, the compilation time grew linearly with the number of queries provided. We implemented several counter-measures to speed up compilation and optimization time.

The first, and most important, optimization was to do the vectorization manually. So instead of producing simple code and let LLVM figure out where vectorization can be introduced, our code generator unrolls the loops and adds LLVM vectorization instructions. LLVM will then emit vectorization instructions for the underlying CPU. This brought down LLVM compilation time drastically.

The LLVM compilation is done by the main scan thread - the other scan threads are not doing anything while this step is executed. Parallelizing compilation is a hard problem and it is not supported by LLVM. Instead we compile the materialization and the selection function in parallel. This simple optimization reduces compilation time by another $20 - 30\%$

While profiling LLVM, we saw that it spends around $40\%$ of its time in the *Loop Strength Reduction Pass*. The following loop illustrates how this algorithm works:

```
for (int i = 0; i < j; ++i) {
    auto value = array[i];
    // do something with value here
}
```

A non-optimizing compiler would generate assembly code that would roughly translate to this:

```
int i = 0;
if (i >= j) goto END;
do {
    // This multiplication here is only done in assembly as machine
    // code is not type aware - but this would be wrong c++ as the
    // compiler would introduce another multiplication here
    auto value = array + i * sizeof(*array);
    // do something with value
    ++i;
} while (i < j);
END: // continue here
```

This code needs to execute a multiplication for each operation and introduces

dependencies between `array` and `i` which is bad for pipelining and branch prediction. LLVM will therefore rewrite this loop like this:

```
int i = 0;
if (i >= j) goto END;
auto iter = array;
do {
    auto value = *iter;
    // do something with value
    ++i;
    ++iter;
} while (i < j);
END: // continue here
```

This means that loop strength reduction just eliminates a multiplication. Since we generate quite a lot of loops that look like the above example, LLVM will execute this loop strength reduction pass very often - slowing down compilation time quite drastically. But as soon as one is aware of this problem, it is very easy to fix: we just emit already optimized code, bringing down LLVM optimization time by nearly $40\%$!

We did a few more minor optimization, but the ones outlined here were the most beneficial ones with respect to compilation time. With these optimizations, compilation time went down to less than $10ms$. But more importantly, the compile time now increases very slowly. We never observed compilation times higher than $20ms$ - even for large query batches.

## 4.7 Durability

TellStore is not durable. While we believe that implementing durability within TellStore is simply an engineering effort, we recognize that this is probably the most important of the lacking features.

There are a lot of ways, how durability can be implemented. The simplest one is by just mapping all in-memory pages into main memory. However, this approach does not guarantee, that committed data does not get lost in case of machine failures as the operating system will decide when to write back dirty pages.

The following subsections sketch a simple algorithm to safely write back committed data to disk for each storage approach.

### 4.7.1 Log Structured

Writing a log to disk by itself is trivial: whenever a segment is written into memory, it is appended to a file on disk. This has to happen before the write operation is acknowledged to the client.

TellDB batches several write requests into large multi-write requests. Therefore, we probably will have to pay for the additional write latency only once per transaction. Furthermore, all write operations are send to all storage machines in parallel - which means that a lot of write requests will be executed in parallel on several disks.

The biggest difficulty for this approach is that garbage collection currently happens within the log as we softened the traditional append-only design of a log storage. [RKO14] describes a similar scheme of a log where in-memory garbage collection happens without appending cleaned segments to the end of the log to prevent write amplification.

The basic idea of this paper is simple: Instead of using one garbage collector a separate garbage collector will periodically clean the disk log. This garbage collector will write clean segments to the head of the log allowing to free memory on disk. As the disk is usually much larger than main memory and because scan threads only process the main memory log, disk-based garbage collection executions are much rarer than in-memory execution.

### 4.7.2 Row Store and Column Map

Making the other storage engines durable is even easier to implement. The insert and update log are append only and write operation are just synchronously written to disk as described above in section 4.7.1. There is a single log for all operations from all tables, instead of a log-file for each in-memory log.

In order to truncate the log occasionally the main pages need to be written back regularly. This process is usually called snapshoting. The main of our delta-main approach is mostly read-only and the writable part does not need to be made durable, as this information can be recovered by replaying the log. This makes snapshoting conceptually simple.

Each page will have an additional flag which indicates whether the page is dirty. Newly allocated pages are always dirty as they are not written to disk. Writing a snapshot can be done in a few simple steps:

1. Mark the current head of the log on disk. This will be the new head after the snapshot was written.

2. Iterate through all pages and write those marked as dirty back to disk. A concurrent garbage collection process will not interfere with this process.

3. Truncate the log on disk.

To do recovery, the latest snapshot of main is first copied into main memory and all newest-pointers are nulled. Then the log is replayed - which will generate new a new insert and a new update log for each table.

# 5

# Experimental Results

## 5.1 Introduction

In order to compare our three KV stores with each other we designed a synthetic benchmark and ran a number of experiments.

We chose Kudu as a baseline because Kudu is the only KV store known to us that has the same goals as TellStore.

The first goal of the experiments was to examine how well the TellStore variants perform as a simple KV store with a get/put load. We expect a row-based model to win but we also want to know how well the columnar storage performs on this kind of workload.

To measure get/put performance we present three experiments: one measuring get-only performance, one measuring insert-only performance and one measuring get/put performance. We don't expect the columnar storage to perform much worse for write-only workloads than the row storage variants. Write operations are written into a log in a row-format on all approaches. For the get-only and get/put workload, we chose to compare to other popular KV stores as well. This will prove that TellStore is a competitive KV store - even if used without its scan feature.

We do, however, expect to see differences for get requests. The columnar storage has to materialize a tuple from several columns scattered over a page in memory for each get request that reads from the main part of the table.

Furthermore we want to quantify the effect of batching of network requests. For that we configured TellDB with different sizes of request buffers to batch fewer

or more requests into a single batch. We expect to see some speedup due to batching.

The main feature that sets TellStore apart from other KV stores is the scan. We want to know:

- Which approach provides the lowest response time for scans? We expect the columnar storage to win.
- How does TellStore compare to Kudu - another KV store that claims to provide fast scans?
- How much do concurrent get and put requests interfere with scans and scans with get and put requests? We hope that the effects are minimal but some interference is expected as scans will put a high load on the shared memory controller of the CPU cores.

This chapter is structured into seven parts. First, the machine and software configuration is described. The next section specifies the workload we use for the benchmarks. The last five sections present the results of the experiments.

## 5.2 Configurations and Methodology

We ran all our experiments on a small cluster of 12 machines. Each machine is equipped with two quad core Intel Xeon E5-2609 2.4 GHz processors, 128 GB DDR3-RAM and a 256 GB Samsung Pro SSD. Each machine has two NUMA units with one NUMA processor and half of the main memory. Furthermore the machines are equipped with a 10 Gb Ethernet adapter and a Mellanox Connect X-3 Infiniband card, installed at NUMA region 0.

The KV stores we benchmarked in this section are all NUMA-unaware which has implications on performance. In order to have consistent numbers, we decided to run every process on only one NUMA unit. Therefore, throughput this section, the term *instance* refers to a NUMA unit which can use half of a machine's resources. *Storage* instances always run on NUMA region 0 such that they have fast access to the Infiniband card, while *processing* instances can run on both regions.

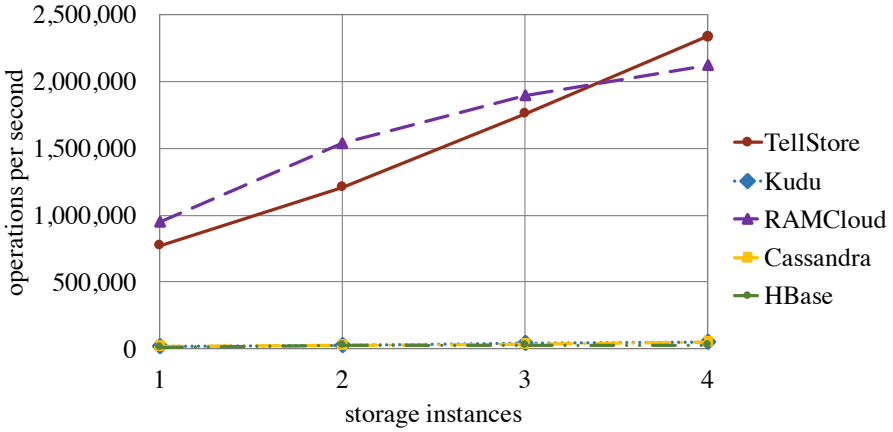
The system under test for all experiments is a KV store. Whenever we compare TellStore to other KV stores, we use the *ColumnMap* implementation, referred to as TellStore-Col(umnar). If we compare different TellStore implementations to each other, we also use Kudu as a baseline. We use Kudu because it is the only KV store that can provide a reasonable scan performance (see table 4.1).

To make sure that the load generation does not become a bottleneck, our experiment setting always uses three times as many processing instances as storage instances. For all our measurements, we first populated the data and than ran the experiment for seven minutes. We ignored the numbers from the first and the last minute to factor out the warm-up and cool-down time. If a query ran for more than seven minutes, as was the case for some of the experiments in table 4.1, we waited until it finished.

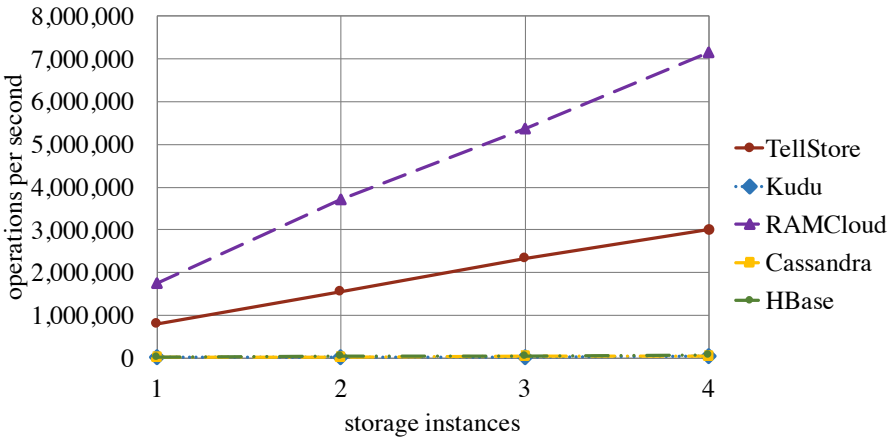
We did a considerable effort to benchmark all KV stores at their best configuration. In order to achieve a fair comparison to the in-memory systems, we put Cassandra's and HBase' data on a RAM disk. For Kudu, we used the SSD and configured its block cache such that it would use all the available memory. According to the Kudu developers, this is the preferred setting for Kudu because, after the warm-up time, all data should reside in memory. The benchmarks for HBase and Cassandra were implemented in Java using the corresponding client libraries [16a; 16b]. For *RAMCloud* and *Kudu*, we implemented the benchmarks in C++ using their native libraries [16c; 16d]. We used *multi-put* and *multi-get* operations in RAMCloud whenever possible and projection and selection push-down in Kudu. TellStore was benchmarked with TellDB, a shared library that incorporates the native TellStore client library and allows to execute *get*, *put*, and *scan* requests in a transactional context as well as manage secondary indexes. TellDB will be described in more detail in the next few chapters. For TellDB and Kudu, we batched several *get/put* request into one single transaction[1]. For TellStore, a batch size of 200 proved to be useful, while a good batch size for Kudu sessions is 50. We turned off replication for all KV stores, except for HBase where this is not possible which is why we used three storage instances (HDFS data nodes) instead of only one for that particular case.

The main result of our experiments is that TellStore is a very competitive KV store because it can deliver a high throughput for *get* and *put* requests. It also provides an order of magnitude higher scan performance than all other KV stores we tested against, both in terms of latency and throughput. Furthermore, TellStore's scan performance does not significantly deteriorate when a moderately-sized *get/put* workload of 35,000 events per second is executed in parallel. It is not surprising that the TellStore variant with the columnar layout provides the lowest scan latencies. However, somehow counter-intuitively, it is also able to sustain a very high *get/put* load which hence makes it the preferred implementation for

[1] Kudu actually uses the notion of a *session* which has weaker transactional semantics than a ACID transaction in TellDB, but is still a useful concept.




(a) get/put workload




(b) get-only workload

Figure 5.1: different KV stores, scaling from 1 to 4 storages instances with a transactional workload

mixed workloads.

## 5.3 YCSB#

### 5.3.1 Benchmark description

The Yahoo! Cloud Serving Benchmark (YCSB) [Coo+10] is a popular benchmark for KV stores and cloud service providers. YCSB, however, does not include selection, projection, or aggregation queries. This is why we slightly extended the benchmark in order to test these additional features. This new benchmark, called YCSB#, makes the following modifications: First, it changes the schema so that it includes variable-sized columns and second, it introduces three new queries that inspect big portions of the data.

The new schema consists of an 8-byte key, named $P$, and tuples that consist of

eight fixed-sized values, named $A$ to $H$. We used the following data types: 2-byte, 4-byte, and 8-byte integers as well as 8-byte double-precision float. Each of these types appears twice. The two variable-sized fields, $I$ and $J$, are short strings of a variable length between 12 and 16 characters. The three queries are:

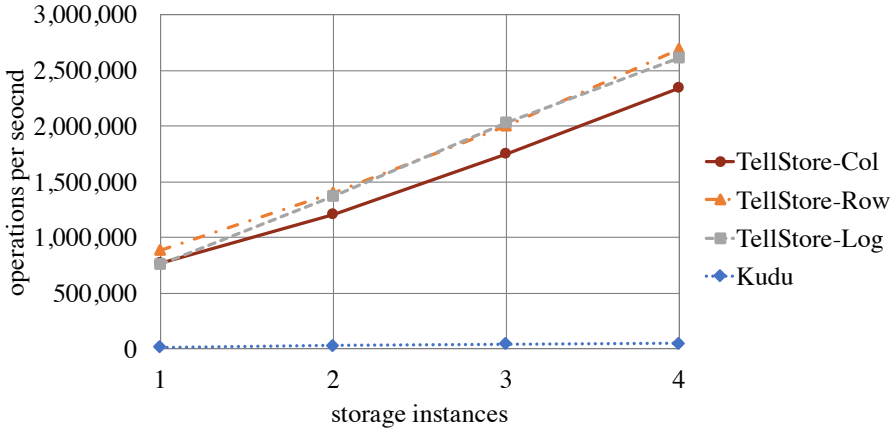
- *Query 1* a simple aggregation on the first floating point column to calculate the maximum value:
  ```
  SELECT max(B) FROM main_table
  ```
- *Query 2* does the same aggregation as query 1, but additionally selects only values in the second floating point column which between $0$ and $0.5$ which is true for approximately half of the values:
  ```
  SELECT max(B) FROM main_table
  WHERE H > 0 and H < 0.5
  ```
- *Query 3* retrieves all records where the first 2-byte integer column is between $0$ and $26$ which results in retrieving approximately 10% of the entire dataset.
  ```
  SELECT * FROM main_table
  WHERE F > 0 and F < 26
  ```

The benchmark also defines a scaling factor that, similarly to TPCH [Cou16], dictates the number of tuples in the database. Throughout our experiments, we used a scaling factor $50$ which corresponds to a test set of 50 million tuples. We found this to be a useful size to compare all the considered KV stores, even the ones not particularly good in analytics, e.g. Cassandra.

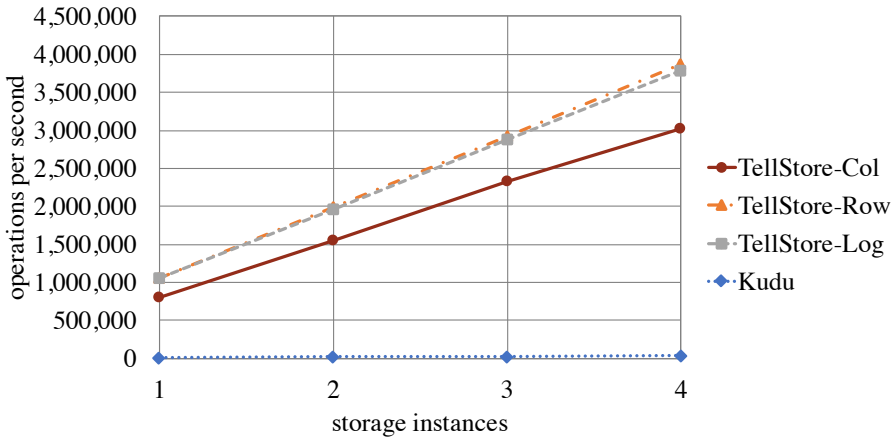
## 5.4 Get/Put workload

In the first experiment, we ran YCSB# in a setting very similar to the original YCSB that does not execute any scan queries. We tested two different configurations: *get/put* and *get-only*. While the *get-only* workload only consists of *get* requests, the *get/put* workload additionally has 50% *put* requests of which one third are inserts, one third updates, and one third deletes. This ensures that the size of the stored data stays constant which is not important for this experiment, but becomes essential as soon as we add scan queries whose latency is heavily influenced by the overall data size.

First, we measured the operation throughput (number of *gets* and *puts* per second) of TellStore-Col against all other considered KV stores as presented in fig. 5.1. It becomes immediately clear that TellStore is indeed very competitive for




(a) get/put Workload




(b) get-only workload

Figure 5.2: TellStore approaches and Kudu, scaling from 1 to 4 storages instances with a transactional workload

this typical KV store workload. In fact, only RAMCloud's performance is on par with TellStore. This can be attributed to two factors: RAMCloud and TellStore are both in-memory and can make efficient use of the Infiniband network. RAMCloud outperforms TellStore for get-only workloads. This is mostly due to the fact that TellStore reserves three CPU cores for scans and garbage collection (and only one for *get/put* requests), while RAMCloud can use all four cores to process operations. For the *get/put* configuration, however, TellStore and RAMCloud perform equally well.

Figure 5.2 compares the three TellStore approaches against each other and shows Kudu as a baseline. While the two row-based storages (TellStore-Log and TellStore-Row) outperform TellStore-Col, this difference is not as large as one might expect. As a *put* operation writes into a row-oriented log in all approaches, its performance is likely to be the same. For *get* operations, however, TellStore-Col

needs to assemble the records from different memory locations which comes at slightly bigger cost. This is also why the gap between TellStore-Col and the other approaches is slightly bigger in case of the *get-only* configuration. Compared to Kudu, this experiment confirms the results already shown in fig. 5.1: TellStore's throughput is nearly two orders of magnitude higher than the one of Kudu.

## 5.5 Insert-only

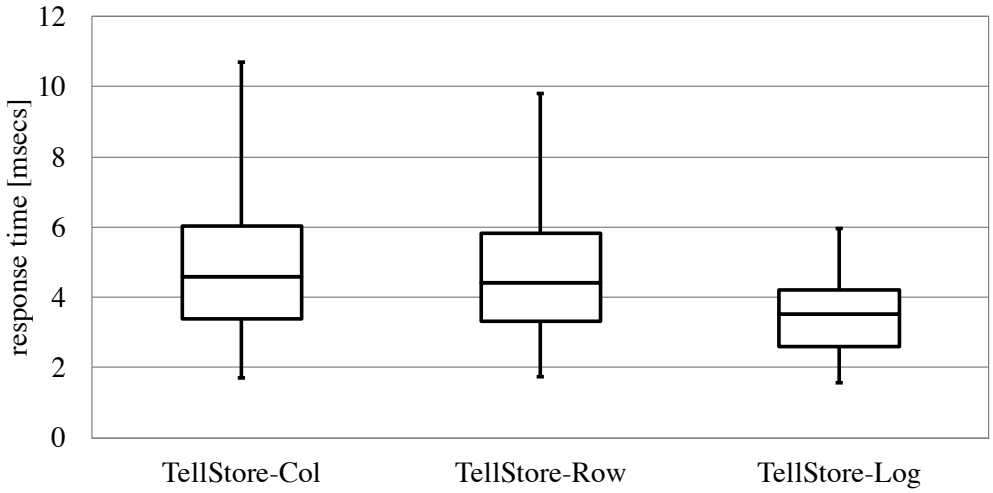



Figure 5.3: TellStore variants, transaction response time variation for 2 storage instances and insert-only workload

Our approaches not only differ in the way the data is organized but also in the hash table implementation. TellStore-Log wastes memory because it allocates a hash table big enough such that it never has to be re-sized. We cannot afford, however, to do that for the *delta* index of TellStore-Row and TellStore-Col as we recreate this index on each garbage collection step. Instead, we use a linked list of fixed-sized hash tables. Whenever an *insert* operation finds the current hash table to be full, it allocates a new one and links it to its predecessor. Allocation of a new hash table (even if small) is relatively costly compared to the other costs involved with an *insert*. This means that the latency of *insert* requests varies greatly between the ones that did and the ones that did not have to allocate a new hash table.

This effect is confirmed by the experiment shown in fig. 5.3 where we measured the response time of transactions (consisting of 200 *insert* operations each) in a setting with two storage instances. As one can see, the *delta-main* shows a bigger variation in the latency of these transactions. In fact, the presented box-plot shows

the 1, 5, 50, and 95, and 99 percentile and not the minimum and maximum. This is because the maximum value is so far apart that the rest of the plot would not be visible. A small number of transactions needed more than half a second to complete.

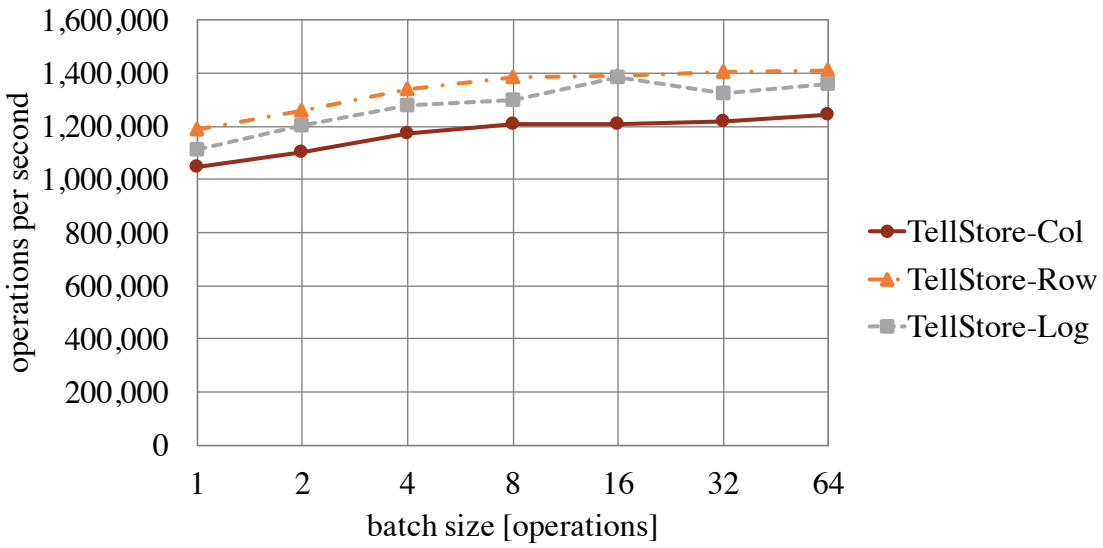
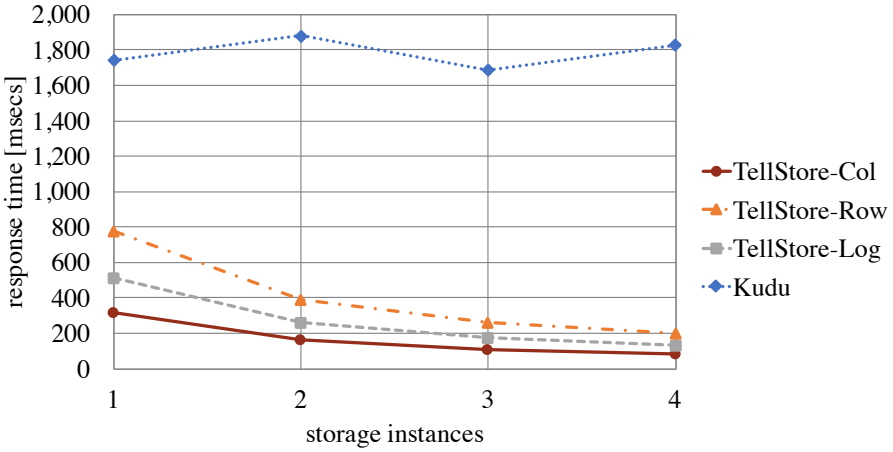



Figure 5.4: TellStore variants, throughput for varying Infinio batch size and 2 storage instances

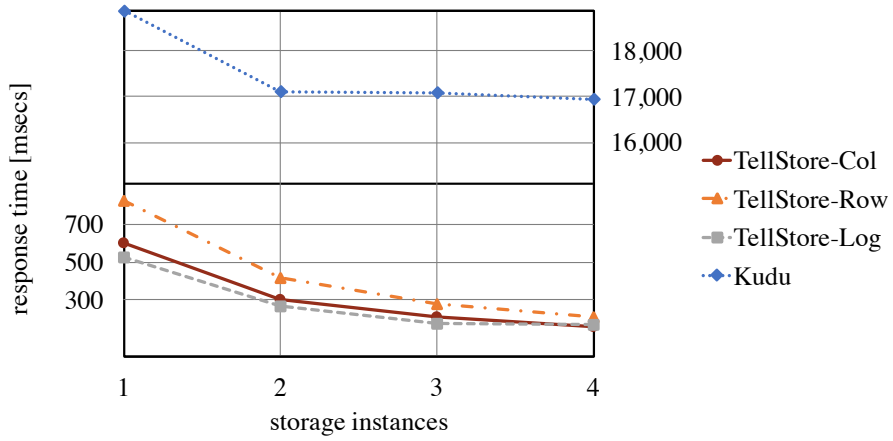
## 5.6 Batching

As will be explained in more detail in section 8.1.1, the processing layer of Tell 2.0 uses batching to get a significant throughput improvement at the cost of slightly increased latency. Whenever a transaction issues a request, this request gets buffered within Infinio and sent to TellStore whenever the buffer is full. While a transaction is waiting for a result, the Infinio runtime system schedules other transactions that can then help fill up the buffer.

Batching is important, as get/put messages are often very small and the message rate is limited on Infiniband. Therefore, in fig. 5.4, we show the throughput of the get/put workload for various batch sizes, using two storage instances. It is not surprising that all storage implementations benefit equally well from batching. Furthermore, we can see that a batch size of 16 seems to be a good default value.




(a) YCSB# Query 1




(b) YCSB# Query 3

Figure 5.5: TellStore variants and Kudu, response time of two different YCSB# queries running in isolation, scaling from 1 to 4 storages instances

## 5.7 Scans

In order to demonstrate the raw scan performance, we first ran the YCSB# queries in isolation (without concurrent *get/put* load). Figure 5.5 shows such results. We do not show the response time of query 2 as it is nearly identical to query 1.

As one would expect, TellStore-Col has the lowest response time for query 1 and query 2 (not shown). For query 3 however, the TellStore-Log is slightly faster. This is due to the fact that query 3 does not do projection and hence TellStore is required to fully materialize the records. This has the same consequence as already described in section 5.4, namely that TellStore-Col has to fetch values from different locations and therefore performs slightly worse.

The surprising result, however, is that the log-based implementation performs so well. The reason is that it does garbage collection while it scans which leaves it with one additional core (three instead of two) for scanning.

## 5.8 Mixed Workload

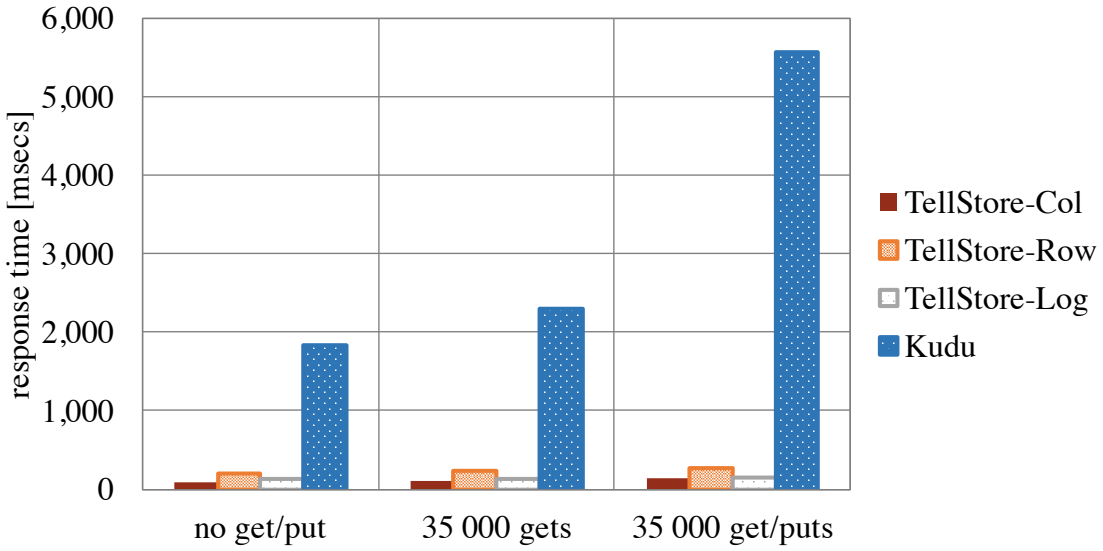



Figure 5.6: YCSB# Query 1 response time from TellStore variants and Kudu for different concurrent transactional workloads with 4 storage instances

To see how well a scan performs while the system also needs to execute *get* and *put* requests, we ran the YCSB# queries with the two different configurations shown in section 5.4 and compared these numbers to the results obtained in section 5.7. However, in order to make all systems comparable, we fixed the *get/put* request rate to 35,000 operations per second, 35,000 being the highest *get/put* load that Kudu could sustain. In fig. 5.6, we show the response time for

query 1 and four storage instances. For the other queries and with a different number of storage instances, the results look similar. Comparing the different scenarios, we can see no notable difference for TellStore. Kudu, on the other hand, does experience a significant increase in its query response time for concurrent *puts*. This clearly demonstrates TellStore's robustness under mixed workloads.

# II Query Processing with TellDB

# 6

# The Bd-Tree

## 6.1 Introduction

### 6.1.1 Problem Statement

Indexing is a very important feature to process OLTP workloads. The key value store only indexes an 8-byte wide primary key. Therefore, for some relations, not even the primary key can be stored as the key in the key value store. While queries could be answered by using whole table scans, this is often slower than using indexes (in OLTP, most queries do point lookups or small range lookups).

It is possible to implement secondary indexes in the storage layer. We did not do that for several reasons:

- Redistribution of key-value pairs would be more difficult, as each storage node would need to maintain secondary indexes for the tuples it holds. Migrating one tuple from one storage to another can be done in constant time in average, if there is only one hash table indexing the primary key. Migrating a tuple and its secondary indexes, however, is in $O(log(n))$, as each secondary index needs to be updated.

- For an index query, the client will have no knowledge about the location of the key value pair. Therefore, *all* storage machines will have to execute an index lookup. As the number of storage nodes increases, index lookups become more expensive. This will make the key value store less scalable.

- Index lookups on indexes that support range queries typically execute in $O(log(n))$. This means that a lookup gets more expensive the bigger the table grows. Range queries typically have a complexity of $O(k \cdot log(n))$, where $k$ is the size of the range. This means that the costs for operations on the storage layer become less predictable.

Elasticity, scalability, and predictability are some of the best properties of key-value stores. To retain these properties, we need to keep the key value store as simple as possible.

The alternative to implementing indexing in the key value store is to implement the indexing in the processing layer. The index still needs to be stored in the storage layer, but as far as the storage layer is concerned, the indexes are nothing more than a set of key-value pairs. This will get rid of the problems discussed above (if done efficiently):

### 6.1.2 Requirements

The first requirement we have for an index is that it has to be able to execute range queries. This means that we probably will need some for of tree-like index.

To be able to store an index in a key value store it has to be possible to break the index into several pieces and assign a unique key to each piece. This is usually possible for trees, but much harder for other structures like log-structured merge trees [ONe+96]. Furthermore, we want to minimize the number of network round trips. Therefore the number of get requests to storage for index lookups and the number of put requests for index modifications should be minimal.

As we want our system to be scalable, the index has to be able to execute concurrent operations executed on different processes. Latches are hard to implement in a scalable way. Therefore we want this index to be lock free. Lock-free algorithms can guarantee throughput as each successful CAS or LL/SC operation will result in global progress. Therefore, we are looking for a lock-free data structure.

### 6.1.3 Solution

This chapter presents the Bd-tree. Bd-tree is a new data structure designed to be used as a distributed index. Bd-tree is heavily influenced by the Bw-tree [LLS13]. Both data structures are lock-free and optimized for scalability.

In a first section, we justify the decision for a B-tree.

The next two sections of this chapter present a scheme for storing a B-tree in a key value store. In the next section, we explain how the Bd-tree is cached in

a correct way on storage. This caching scheme allows execution of most index lookups with a single get operation and most modifications with a single write operation as soon as the cache is warmed up.

In the following sections, we present the algorithms for the supported operations. We describe how split and merge operations can be done without the use of latches. Instead, we completely rely on the storage to provide some form of CAS or LL/SC.

The last section defines a contract which a storage has to fulfill to be usable with the Bd-tree. Furthermore, the invariants and the contracts of the Bd-tree and its operations are presented.

We finish the chapter by presenting a proof for liveliness of the data structure.

## 6.2 Which data structure should be used?

Indexing is done on client side only. While the index is stored in the storage, it is processed by TellDB. There are a lot of data structures which can be used to index data, and we invested a lot of time to find one that suits our requirements. Since we want to support range searches on indexes, we can not use unordered indexes likes hash tables.

Tell 1.0 implemented a hash table as well. It turned out that sacrificing the ordering of some indexes did not give us the desired speed-up and we did not port this index to Tell 2.0. Skip list variants are popular to index data, as they are relatively easy to implement.

But skip lists have a huge space overhead ($O(n \cdot log(n))$) which makes them unattractive for Tell. Having a small space overhead is important for several reasons: TellStore stores all data in main memory, therefore space overhead is expensive in term of cost and energy. Since the index needs to be processed in TellDB, the entries need to be shipped over the network which increases the network load. One network request has to be made per element within a range which increases the packet rate within the network.As discussed in section 2.4, network throughout and message rate are potential bottlenecks. The same arguments are true for balanced binary trees (like AVL tress or Red Black trees).

The data structure that suits our needs best is the B-tree. B-trees work well for indexes on disks. While Tell is an in-memory system, there is still a significant cost in fetching tree nodes. Instead of fetching them from a disk, we are fetching over a network. A big benefit over disk is, however, that it is much easier to have tree nodes of different sizes. This means that a half-empty tree node will only use up half as much memory as a full tree node. A B-tree stores arrays of elements within

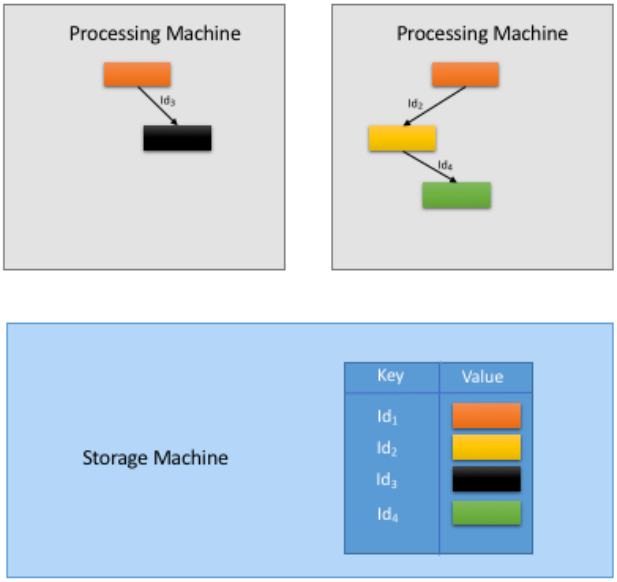



Figure 6.1: Storing the index in storage and process it in upper layer

each node, decreasing the memory overhead and the network load. For range queries, each node that gets fetched will contain several keys and not just one which will reduce the number of network requests made to the storage. To optimize range queries even further, we decided to implement a variant of a B+-tree. While B-trees store key value pairs within each node, a B+-tree stores the values only in its leaf nodes and the leaf nodes form a linked list. This simplifies range lookups, as one needs to just follow a linked list of leaf nodes after the smallest key within the range has been found.

## 6.3 Sharing the Bd-Tree between Processes

The Bd-Tree stores all data in two tables in the underlying key value store. It uses one table to store the tree nodes, called the *node table*, and one to store logical ids called the *pointer table*. The node table stores key-value pairs where the value is a serialized tree node and the key is a key identifier. It does not matter how this identifier is generated. In Tell it is generated by incrementing a global counter, but it could be some UUID or a random value as well. This scheme is illustrated in fig. 6.1. The pointer table is used as the central point of synchronization. Fetching a tree node from the storage requires two network requests. First, we look up the physical pointer in the pointer table. Then we get the serialized node from the node table. The whole process is explained in more detail in section 6.6.

The Bd-tree is organized like a linked-tree: each node, leaf nodes as well as

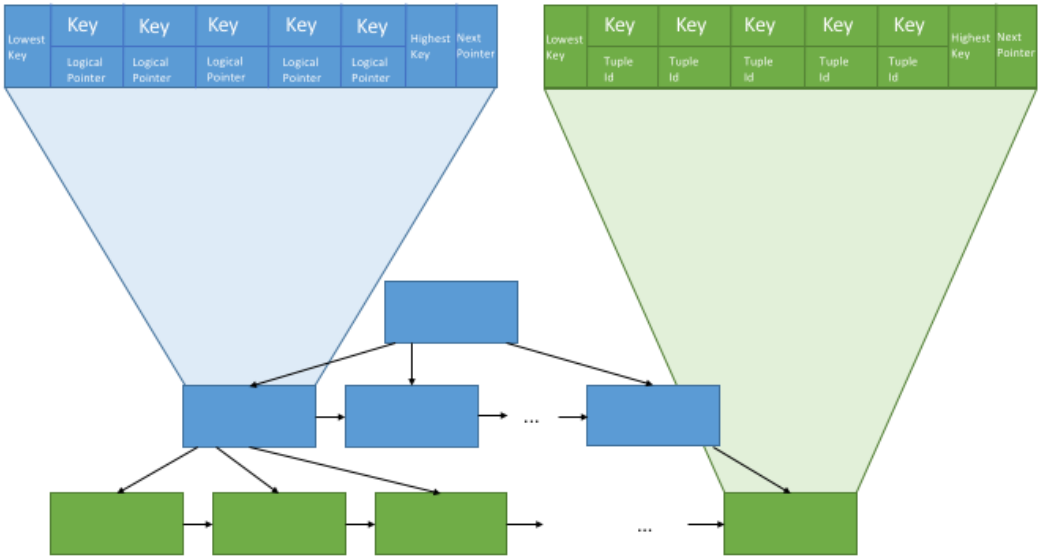



Figure 6.2: Structure of the nodes in the Bd-Tree

inner nodes, store the logical id of its right sibling. The tree, as well as the contents of nodes is illustrated in fig. 6.2. Each node holds a list of key-value pairs, sorted by key. The only difference between an inner node and a leaf node lies in the type of the value: A leaf stores a tuple id for each key, where the key in the tuple is equal to the one in the tree node. For inner nodes, the value is a logical id which points to a subtree where all keys are smaller or equal to the associated key. Each node also stores a lowest key and a highest key. These keys define the range of keys that are stored in the subtree with itself as the root.

This chapter described the Bd-tree in detail. Furthermore, chapter A lists C++-like pseudo-code of an actual implementation.

### 6.3.1 Concurrency

To allow for fast concurrent transaction execution, the B-tree needs to be synchronized, as we want several threads of execution, probably running on different machines, to be able to read and modify the B-tree concurrently. Reading is not very problematic: unlike a B-tree hold in local memory, looking up a tree node is not dangerous. For an in-memory B-tree, looking up a node that was deleted by another concurrent thread might result in a segmentation fault. One way to solve this problem would be by using hazard pointers ([Mic04]) or a form of epoch ([KL80]). If the nodes are stored in a key value store, this problem vanishes, as we will just get an error from the storage layer from which we can recover by rereading

the parent node.

Unfortunately, writing is a bigger challenge. Updating a single node can be done by using the compare and swap functionality of the underlying key value store, but this is not sufficient as we also need to be able to execute split and merge operations. A naïve approach would be to use CAS for simple node updates and a global lock for split and merge. However, this will be problematic as it would not allow for several threads to shrink and grow the tree concurrently. To make concurrency better, one can use locking at the node level instead of the tree level.

Our first B+-tree implementation was, therefore, a B-linked-tree, as described in [LY81]. This B-tree allows concurrent updates while holding at most three locks at a time. But to do that, one needs to have one mutex per tree node. In Tell, we can not simply use $std :: mutex$, as this would only lock the tree node on one machine. But our lock needs to be globally visible, i.e. other machines processing transactions must be aware of the locks and respect them (i.e. wait on them). To do that we implemented a locking service that did fair locking. But locking does not work well in distributed environments. With locks, every split or merge operation issues three additional network requests to the lock service, in the *best* case. The locking service also introduces new complexity to the system. With a locking service, we need to handle failures of the locking service and in case of contention the locking service might experience a high load. But the biggest problem with locks is that they increase latency as several additional network hops are needed to update the tree.

To get rid of these problems, we implemented a lock-free B+-tree variant within Tell. This B+-tree was highly inspired by the Bw-tree [LLS13], a lock-free Bw-tree published by Microsoft Research which is used within Hekaton [Dia+13]. We call this variant of the Bw-tree a Bd-tree (d stands for distributed).

## 6.4 Generating Ids

The two tables in which the Bd-tree is stored uses globally unique ids to identify pointers and tree nodes. To generate these identifiers we use simple counters. TellDB creates a global table which stores all counters. In TellStore and RAMCloud, tables are identified with a 64-bit unsigned integer. In the counter table, we create one entry per index table, where the key is equal to the corresponding table id. This entry is added when an index is created and initialized with $1$. Whenever a new id needs to be generated, we atomically increment the value for the corresponding index table which will give us a new, unique identifier (another 64-bit integer).

An atomic increment can either be supported by the storage engine as a small optimization (as it is the case for RAMCloud, but not for TellStore), or we can use the compare and swap operation to implement the increment operation within the processing layer.

In Tell, the identifiers are not recycled. This will become a problem as soon as an overflow happens. We argue, however, that this will never be the case, as 64 bit is sufficiently large: with a rate of one update per microsecond, it will take 600 thousand years until an overflow occurs.

## 6.5 Cache

B+-trees are very shallow. The number of records indexed by a B+-tree of order $b$ and height $h$ is bound by:

$$n_{max} = b^h - b^{h-1} \tag{6.1}$$

$$n_{min} = 2 \cdot (\left\lceil \frac{b}{2} \right\rceil^{h-1} - \left\lceil \frac{b}{2} \right\rceil^{h-2}) \tag{6.2}$$

In a B+-tree of order $b$, every inner node can have up to $b$ children and a leaf node can index up to $b-1$ records. Therefore, a completely full B+-tree indexes $b^{h-1} \cdot (b-1) = b^h - b^{h-1}$ records. Calculating the minimal number of records is only slightly more complicated: the root node of a B+-tree has at least $2$ children, all other inner nodes at least $\lceil \frac{b}{2} \rceil$. A leaf node indexes at least $\lceil \frac{b}{2} \rceil - 1$ records. The root node has two children which themselves are B+-trees of height $h-1$. Each of these trees indexes at least $(\lceil \frac{b}{2} \rceil - 1) \cdot \lceil \frac{b}{2} \rceil^{h-2}$ records.

In Tell, a B+-tree will typically have an order between $500$ and $2000$ (depending on the data type that is indexed). Therefore a tree of height four can index at least up to 63 billion records, one of height five more than 30 quadrillions. In practice, we can, therefore, assume that an index lookup needs at most 5 get operations to TellStore. $5$ get operations on TellStore can be executed in less than $100\mu s$. To put this number into perspective: $100\mu s$ is roughly the seek time of a solid state disk.

It still is beneficial to do some caching for the indexes. The obvious reason is that walking down the tree increases the packet rate and the load on the network. But there is another reason: fetching the tree nodes is a serial process. The identifier of a tree node is only known after its parent was processed. This makes message batching on a transaction level impossible.

Caching is difficult because a machine can not know whether a cached tree

node was updated by another machine since it was read from cache without asking the storage. But we can make use of an important observation: the higher up a node is in the tree, the less frequently it is updated and most updates only hit the leaf nodes. We make use of this property by only caching inner nodes. A tree node therefore only needs to be fetched if one of the following is true:

(a) The tree node is not cached.

(b) The tree node is a leaf node.

(c) It is detected that the cached version of the tree node is outdated.

Detecting whether a tree node is outdated works as follows: Each tree node stores its upper bound (the highest key which could potentially be stored within the current subtree) as a meta information. This upper bound changes whenever a split and a merge happened. Whenever we fetch a tree node from the storage layer, we check whether its upper bound is equal to the key we followed. If this is not the case, we know that the parent is outdated and we evict the parent from the cache and fetch it again. We continue recursively until we either fetched a node where the upper bound is equal to the parent key or until we reach the root node of the tree. Thanks to this caching mechanism, most index lookups only need to issue one *get* request to the key value store.

The cache is implemented as a hash table that indexes nodes. As a key for the cache we could either use the physical id (the id within the node table) or the logical id (the id within the pointer table). Caching by logical id is preferable for two reasons: (a) since the inner nodes store only logical ids, we would still need to look up the physical id in the pointer table and (b) logical ids typically live much longer than physical ids (the reason will be clear after we discuss the write operations).

Our cache is as four-way associative. It is implemented as a hash table with four entries per hash index - the size of the table can be configured by the user. Whenever we fetch a tree node by its logical pointer, we first check, whether the logical id is stored in the cache. If it is, we fetch it from there, if not we first fetch the physical id from the storage, then the tree node. We then store this tuple in the cache. If one of the four spots is empty, we put it there. Otherwise, we evict randomly one of the entries from the cache. This simple eviction strategy ensures that maintaining the cache is cheap. Updates are then executed in-place within the cache and then written back to the storage.

In the following, fetching from the storage always means that the cache is in between. Therefore fetching a tree node from the storage might not issue any network requests at all.

## 6.6 Search operation

Before the search starts, it initializes an empty stack where it stores the nodes it reads during the operation. This is needed in case of failure. A node could get deleted or updated, while the search operation is running. Instead of restarting at the root node, it will try to recursively reread at the lower levels until it has a proper state.

The root node has the logical id $0$. Therefore it will first fetch this node. It will then do a binary search on the key array to find the smallest key that is bigger than the one it is looking for. It will then follow the logical pointer associated with the found key and push the parent node onto the stack. It will do so until it finds the leaf node with the key it is looking for or it will fail due to concurrent write operations on the index. The operation can detect three types of failures:

1. The logical id it is looking for does not exist. This can happen if a concurrent operation merged the node it is looking for and deleted the node and its logical id from the storage.

2. The physical id it is looking for does not exist. This can happen during a concurrent merge as well. But in this case, the logical pointer did not yet get removed.

3. The key is not in the range of the subtree of the node the parent was pointing to. This happens if a concurrent split operation finished since we last read the parent node.

If one of these failures occur, it will pop the parent from the stack, evict it from the cache and restart from the grandparent. It will do so until it either reaches the root node or until the key is within the subtree again. Because of the caching, it might happen that several split and merge operations occur before it is detected on a certain machine. But this is fine since leaf nodes are never cached and we can, therefore, be sure to always read a valid state from the storage.

The number of network requests for a search operation is $2$ in the best case (reading the logical id from the lowest inner node followed by a *get* to get the leaf node) and $2 \cdot log_2(n)$ in the best case if the cache is cold. It might theoretically happen that the search operation starves if a split or merge operation or split operation finishes whenever a node gets read from the storage. But since merge and split operations typically happen rarely and since updates, in general, are slower than search operations; this is not an issue in practice.

## 6.7 Updates

On an update, we first need to issue a search to find the leaf node that needs to be updated. For insert and delete operations, we first check whether the node will need to be split or merged after applying the operation. If this is the case, the split/merge is executed *before* the update is applied. This is important since otherwise a leaf node might grow unbounded or shrink to an empty size, if a lot of concurrent transactions execute insert or delete operations within the same range.

Afterward, the update is applied to the leaf node locally (i.e. in the local memory of the processing node executing the transaction). The client then generates a new physical id and writes the new leaf with the new physical id to the node table in the storage. Afterward, a compare and swap operation is executed to change the value of the logical id to the new physical identifier. If this compare and swap operation succeeds, it will delete the old leaf node from the node table. Otherwise, it will delete the new leaf node and retry the update.

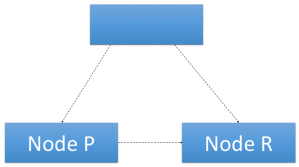
An update operation issues at least as many requests to the storage as a search operation plus three additional requests: An increment to get a new physical id, a *put* request to write the new leaf node, and one operation to update the logical pointer of the leaf node.

It might happen that an update process starves if several concurrent transactions execute updates on the same leaf node. This would be particularly bad, as it would create a lot of load to the storage, since writing the new leaf node will always succeed. In our experiments, however, we never observed this problem.

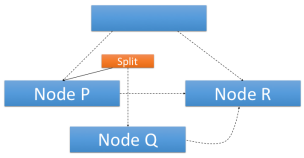
## 6.8 Split

If a node grows to a certain size, it needs to be split into two nodes. Unlike updates, splits can't be executed with a single compare and swap operation, as split needs to write two new nodes and then insert these two new nodes into the parent node. Potentially, this will result in another split of the parent node which might again lead to a split in its parent — cascading up to the root node in worst case. Only inserts will lead to a split of a leaf node and only splits will lead to a split of an inner node.

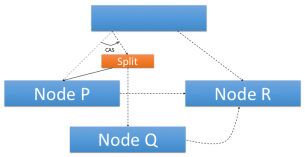
The split function is illustrated in fig. 6.3. Whenever a process decides that a node $P$ needs to be split, it will first create two new nodes: a split node $S$ and a new node $Q$. $S$ stores the physical id $P$ and the logical id of $Q$. $Q$ contains all key-value pairs from $P$ that are within its range, generated by using the middle entry from $P$ as a new lower key and $P$'s upper key for $Q$. Its next pointer is the physical id of the right sibling of $P$. This first step does not need any synchronization, as only



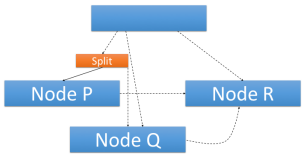

(a) Node P needs to be split



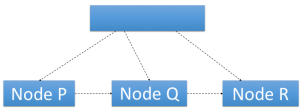

(b) Insert of new node




(c) CAS from parent to split node




(d) Update parent node



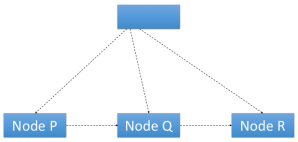

(e) Consolidate split node

Figure 6.3: Split. Dashed arrows represent logical id links, solid arrows represent physical id links.

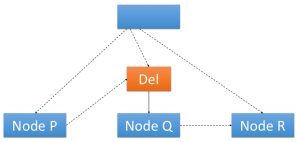
new nodes are written to the storage, and these are not visible to other threads (see fig. 6.3b).

In the next step, the split thread tries to update the logical id of node $P$ in the logical id table with a compare and swap operation. This compare and swap operation might fail for three reasons: (a) another thread updated a key-value pair within node $P$, (b) another thread removed a key value pair from $P$, or (c) another thread started a split operation. For (a), the split first updates $Q$ without synchronization, as it can simply rewrite $Q$, if necessary, and tries to execute the compare and swap again. If (b) happened, the split might not be needed anymore and the thread removes $S$ and $Q$ from storage and return. Whenever (c) happens, the thread deletes nodes $Q$ and $S$ and helps the other thread finish the split. Any combination of the above scenarios can happen during the creation of $Q$ and $S$. In that case the thread just removes $Q$ and $S$ and retries the operation it was doing before it started the split operation. If the compare and swap operation succeeds, all other processes might potentially observe the new split node $S$. If they do, they try to complete the split. This is important, as the process that started the split operation might have crashed during the split operation.

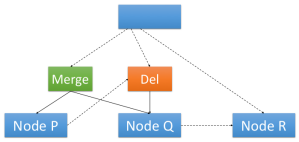
As shown in fig. 6.3d, the parent node now needs to be updated. Since the key within an inner node is always the lower bound of its corresponding child this pointer does not need to be updated. The only thing that needs to be done is an insert for the new child $Q$. This insert will let the parent grow and as a consequence, the parent node might need to be split as well. As soon as the parent node was updated, node $P$ needs to be consolidated. Consolidation in this context means that the split node can be deleted as it is not needed anymore. But simply removing



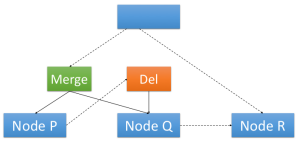

(a) Node Q is smaller than a given threshold



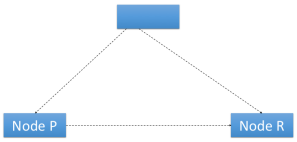

(b) Insert delete node




(c) Insert merge node




(d) Update parent node




(e) Consolidate split node

Figure 6.4: Merge. Dashed arrows represent logical id links, solid arrows represent physical id links.

the split node will create an invalid state - node $P$ still contains keys which are also stored in $Q$. Therefore, node $P$ is rewritten and the pointer from the parent to $S$ is atomically reset to the new node $P$. This last step is shown in fig. 6.3e.

#### Root split

A special case of the split is the root node split. If the root node needs to be split, we have a small problem: the logical id of the root node is fixed to the logical value $0$. Furthermore, there is no parent node to update. Therefore we change the split algorithm slightly for the root node. Instead of creating a split node and the sibling, we simply create two new nodes (which represent the truncated root node and its right sibling) and a new root node with two entries. The already consolidated old root node and the new right sibling get new logical identifiers. Then we execute a compare and swap operation to let the logical identifier of the root node point to the newly created node with two entries.

## 6.9 Merge

While splitting is done to the right, merging is done to the left. That means that whenever a process encounters a node that needs to be merged, it will merge it with its left sibling. Like split, merge is a multi-step process. The merge operation is illustrated in fig. 6.4.

A merge starts when a node becomes smaller than some predefined threshold. A merge is always executed *after* the corresponding delete operation succeeded. In a first step (fig. 6.4b), the process creates a deletion node and prepends it to

the node that should get merged. This is done by first creating a deletion node and then executing a compare and swap operation on its logical identifier. If this compare and swap operation fails, the process just deletes the deletion node and returns without retrying. The reasoning behind that is that the logical identifier now points to a different node which will be either bigger, because a concurrent update operation executed an insert, or another process started the merge process.

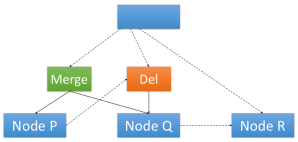
Next, the process will find the left sibling of the node to merge. This can be done by looking at the parent node (if the node is the left-most child of the parent we handle the case specially, as described below). It will then create a special merge node which will save the physical id of both nodes that should be merged (node $P$ and $Q$ in the example from fig. 6.4). It will then change the value of the logical id of the left sibling to the merge node with a compare and swap operation. This can fail for several reasons:

1. The node got updated after the last read. In this case, we will simply retry.
2. Another process already successfully executed this step. In this case, we will just continue with the next merge step.
3. Another process tries to merge the left sibling of the node we tried to merge and we, therefore, find a deletion node. Whenever this happens, the left merge needs to be completed first. This is a recursive process: we merge the left nodes until we can merge the current nodes.

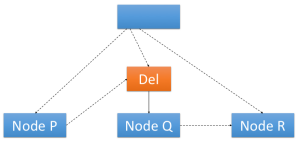
After that, the merge has mostly succeeded. Next, we update the parent node (still assuming that node $P$ and node $Q$ have the same parent). This is done by simply removing the key value pair for node $Q$. Since the key in the parent is equal to the lower bound of the corresponding node, the key for node $P$ does not need to be updated. Finally, a new node containing all key-value pairs from nodes $P$ and $Q$ is created and the parent is updated. These to last steps are done with one compare and swap operation which lowers the synchronization overhead.

#### Root node

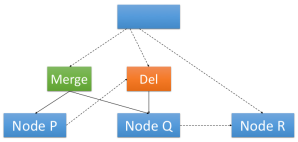
The root node is a special node within the Bd-tree. Within the tree, all nodes are allowed to shrink down to a given threshold. For the root this is not true anymore: a root node is allowed to be empty (if it is a leaf node) or to shrink down to have two children. If a root node has only one child, it gets deleted. This is a very simple process: first, a compare and swap is executed on the logical id so that the root id now points to its child. If this compare and swap succeeds, the old root node is deleted.



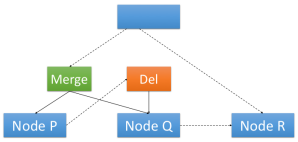

(a) Merging $P$ and $Q$ would result in a too large node




(b) A new split node is created




(c) The new split node is installed




(d) The split is executed

Figure 6.5: A merge with a concurrent split. Dashed lines represent logical ids, solid lines physical ids.

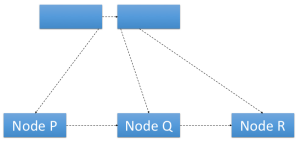
#### Special case: merged nodes are too big

Another corner case can come up when merging nodes: it can happen that a merge would produce a tree node that would be too large. Whenever this happens, a merge is immediately followed by a split operation. As an optimization and to make sure that no storage nodes bigger than allowed are written to the storage, the split is executed before the merge is consolidated. After the merge node is installed and the parent node is updated, all processes that visit the node will try to finish the merge. They will first read the two nodes to be merged and will encounter the situation in fig. 6.5a. As no updates are allowed to be executed on the nodes $P$ and $Q$, all concurrent processes will reach the same conclusion whether a split is needed or not.

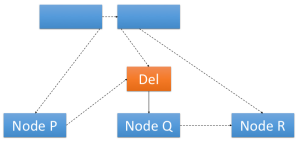
The trick here is simple: a merge node with two child nodes can be treated like one tree node. The split algorithm can be executed like there would be only one node. The process will load node $P$ and $Q$ into its local memory and generate a new node $S$ containing half of the elements from nodes $P$ and $Q$. Then it writes back these nodes to the storage (fig. 6.5b). Finally, it will try to install the split node with a compare and swap operation (fig. 6.5c). If this fails, another split node has been installed by another process. The split is then executed like a normal split (as illustrated in fig. 6.3).

#### Special case: different parents

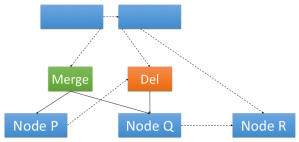
The Bd-tree is highly inspired by the Bw-tree [LLS13]. During our implementation work we found a bug in the paper and our Bd-tree design: If two nodes are merged that do not share a common parent, an important invariant is violated. If the



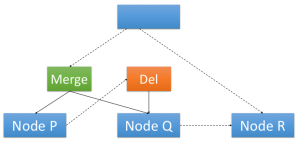

(a) Node Q is smaller than a given threshold



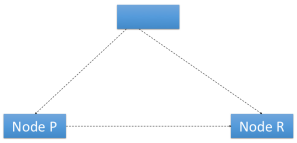

(b) Insert delete node




(c) Insert merge node




(d) Merge and update parent nodes




(e) Consolidate split node

Figure 6.6: Merge with two parents. Dashed arrows represent logical id links, solid arrows represent physical id links.

situation from fig. 6.6c occurs, node $Q$ would get deleted from the right parent node. But the new node which is created from merging node $P$ and $Q$ will contain keys that are bigger than the upper bound of the new parent. This means that future tree operations won't be able to find these keys, as they will search for them in the wrong sub tree. To fix this, both parent nodes need to be updated atomically. The simple trick of having logical and physical identifiers, however, only helps us to update one tree node atomically.

To solve this problem, we use a simple trick: we just merge the parents. This is illustrated in fig. 6.6. While merging the right parent, the parents key to node $Q$ is already removed in the first step. This has to be done to prevent another corner case: it might happen that the merging will result in an immediate split and the split will re-establish the initial position. While executing a merge (and potentially an immediate split) might be overkill, this happens very rarely in practice. Furthermore, compacting nodes more often will be beneficial for search operations.

## 6.10 Error recovery

A challenge for distributed systems, in general, is that every machine in the network can potentially fail at any point in time. This could happen due to software bugs (and since TellDB is a shared library, we can not control the level of robustness of a processing node), due to hardware failures or due to a power outage. Operations on the Bd-tree involve several operations on the key value store. Therefore it is important that a failing node can not leave the tree in a state that can not be

recovered from by other machines.

A Bd-tree is always in a consistent state. If a machine crashes while doing a split/merge operation, another process will eventually finish the work. Updates are atomic: either they succeed, or they fail. Therefore, a failing update is not problematic for correctness. It is important to point out that removing invalid updates due to failing machines while they are executing a commit is done by the transaction processing logic.

The only problem with failing machines is that they might generate garbage in the storage. Every operation will first generate a tree node (or a delta node) and then execute a compare and swap operation to install the change. If a machine crashes before this compare and swap operation succeeds, the written data will not get deleted. Our current implementation does not account for this problem, as error recovery is out of scope for this work.

## 6.11 Correctness

In this section we present the invariants (section 6.11.2) of the Bd-tree and a proof of liveliness.

To simplify this proof, we will assume a slightly simpler a variant of the Bd-tree:

- We assume that caching is turned off.
- The presented Bd-tree does not support concurrently merging and splitting two nodes. This operation is executed if a merge of two nodes would otherwise result in one node that is larger as the fill factor. Instead, we allow growing of nodes to up to twice the size of its fill factor.
- We assume, that the index key is of fixed size.

### 6.11.1 Storage Contract

The storage interface has to provide three function: *get*, *put*, and *erase*. We assume that the storage providing this interface is bug-free, i.e. no operation will ever violate its contract. We define the storage mathematically as a set:

**Definition 6.1** A storage state $S_i$ is a set of 4-tuples $\langle t, k, v, V \rangle$. $t$, $k$, and $v$ are natural numbers in the range $0$ to $2^{64} - 1$ and $V$ is a word over the alphabet $0, 1$. We call $t$ a table within the storage, $k$ key, $v$ version, and $V$ value.

**Invariant 6.1** For any tuple $\langle t_i, k_i$ and any storage state $S_i$ the expression the statement $|\{\forall\langle t, k, v, V\rangle \in S : t = t_i \wedge k = k_i\}| \in 0, 1$ is true.

Invariant 6.1 simply states that any $t$ and $v$ pair define at most one tuple within any storage state. With that we can now define the storage itself and a helper function $gets$.

**Definition 6.2** The storage $\varsigma = \{S_1, S_2, ...\}$ is a totally ordered set of storage states.

Furthermore, we define the versioning of tuples:

**Definition 6.3** $\forall\langle S_i, S_j\rangle \in \varsigma : i < j \implies \forall\langle t_i, k_i, v_i, V_i\rangle \in S_i : (\forall\langle t_j, k_j, v_j, V_j \in S_j : t_i = t_j \wedge k_i = k_j \implies v_i < v_j)$

**Definition 6.4** $gets(T) \rightarrow S_i$ is a function that returns the storage state $S_i$ at time $T$ where $T$ is a natural number representing time. For two different point in times $T_i$ and $T_j$, where $T_i < T_k \wedge gets(T_i) = S_k \wedge gets(T_j) = S_l$ it applies that $k \leq l$.

With these definitions, we can define the storage operations get, put, and erase. $get$ either returns the empty set or a set that contains one pair of version and value.

**Definition 6.5** $get(T, t, k) \rightarrow \{\langle v, V\rangle \in \{\forall\langle t_i, k_i, v_i, V_i\rangle \in gets(T) : t_i = t \wedge k_i = k\}\}$

**Definition 6.6** $put(T, t, k, v, V) \rightarrow \{0, 1\}$ will return $1$ iff for $v_i$ in $\langle v_i, V_i\rangle = get(T, t, k)$ it applies that $v_i = v$. Otherwise, $put$ returns $0$.

**Definition 6.7** $erase(T, t, k, v) \rightarrow \{0, 1\}$ will return $1$ if for $v_i$ in $\langle v_i, V_i\rangle = get(T, t, k)$ it applies that $v_i = v$ or if $get(T, t, k) = \varnothing$. Otherwise, $erase$ returns $0$.

Now that we defined when put and erase have to succeed (return $0$) we can define a contract for them to define what they have to do:

**Contract 6.1** $put(T, t, k, v, V) = 1 \implies get(T + 1, t, k) = \langle v + 1, V\rangle$

**Contract 6.2** $erase(T, t, k, v) = 1 \implies get(T + 1, t, k) = \varnothing$

We want to prove later that the Bd-tree is lock-free. To do so, we assume that put operations never fail with false-positives:

**Contract 6.3** $put(T, t, k, v, V) = 0 \implies v \neq v_i$ with $\langle v_i, V_i\rangle \in get(T, t, k)$

This last contract guarantees that for concurrent $put$ operations exactly one will succeed. On hardware CAS and LL/SC operations, this is often not guaranteed. However, a system could easily meet this contract by reading after a fail and automatic retry.

| Term | Definition |
|---|---|
| Physical pointer | An 8 byte wide unsigned integer used to index tree nodes. |
| Logical pointer | An 8 byte wide unsigned integer used to index physical pointers. |
| Pointer table | A table within storage that maps logical pointers to physical pointers. |
| Row pointer | An 8 byte wide unsigned integer. The mapped type of any Bd-tree. |
| Key | The key type within the Bd-tree. |
| Order $N$ | The order of the Bd-tree. |
| Height $H$ | The height of the Bd-tree is the number of inner nodes on the shortest path between the root node and a leaf node. |
| Indexed keys $K$ | The set of keys that are stored within the Bd-tree. The set $K$ changes over time. |
| Pointer table | A table within the storage that maps logical pointers to physical pointers. |
| Node table | A table within the storage that maps physical pointers to tree nodes. |

Table 6.1: Definitions of Bd-tree terms

Storage is an implementation of a class which provides functions get, put, and erase which behave like the functions $get$, $put$, and $erase$. The time variable $T$ is defined as the timestamp at the time when the request is executed. This means that we can not control T, but it is implicitly given instead.

### 6.11.2 Bd-Tree Invariants

#### Definitions

The terms used for formulating the invariants and the proofs are defined in table 6.1.

Within a Bd-Tree, every node has a type which is either *inner*, *leaf*, *deletion*, *merge*, or *split*.

A leaf and an inner node store a logical pointer called *logicalNextPointer* and two keys called *smallestKey* and *largestKey*. Furthermore, they all contain an array called *pointers* of pairs of key and pointer - row pointers for leaf nodes and logical pointers for inner nodes.

#### Invariants on Tree Nodes

**Invariant 6.2 — Ordering within Nodes.** For any inner or leaf node $n$ each key in

the pointer array is unique and the pointer array is ordered by key:

$$\forall i \in [1, sizeof(pointers) - 1] : pointers[i].key > pointers[i-1].key$$

**Invariant 6.3 — Key Range within Nodes.** Each key in the pointer array of an inner or leaf node is strictly greater than $smallestKey$ and smaller or equal than $largestKey$:

$$\forall p \in pointers : p.key > smallestKey \wedge p.key \leq largestKey$$

**Invariant 6.4 — Tree Order.** In a Bd-tree of order $N$, the size of any array within each inner and each leaf node has fewer than $2 \cdot N + 1$ and at most $\frac{N}{3}$ entries:

$$sizeof(pointers) \leq 2 \cdot N \wedge sizeof(pointers) \geq \frac{N}{3}$$

In the actual implementation, the minimal and maximal size of this array is a compile time constant and expressed in bytes instead of entries.

#### Invariants on Structure

**Invariant 6.5 — Children of Delta Nodes.** We call *split*, *merge*, and *deletion* nodes delta nodes as they signal an intended change to the tree and not a change by itself.
All children of all delta nodes are either leaf or inner nodes.

**Invariant 6.6 — Balanced tree.** For any leaf node $L$, the shortest path $H$ from the root node to the leaf node contains the same number of inner nodes.

**Invariant 6.7 — Global Order.** On a Bd-trees storing the set of keys $K$, the pointer of each key/pointer pair at array index $i$ within any inner node either points to a *split*, *merge*, or *deletion* node or to a Bd-tree that contains

- the keys $\{k \in K : k \leq pointers[i] \wedge k > pointers[i-1]\}$ or
- the keys $\{k \in K : kleqpointers[i] \wedge k > smallestKey\}$ if $i = 0$.

If the pointer does not point to an inner or leaf node this invariant applies to the key and

- all keys within the Bd-tree that is the left child of the *split* node if it points to a *split* node, or

- all keys within the Bd-tree that is the right child of the *deletion* node if it points to a *deletion* node, or
- all keys within the Bd-tree that is the left child of the *merge* node if it points to a *merge* node.

#### Invariants of Operations

**Invariant 6.8 — Complexity.** All operations within the Bd-tree have a complexity of $O(log(|K|))$.

The distinguished property of the Bd-tree is lock-freedom. We define lock-freedeom accordingly to [TP14]:

**Definition 6.8 — Lock-Freedom.** An algorithm or data structure is lock-free if for a finite but unbounded number of concurrent processes there will be at least one process that will finish its operation in a finite number of steps.

#### Contracts

Below are the contracts for the operations. They are said to start at $t_i$ and finish at $t_j$. Unlike cohesive operations, a simple operation can never starve.

**Definition 6.9 — NodeValue.** We define a NodeValue as

```
struct NodeValue {
  // A node which can be of any node type
  Node node;
  // The version of the logical pointer
  // pointing to this node at the time it
  // was fetched or nothing if it was fetched
  // from a physical pointer.
  optional<Version> ptrVersion;
  // The logical pointer that was pointing to
  // this node at the time it was fetched or
  // nothing if it was fetched from a physical
  // pointer.
  optional<logical_ptr> lptr;
  // The logical pointer that was pointing to
  // this node at the time it was fetched
  physical_ptr pptr;
};
```

**Contract 6.4 — Fetch Node.** We define two functions to fetch a node:

```
optional<NodeValue> fetch(physical_ptr ptr);
```

It satisfies the following contract:

**Ensure:** $ptrVersion = nothing$

**Ensure:** $lptr = nothing$

**Ensure:** **return** nothing if ptr never existed in the node table while this function was running.

**Ensure:** **return** a $NodeValue$ with node set to one value of the entry in the node table if this entry existed before the function started and never was deleted before it returned.

**Ensure:** **return** $NodeValue.pptr = ptr$.

**Ensure:** If the node did exist for some time during the time this function ran, it either returns a value that existed during this time period or it returns nothing.

The other function fetches from a logical pointer:

```
optional<NodeValue> fetch(logical_ptr ptr);
```

Its contract is functionally equivalent. However, all optional values will also be set. The function might fail if the logical pointer is changed while it is running and the old entry in the node table got removed.

Below are the contracts for the operations over the Bd-tree. They are said to start at time $t_i$ and either starve and never return or return at time $t_j$.

**Invariant 6.9 — Success of Operations.** We define $P(k) = k \in K \forall t_x \in [t_i, t_j]$ and $Q(k) = k \notin K \forall t_x \in [t_i, t_j]$. An insert or update on key $K$ will succeed if $P(k)$. It will fail if $Q(k)$. Otherwise, its result is undefined.
An erase on key $K$ will succeed if $Q(k)$. It will fail if $P(k)$. Otherwise, its result is undefined.

**Contract 6.5 — Find.** The find operation has the following signature:

```
optional<RowPointer> find(Key k);
```

If it succeeds, it will return a RowPointer that was at one point in time stored within the B-tree in a tuple with $k$.

**Contract 6.6 — Insert.** The insert function has the following signature:

```
bool insert(Key k, RowPointer p);
```

If it succeeds, all subsequent find operations called with $k$ will find $p$ unless an erase or update succeeded in between.

**Contract 6.7 — Update.** The update function has the following signature:

```
bool update(Key k, RowPointer p);
```

If it succeeds, all subsequent find operations called with $k$ will find $p$ unless an erase or update succeeded in between.

**Contract 6.8 — Erase.** The erase function has the following signature:

```
bool erase(Key k, RowPointer p);
```

If it succeeds, all subsequent find operations called with $k$ will fail unless an insert or update succeeded in between.

### 6.11.3 Lock-Freedom Proof

In this section, we are proving that the Bd-tree is a lock-free data structure. For this, we prove the following theorem:

**Theorem 6.11.1 — Lock-Free Bd-Tree.** The Bd-tree is a lock-free data structure - as defined in definition 6.8.

This means two things:

1. *Liveliness*: Any single process might starve. As it tries to make progress it might need to help other processes with split and merge operations forever and will never be able to complete its operations. However, there will always be at least one process that does not starve.

2. *Guaranteed Throughput*: As there will always be a process that will succeed after a finite number of operations a minimal throughput is guaranteed.

The Bd-tree is lock-free but not wait-free. Wait-free is usually defined as an algorithm where each operation finishes after a finite number of operations. Therefore, for wait-free algorithms, no process will ever starve. Achieving wait-freedom is much harder and wait-free algorithms are usually much slower than lock-free counterparts (see [TP14]).

The first observation we use is that all operations eventually try to read from or write to a leaf node. Lock-freedom can be proven by contradiction: we assume that there is a situation where all processes are starving.

The only situation that a process can starve is if it never succeeds to either reach the leaf node it wants to execute its operation on or if it never succeeds to rewrite the leaf node.

Because split and merge operations always start at the bottom of the tree we know that there are processes that did walk down to the leaf node they want to change.

If a read operation reaches the leaf node it is done. This contradicts with the assumption that all processes are starving. Therefore we assume that either no process is executing a read operation or all read operation are helping to execute split and merge operations and therefore never reach a leaf node.

A write operation will fail to rewrite the leaf node if it either has to initiate a new split or merge operation or if the conditional write fails. A conditional write of a leaf node can fail for two reasons: (1) Another operation updated the tree node or (2) another operation initiated a split or merge operation. (1) can not happen as this would mean that a process would have succeeded in executing its operation.

Therefore we know that on each leaf node that was already reached by a process one process succeeded to initiate a split or merge process.

The next observation is that split and merge operations always finish within a finite number of steps. For each step within a split and merge a conditional store will only fail if another thread either succeeded to continue to the next step or rewrote a leaf node (which cannot happen as this would again mean, that this processes successfully finished an operation).

Therefore it has to be the case, that merge and split operations are fluctuating and there will never be an actual operation that can change the content of the tree.

For a split to happen, the number of keys stored in a node has to be $\geq N$ and for a merge the number of keys has to be $= \frac{N}{3}$. Therefore, for all leaf nodes that are reached by write operation we know that the number of keys $|k|$ stored in that leaf is $\frac{N}{3} = |k| \vee |k| \geq N$.

Because of invariant 6.4 we know that $|k| < \frac{N}{3} \vee |k| > 2 \cdot N$ is never never true.

Therefore after a split, each resulting node will have $|k| \leq N$ keys. Another operation might merge the resulting left leaf node with its left sibling resulting in another leaf node with $|k| \geq N$ keys. However, this can only continue until the left most leaf node is split. The right sibling might only be split as the condition for merging does not hold.

For merging a similar analysis can be made. After $O(|K|)$, where K is the set of all keys stored in the Bd-tree, merge and split operations on the leaf level, for each leaf that is of interest for a write process it will be true that $|k| > \frac{N}{3} \wedge |k| < N$.

A write process will then reach the leaf node it wants to update. The only way it can fail at this point is if another process writes a new leaf node. However, this means that another process succeeded with its operation and this is a contradiction.

# 7

# Transaction Processing

Transaction processing is done at the processing layer. The storage does not need any knowledge about transactions. Tell implements a special form of snapshot isolation. Earlier versions of Tell also implemented locking and timestamp-based concurrency control, but those proved to be significantly slower on all tested benchmarks. Please consult [Loe15] to get more details on these implementations.

The most important aspect of Tell is that the transaction process happens nearly entirely within the processing layer. For Tell 1.0, the storage did not have any knowledge about transactions. In Tell 2.0, versioning is pushed down to the storage layer. This chapter will first describe how snapshot isolation is implemented in Tell 1.0. This is also described in more detail in [Loe15]. The last section describes changes which were done in Tell 2.0.

In this chapter I will assume that every transaction is executed by one single process in the processing layer. Currently, Tell only supports intra-transaction parallelism for read-only transactions. But since OLTP transactions are typically computationally cheap, intra-transaction parallelism is usually not beneficial.

This chapter covers transaction processing within TellDB. Chapter B provides the curious reader with a pseudo-code implementation of the transaction life-cycle within a TellDB instance.

## 7.1 Introduction

### 7.1.1 Problem Statement

A key value store that provides versioning is very useful by itself. In the previous chapter we presented a solution an index data structure so that we can support range queries.

However, we want to have stronger consistency guarantees. Optimally we want to have serializabilty guarantees or something that is only slightly weaker.

### 7.1.2 Requirements

The transaction implementation should at least provide snapshot isolation or repeatable read guarantees. These are still simple enough to reason about.

Snapshot isolation allows for write-skew. However, this can be solved, if necessary, by the user. One way to do that is by materializing the conflict which will generate a write-write conflict.

Repeatable read allows for phantom reads. A lot of workloads are, however, not sensible to phantoms and if they are they can be changes to account for this problem as well.

There are a lot of algorithms to support transactions that can be used for distributed systems. However, as we want to be able to run mixed workloads, some algorithms, mostly algorithms that rely on locking, won't work well. The problem with locks is, that they can block a concurrent read. In fact, most serializable concurrency control algorithms have this problem.

### 7.1.3 Contributions

This chapter is mostly here to provide a complete picture on Tell and does, by itself, not make a lot of contributions.

Most of the work described in this chapter has been presented before in [Loe+15], [Loe15], and [Loe+13].

The main contribution of this chapter is a solution how we can run long-running analytical transactions without slowing down parallel short-running OLTP transactions.

## 7.2 Architecture

Tell implements the shared data architecture and is therefore very modular organized. While Tell was built with the whole system in mind, nearly every component can be used in isolation. Tells architecture and its components are shown in

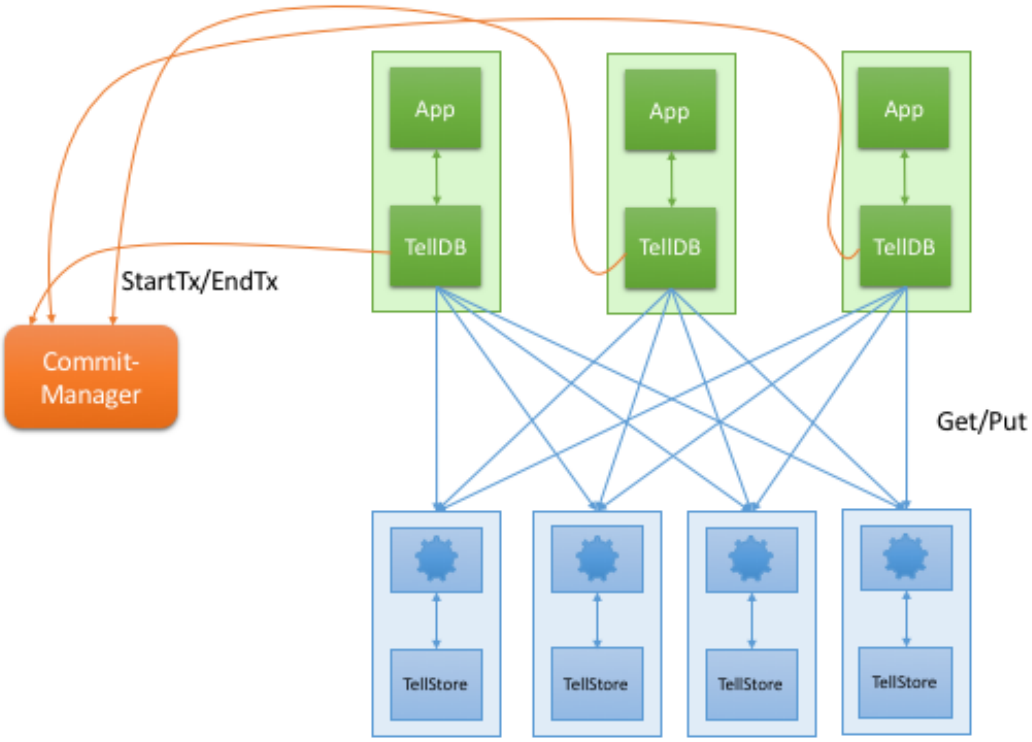



Figure 7.1: Components from Tell for elastic OLTP

fig. 7.1. Transactions are executed within a client library called TellDB. TellDB implements most transactional functionality as well as secondary indexes. The commit manager is a central service ordering the transactions. TellDB gets a new version id and a snapshot descriptor (described in chapter 7) at the start of a new transaction. When the transaction finished, it will inform the commit manager. The commit manager does not need to know, whether the transaction succeeded or not. TellStore is Tells default storage layer (for Tell 1.0 we used RAMCloud) and is oblivious with respect to transactions and secondary indexes. It is important to note here that the B-tree nodes for all indexes are stored as serialized objects within the storage. The underlying key value store implements a simple *get/put* interface. There is no communication between TellDB nodes. The two points of synchronization are the commit manager and the storage layer.

An application that uses Tell as a database management system would therefore link against TellDB instead of a database driver (like JDBC or ODBC). This implies that most database functionality will run in the middle-ware layer. This layer is usually easier to scale than the storage layer as it is ideally stateless (or at least has very little state). TellDB has very little state as well or no state at all if no transaction is running. This implies that TellDB is elastic and scalable. Adding machines is as trivial as removing machines from the cluster.

Each TellDB instance has read and write access to all data, as the data is simply stored in a key value store. The key value store partitions the data independent of the workload, in case of TellStore just by hashing the keys and distributing the key value pairs to the TellStore instances. While shared nothing databases try to limit data access from each transaction, every transaction within Tell will potentially read and write from and to all storage instances. This works well because the TellStore instances do not need to do any synchronization between them.

## 7.3 Transaction Life Cycle

The transaction life cycle is quite simple: whenever a TellDB node wants to start a transaction, it first requests a new snapshot descriptor from the commit manager. A snapshot descriptor basically describes a set of versions which belong to committed transactions and a version number for the new transaction.

Then it executes the transaction locally: that means it only reads from the underlying storage and cache all updates locally. During this time it might already detect conflicts if a tuple should be updated where the newest version is not in its read set. It aborts if such a conflict is detected.

For error recovery, specifically if a processing node dies, we write an undo-log. This is not literally a log but a set of entries within a private table on the storage which contains the changes it is going to make.

After writing this log it tries to commit: It writes back the updates, utilizing the compare and swap operation provided by the storage engine. If this succeeds, it informs the commit manager and then deletes the transaction log. Otherwise it rolls back all written updates, deletes the transaction log and informs the commit manager.

## 7.4 Commit Manager and Snapshot Descriptor

The commit manager is a special service which exposes two simple operations: start transaction and end transaction. The commit manager does not need to know whether a transaction succeeded or failed. The only state it has is the set of currently running transactions. The commit manager has a simple job: keeping track of all running transactions and constructing snapshot descriptors whenever a new transaction starts.

Whenever a process wants to start a transaction, it issues the start transaction command to the commit manager. The commit manager will then create a new version by incrementing a local counter, create a new snapshot descriptor, and

send the snapshot descriptor to the client. The snapshot descriptor is conceptually simple: it consists of the version of the newly started transaction and a read set. The read set is just the set of all versions of all transactions that have committed before the transaction started. Obviously, this version set is constantly growing as new transactions start. Sending this set over the wire would very soon become a bottleneck. To compress the snapshot descriptor we use a simple scheme. We encode the snapshot descriptor as two eight byte numbers, the version itself and a base version, and a bitmap the size $\lceil \frac{version - baseVersion}{8} \rceil$ bytes. The version is the version of the transaction that owns the snapshot descriptor. The base version is the highest version up to which all previous transactions finished at the time the snapshot descriptor got created (this idea was presented before, e.g. [BG81]).

The commit manager always keeps the newest snapshot descriptor in memory. Whenever a processing node starts a transaction, it will just increment the version number and send a copy of the snapshot descriptor to the processing node. If a processing node finishes a transaction it will send its version number back to the commit manager. The commit manager will then set the bit at position $version - baseVersion - 1$ to 1 within the bitmap of its local state bitmap and then it will increment the base version as long as the first bit within the bitmap is set to 1 and truncate the bitmap to the first bit that is set to 0.

Furthermore, the commit manager keeps track of the lowest active version: this is simply the lowest base version of all currently active transactions. This number is used for garbage collection: if the version of a tuple is lower than the lowest active version, it can be deleted unless there is no newer version of this tuple.

These two operations the commit manager provides are inherently serial. By default the commit manager is single threaded. This was never an issue, during our benchmarks: a single commit manager can only be saturated with 200'000 transactions per second and we could not produce this load with the hardware available to us. Nonetheless, we described in [Loe+15] and [Loe15] how to run several instances of a commit manager.

## 7.5 Snapshot Isolation

Tell implements Snapshot Isolation. This means that we allow for write-skew. While Snapshot Isolation is good enough as a concurrency guarantee for a lot of workloads, the occurrence of write-skews might be a problem for some applications. In this case, a user would need to work around this problem for example by materializing the potential conflicts. It might be possible to implement serializable

snapshot isolation [CRF08] on Tell, but this is a subject of future work.

Our snapshot isolation algorithm is designed to be simple and to keep the network communication needed for synchronization minimal. A TellDB instance starts a transaction by getting a snapshot descriptor from the commit manager. From that point on, the TellDB instance will work in read-only mode: it will read from the storage and keep updates locally. Therefore, whenever a conflict is detected before the transaction tries to commit, the transaction does not need to be rolled back as we can just discard the local memory containing the updates of the transaction. For performance reasons and to be able to detect conflicts earlier, we also cache all reads locally.

When TellDB fetches a tuple from storage, it will send the snapshot descriptor of the transaction together with the request to TellStore. TellStore will then give back the newest readable tuple together with a boolean which is true if the tuple is writable for this transaction and false otherwise. Writable in this context means that there is no newer version than the one TellStore returned. This boolean is stored in the cache together with the read tuple.

Tell does conflict detection twice: first when it writes to the cache, and then again at commit time. If a transaction tries to update a tuple that was not read before, TellDB will first fetch the tuple into cache. TellDB will then check whether the tuple was writable for the transaction at the time it was read. If this was not the case, it will immediately abort.

The main drawback of our snapshot isolation implementation are unnecessary aborts: a transaction immediately aborts when it detects a potential conflict. However, it might be possible, that the conflicting tuple will be rolled back later. Traditional single-thread resolution and two-phase locking do not have this problem. This is the price we pay for higher concurrency. In our experiments we observed very few unnecessary aborts.

## 7.6 Long running Transactions

TellDB will send its snapshot descriptor to TellStore for each get, put, and scan requests. This works well as long as the snapshot descriptor does not grow too large.

Even with a high load the snapshot descriptor will be sufficiently small as the bitmap will mostly include in-flight transaction. However, as soon as long and short running transactions are mixed together into the same workload the snapshot descriptor might become large.

As described in section 7.4, each transaction fetches a snapshot descriptor from the commit manager when it starts. This snapshot descriptor consists of the base version, the lowest active version, the version of the transaction itself as well as a bitmap of the size $transactionId - baseVersion$. This snapshot descriptor will then be sent to each storage node on each request.

While this works fine for OLTP workloads as well as for analytical workloads, it significantly slows down the execution of mixed workloads. The reason is that the bitmap within the snapshot descriptor will grow very large: if the OLTP system is generating a load of $10,000$ transactions per second (a load Tell can easily sustain), and one analytical transaction takes 1 minute to complete, the bitmap will grow to 75KB. Sending 75KB of data to each storage node for each request is, however, a huge overhead and the network throughput will become a bottleneck much earlier. Furthermore this will increase the latency if get and put requests significantly.

There are several countermeasures we could implement for this problem. Our first approach was to compress the snapshot descriptor. This works pretty well in general, since we can assume that most transactions will be short-running. Therefore, within that bitmap, most bits will be set to 1. This makes it simple to do efficient compression. However, it will also slow down query processing. Storage nodes will need to decompress the snapshot descriptor for each request. Therefore, the current implementation does not use compression for snapshot descriptors.

Another possible solution is to cache the snapshot descriptors on each storage node instead of sending it with each request. While this does not decrease the size of the snapshot descriptors, it lowers network load. But it will increase the load on the storage nodes, as they will need to keep track of the active snapshot descriptors. A processing node would need to send the snapshot descriptor to each storage node it wants to get data from and then inform the storage nodes, when the snapshot descriptor can be evicted again. Apart from this additional overhead, there is another major drawback: failure recovery. If a processing node crashes while executing a transaction, its snapshot descriptor would never be deleted. Therefore the current error recovering protocol needed to be changed.

Instead of these solutions, we implemented two additional transaction types: read-only transactions and analytical transactions. While the introduction of these transaction classes does not solve the problem of long-running transactions in general, it does solve it for a lot of use-cases. In the future, Tell should implement snapshot descriptor compression after a certain threshold for the size of the descriptor is reached. Thanks to read-only and analytical transactions, snapshot

descriptors never reached a problematic size during our experiments. Compression was implemented within Tell 1.0 and can be ported to Tell 2.0 as soon as it is needed.

### 7.6.1 Read-Only Transactions

Read-only transactions optimize transactions that guarantee not to write to the storage. A read-only transaction will, like read-write transactions, fetch a base version, a bitmap, and its version from the commit manager when it starts. The only difference is that the commit manager will immediately mark this new transaction as readable for all other transactions. This allows growing the base version beyond the versions of all read-only transactions. Therefore, read-only transactions will never contribute to growth of the snapshot descriptor.

The commit manager will still need to keep track of read-only transactions, as it is not allowed to increase the lowest active version as long as read-only transactions could potentially read this version. Therefore, the commit manager needs to keep two distinct bitmaps: one for the read-write transactions and one for the read-only transactions.

Adding the version of an uncommitted read-only transaction to the read set of another transaction is not problematic: since the read-only transaction will never write anything to the storage, other transactions will never read anything written by a read-only transaction. It is important to note that the processing node needs to make sure that the transaction does not write to the storage. This mechanism is implemented in TellDB: it will throw an exception as soon as the user tries to write with a read-only transaction.

### 7.6.2 Analytical Transactions

Whenever a transaction sends a scan request to a storage node, it will also send its snapshot descriptor. The shared scan will then first evaluate all selection predicates. If all selection predicates are satisfied by a tuple, the scan process needs to check whether the tuple is within the read set of the transaction that issued the query. For a tuple, the storage knows two numbers: $validFrom$ and $validTo$. $validFrom$ is the version of the transaction that wrote this version of the tuple, $validTo$ the version of the transaction that either deleted the tuple or wrote a new version of it. Such a tuple is in the read-set of a transaction iff $validFrom$ is readable for the transaction and $validTo$ is not readable ($validTo = \infty$ for the newest version). The code for this check is drafted in fig. 7.2.

The code to check the snapshot descriptor can be optimized better than what is presented in fig. 7.2 as the snapshot descriptor is known during compilation time

```
bool isReadable(const Snapshot& sd,
                uint64_t validFrom,
                uint64_t validTo)
{
    if (validTo < sd.baseVersion()) {
        return false;
    }
    // check the bitmap
    if (sd.versionMap[validTo - sd.baseVersion()] == 1) {
        return false;
    }
    // validTo is not in the read-set of the transaction:
    // at this point, we know that the tuple is valid iff validFrom
    // is in the read set of the transaction.
    if (validFrom < sd.baseVersion()) {
        return true;
    }
    return sd.versionMap[validFrom - sd.baseVersion()] == 1;
}
```

Figure 7.2: Checking whether a tuple is readable for a transaction

and this code is compiled to machine code at run time with LLVM. Nevertheless: checking the version of a tuple is a significant overhead for the scan threads.

To reduce this overhead we introduced the notion of analytical transactions. Analytical transactions are, within Tell, read-only transactions that do not require the absolute newest version of the data. The only difference between an analytical transaction and a read-only transaction is that the analytical transaction does not get a version bitmap from the commit manager. That means it will only read data written by transactions before the base version at the point the analytical transaction started. This is a legal optimization, as it just reorder these transactions

```
bool isReadable(const Snapshot& sd,
                uint64_t validFrom,
                uint64_t validTo)
{
    if (validTo < sd.baseVersion()) {
        return false;
    }
    if (validFrom >= sd.baseVersion()) {
        return false;
    }
    return true;
}
```

Figure 7.3: Checking whether a tuple is readable for an analytical transaction

- it does therefore not violate any serializability guarantees.

With only a base version to check against, the readable-check within the scan gets significantly easier. This code is illustrated in fig. 7.3.

## 7.7 Commit Protocol

While a transaction is running, it is not writing to storage. At commit time, all updates are cached an need to be written back to storage. Commit is a four step process.

1. Write back the undo-log.
2. Write back the updates.
3. Write back the index changes.
4. Inform the commit manager that the transaction completed.
5. Delete the transaction log.

TellDB also needs to write back a transaction log for error recovery. It writes back a very simple and compact undo-log. This is necessary, as the machine executing the transaction might crash at commit time. Therefore another node needs to roll back this transaction. Each client machine creates a table within TellStore for its undo-logs when it starts up and deletes it when it shuts down gracefully. Within that table, there is one tuple per transaction where the undo-log is stored. This undo-log is just a list of tuple it is going to write. So if the machine crashes, another machine can scan this table and revert all changes that it wrote. Additionally, the undo-log contains all index updates that the transaction will try to execute. This is not strictly needed but it makes error recovery simpler and less expensive.

As soon as this undo-log is written, TellDB tries to write back all updates. TellStore will do a conflict detection on every write: TellDB will send its snapshot descriptor with every request. If TellStore detects a conflict on one of the updates, TellDB will roll back and abort. Therefore, TellDB implements a first writer wins protocol. The drawback of first writer wins is there might be cyclic dependencies between two more concurrent transactions, resulting in unnecessary aborts. For example: if transaction $A$ is writing tuple $x$ and then tuple $y$ while transaction $B$ is writing tuple $y$ and then tuple $x$, both transactions will detect a conflict and both

transactions will abort. However, first writer wins allows committing several transactions concurrently without any direct communication between the processing nodes. The storage is the only point of synchronization.

As soon as writing back the updates succeeds, the transaction succeeded. But the updates are still not readable for new transactions. In the next step, the index updates are executed. The reason we wait until this point to write back the index is that rolling back index operations is potentially expensive. But index changes usually don't fail unless there is a uniqueness constraint violation.

The last step during a commit is to inform the commit manager that the transaction finished. The commit manager will then mark the version of the transaction as readable for newly started transaction.

Our commit protocol therefore works in two simple phases: the processing node is the master asking all slaves (the storage nodes) whether they can commit (have conflicts) and then commits or aborts.

The reason we do not need a more complex protocol like Paxos or two-phase commit is because of the shared undo-log. If a processing node fails, another machine can read its undo-table and revert all in-flight transactions.

If the storage does not implement the versioning protocol described in chapter 3, the above protocol does not work. Tell 1.0 therefore implemented a slightly different protocol. But conceptually not that much changed. The main difference is that for each key, the storage (RAMCloud) stored a serialized array of version-tuple pairs. Whenever a tuple got read, the processing node received all versions of that tuple and it wrote back a new array of version-tuple pairs. To do conflict detection, the compare and swap feature from RAMCloud was used. Garbage collection of old versions was done lazily on the processing node: it would just not write back versions which are not readable by any active transactions. Obviously, this resulted in a higher network load and more memory consumption on the storage - but for pure OLTP workloads, the network load was usually not the bottleneck of the system.

### 7.7.1 Recovery

If a processing node fails, it might have partially written some transaction to storage. This prevents any other transaction from writing to tuples that have been written by this machine until another machine rolls back these transactions.

To do so, the commit manager will periodically ping hosts with open transactions. If a host does not answer for some amount of time, it will be declared dead and its identifier will be sent to a processing node.

To prevent undefined behavior during network partitions, each processing node pings the commit manager regularly. If it can not reach the commit manager it will crash.

All undo-logs are stored within a table on storage. Its name can be computed with the process ID. This process ID has to be globally unique. For a rollback a processing node will read this table and execute all undo-logs.

Executing an undo-log is trivial: it contains the version of the transaction and a list of all keys that should have been written during the transaction. The processing node will send these keys and the version of the transaction to the storage. The storage supports a roll-back function that just removes the newest version of the key if it has the version from the log. Executing an undo on a key that was not yet written therefore does not have an effect, as the newest version on storage will be either larger or smaller than the one of the aborting transaction.

To roll back the Bd-Tree, the log will also contain a list of all Bd-Tree identifiers and the corresponding changes it intended to execute. The rollback process will walk all Bd-Trees and delete the key/version pair within that tree if it exists.

The undo-log entry of each transaction will be deleted after the rollback did successfully finish. At this point the commit manager is also informed, that the transaction did finish. Therefore, even if the processing node executing the rollback fails, another processing node can continue with the rollback. When the undo-table is empty, the table will get deleted.

## 7.8 Bd-Tree

Our Bd-Tree implementation can only handle unique keys. This is a problem, if we want to index a column which does not have a uniqueness constraint. Furthermore, we can not delete keys from the Bd-Tree at the point we update or delete a tuple. The reason is that the Bd-Tree is not aware of the snapshot isolation mechanism. Therefore, older transactions might still need to read a tuple after it got deleted and if we would delete the tuple immediately from the index, older transactions would not be able to find the tuples by using the Bd-Tree.

We solve both these problems by constructing a composite key. For indexes with uniqueness constraints the constructed key will have the form $< key, validTo >$ and indexes without uniqueness constraints we construct the tuple $< key, value, validTo >$. The $value$ will always be an 8 byte number: either the id of a child node within the Bd-Tree, or the id of a tuple. $validTo$ is an eight byte unsigned integer as well. $validTo$ is the version up to which the entry is valid. Therefore a transaction can

simply check whether $validTo$ is within its read-set (which will be always false for $2^{64} - 1$) - if it is, it means that the transaction can see the update operation that resulted in the deletion of the key from the index, and it can therefore safely ignore the index entry. If a transaction wants to delete from the index, it just sets the $validTo$ field to its own version number.

This simple version handling within the Bd-Tree, while fast and easy to implement, has some minor drawbacks. First of all, the Bd-Tree can not detect all uniqueness constraint violations, as it might happen that a deleted key is still readable for the updating transaction. Therefore TellDB has to check the uniqueness constraint with an index operation before it can update the index. It does however prevent uniqueness constraint violations produced by concurrently committing transactions.

The second drawback is that the Bd-Tree needs to be garbage collected. This can not be done within the storage layer, as storage is oblivious to the fact that it is storing index nodes. TellDB does garbage collection lazily whenever it reads a leaf node: if the $validTo$ field is smaller than the lowest active version, the entry will get deleted. As a consequence, an index entry might never get deleted, if the leaf node it is stored in never gets read from a processing node. But this could be easily solved by running a distinguished garbage collection process on a processing node.

## 7.9 Query Execution

Analytical queries are usually computationally more expensive than online transactions. TellStore provides the client with an interface to read large amounts of data. It provides a high data throughput as opposed to the high operation throughput of the get/put functions. But TellStore, by design, does not support the execution of complex computations (see section 2.3 for a description of the reasons). Therefore, after receiving the data, the processing node has to execute these computations.

The way this would work is illustrated in fig. 7.4. In that case TellDB is used to start a transaction and issue scan requests to the storage nodes. In that case, all computation is implemented in C++ by the user (as TellDB does not yet implement all SQL operators).

The main benefit and the main drawback of this approach is simplicity. For queries that can push down selections with a strong selectivity for each scan query they issue, this execution model works well. In that case most of the processing (the filtering) is done in parallel by all storage nodes and the query executor only

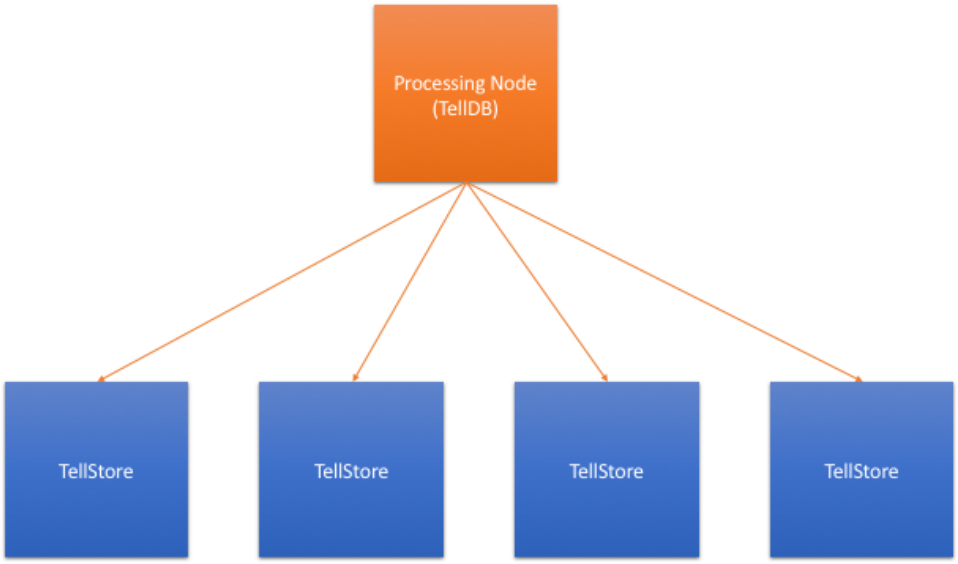



Figure 7.4: Executing analytical queries like online transactions

needs to process a small amount of data.

There are two scenarios where this execution model reaches its limits: if CPU time is the limiting factor, as only the CPUs from the machine that executed the query can be used, or if intermediate results are larger than the main memory of the machine. It could very well be that both CPU time and main memory become a bottleneck. If CPU time is the bottleneck, the user has no other choice than to wait for the result. If main memory becomes a bottleneck, the process can write intermediate results to some secondary storage (for example local disk or a private non-transactional table on TellStore). This, however, will slow down the execution significantly.

### 7.9.1 Distributed Query Execution

For more complex workloads, distributed query execution engines get more attractive. Instead of one machine per query, a distributed query engine can execute one query with several machines (intra-query parallelism).

The basic idea here is illustrated in fig. 7.5. This idea is not new and there are several distributed query engines on the market. The basic idea here is that one machine, called the master node, starts a transaction and generates a query plan. It will then split the work into several tasks and send them (together with the snapshot descriptor) to other nodes in the cluster (slave nodes, or workers). Each worker will then process a partition of the data and send back its results to the

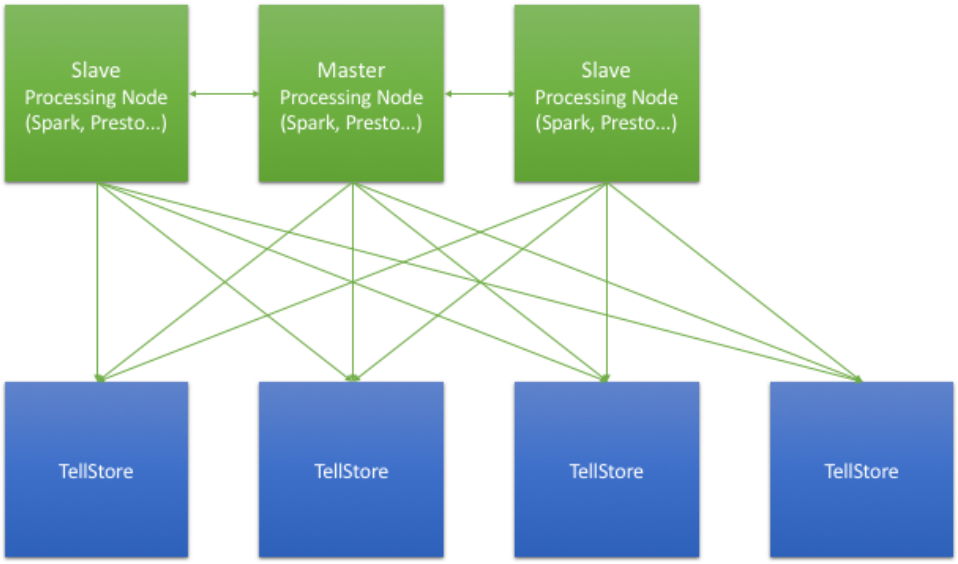



Figure 7.5: Using existing analytical query engines

master node. The master node will then aggregate the results and send the final result back to its client.

Building an analytical query engine for Tell is out of scope for this work. We did, however, implement a Spark [Zah+10] layer and a Presto [Fac16] plugin for TellStore. Spark and Presto are two popular distributed query engines which support several storage engines.

Distributed systems like Spark and Presto usually try to partition the data in order to be able to process only parts of it. The shared-data architecture on the other hand tries to give each processing node a complete view over all data. This looks at first like a fundamental conflict. The important tool here is selection push-down: by pushing down a selection, each node can create its own partition. This partition is completely independent of the physical storage location of the data. The query engine could, therefore, use knowledge about the distribution of the data, to generate an optimal partitioning over its data. If the query processor, for example, knows that the keys are uniformly distributed, it could push down the selection $key \equiv_{48} 12$ to read the twelfth partition out of 48 partitions. If it does not have any knowledge about the data, we could introduce a random field that just contains a uniformly distributed random number and use that for partitioning.

Our benchmarks with Spark, which are not included in this thesis, show that Spark is unable to saturate TellStore with scan requests. We implemented and ran TPC-H on Spark, but we measured that only a minor fraction of the processing time

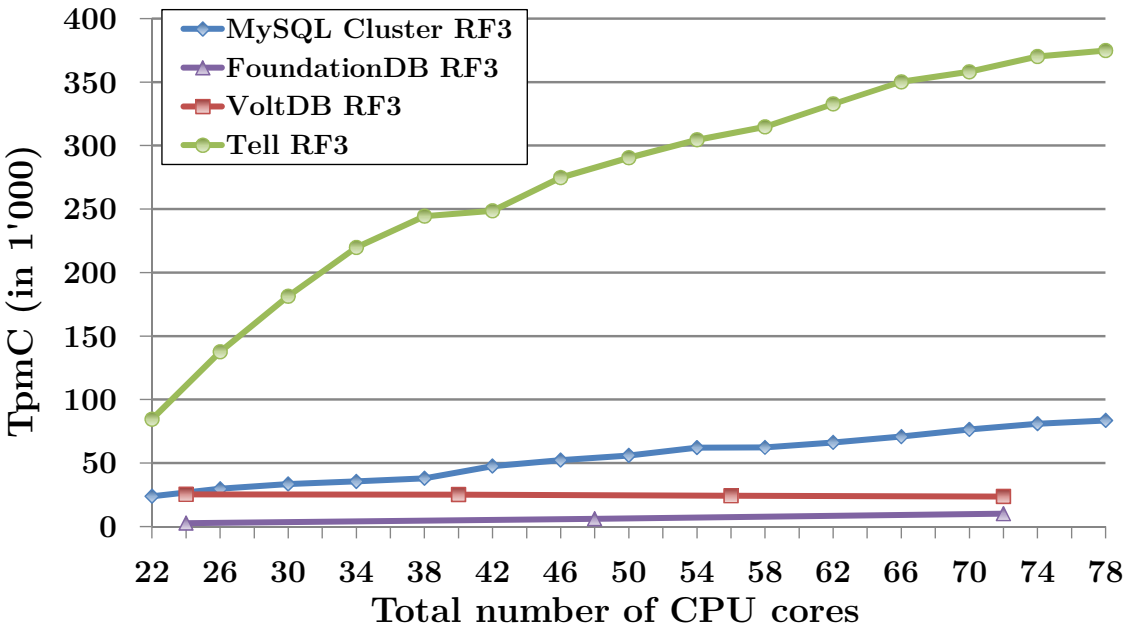



Figure 7.6: TPC-C scale out with total number of cores

was used for fetching the data - this made the results useless for us as it did not help us to compare different storage implementations with Spark. These results are coherent with the results published in [CO15]. Spark had similar problems. We therefore used another workload to benchmark the storage which executes each query on a single node and is heavy on scans with predicate push down and projections.

## 7.10 Experimental Results

As the experimental results with Tell 1.0 are presented in detail in [Loe15], this section will only present the most important result from the work. In chapter 8 I will compare these results with the new results we got after optimizing Tell. Previous work focused more on different concurrency control algorithms (snapshot isolation, locking, and timestamp ordering). As snapshot isolation was the clear winner in all our benchmarks, we abandoned the implementation of the other concurrency control algorithms in favor of a cleaner and simpler design.

To test elastic OLTP we ran, apart from some micro benchmarks which are not presented in this work, the synthetic TPC-C workload. TPC-C simulates a transactional order-entry system. The results are presented in fig. 7.6.

As one can see, Tell can scale out well with more resources. The reason VoltDB is doing much worse (VoltDB gets slower when more resources are added) is due to the way it partitions data. VoltDB creates one partition per CPU core. Each

transaction is then executed single-threaded on one core. This works great, as long as no transaction is accessing more than one partition. This has some rather big advantages: no concurrency control is needed (which speeds up the execution of transactions) and VoltDB performs well if there is high data contention (the worst-case with high data contention is that there will be no concurrency anyway, as all transactions need to be serialized). However, as soon as more than one partition is read or written to, VoltDB needs to lock these partitions and execute the transaction on both partitions in parallel. Within TPC-C though, not all transactions can be executed within the context of a single partition.

Unfortunately, we do not know how many machines Tell can scale to. The cluster which was available to us to run benchmarks consisted of twelve machines. Most probably though, the first bottlenecks we would reach would be either the message rate within the network or the commit manager. But according to some micro benchmarks, the commit manager can easily handle up to 200,000 transactions per second - which would roughly translate to six million TpmC. But as shown in previous work, the commit manager can be distributed as well.

# 8

# Tell 2.0

The main reason leading to Tell 2.0 was that we were unable to perform analytical workloads on Tell. This was not due to a fundamental flaw of Tell, but due to the inability of RAMCloud to execute fast scans over data within storage.

Nevertheless, Tell 1.0 suffered from major problems which we wanted to get rid of during the rewrite and redesign.

The first main issue was its inflexibility: Tells API consisted of a set of C++ templates and a set of concepts which had to be implemented by the user. The schema was described by some classes and the workload was described by a domain specific language. The workload was then compiled to machine code by the C++ compiler. The main benefit of this approach was, of course, its speed, as the workload specific code could be analyzed and optimized by the C++ compiler. But for every new workload Tell had to be completely recompiled. This also meant that ad-hoc queries were impossible and adding a SQL-engine would be virtually impossible.

The next major problem was that Tell 1.0 was using a synchronous execution model. But a synchronous execution model does not work well together with Infiniband. Infiniband gets its low latency partly because it bypasses the operating system kernel when sending and receiving data. Therefore, a thread can not block within the kernel while it waits for a response. Tell did, however, simulate a socket interface on top of the Infiniband verbs library. It did that by using one open connection to storage and one dedicated thread, polling on this connection. For each transaction, there would be a heavy-weight OS thread that puts requests

onto a queue which will then be sent by the poll thread to the storage. Whenever an answer was received, the poll thread would put the answer into a predefined data structure and set a boolean which is polled on by the caller. The transaction thread would then observe this boolean and call yield within a loop. This design was a quite horrible hack. Profiling showed that around 70 percent of the overall CPU time was spent within the kernel - more specifically within the yield system call and to execute context switches.

With Tell 2.0 a lot of effort was spent to design a system that would get rid of these limitations. This section describes how storage was designed and the execution model looks like. I will not, however, describe how data is organized within the storage as this was covered in part I.

## 8.1 Tell Components

Tell has a very modular design. This was a necessity, as we implemented three different key value stores and wanted to reuse as much code as possible. Tell 2.0 is rewrite except for the Bd-tree. In the following sections I outline the components of the system. All these components can be used independently from each other. All components can be downloaded as independent projects[1]. Our build-system, however, supports the whole system to be built within one project [2].

#### Crossbow

Crossbow is a utility library. The reason it is called Crossbow will be obvious to a Swiss reader: Willhelm Tell, after whom the system was named, was a hunter who used a Crossbow to shoot an apple from his son's head. Later in the story he also shot Gessler, the local despot, using the same Crossbow.

Crossbow is a collection of C++ libraries exposing general purpose functionality. These libraries are used by all other components of the system. Crossbow contains the following libraries:

- *string*: a std::string compatible string class that allocates small strings (<= 31 bytes) on the stack.
- *singleton*: A general purpose, thread safe implementation of a singleton, allowing to control the lifetime of the held object. The design of this singleton was heavily influenced by Andrei Alexandrescus Loki library[3].

[1] https://github.com/tellproject
[2] https://github.com/tellproject/tell
[3] http://loki-lib.sourceforge.net/

- *single consumer queue*: A lock-free queue which allows multiple producers but only one consumer. The main advantage over other implementations is that the consumer can atomically get all elements off the queue.

- *program_options*: A small header-only library to parse command line arguments. The design was influenced by Boosts program_options library[4], but Crossbows version of the library is simpler and does not have any link time dependencies.

- *allocator*: The allocator library implements two special allocators: (1) a chunk allocator and (2) an epoch implementation. The chunk allocator is a very simple allocator that can only be used to allocate memory, but not to free it. Instead all memory is freed when the allocator is destroyed. This helps to optimize processes that have a clear begin and a clear end of execution (for example a parser that needs to allocate a lot of small objects but will at the end of its lifetime not need them anymore)
  The epoch allocator is an implementation of the epoch garbage collector [KL80]. This is heavily used to manage the memory for lock-free data structures. The main problem of lock-free data structures is that when an object is removed from the data structure, it can not be immediately freed, as another concurrent thread might still have a pointer to the given memory. Epoch works by tracking threads that enter and leave critical sections, making sure to free the memory only after all threads that were active when free was called on the object have left and therefore do not hold any references to the given element.

- *logger*: A very simple logging library, existing solutions were too heavy for our needs.

- *infinio*: This is the most sophisticated library within Crossbow and implements an Infiniband library on top of the low-level Verbs library[5]. It implements the whole execution model using user-level threads. This library is documented in more detail in section 8.1.1.

#### Commit Manager

The commit manager as described in section 7.4. The commit manager uses Crossbows infinio for communication and is completely independent of the other

[4] http://www.boost.org/doc/libs/1_60_0/doc/html/program_options.html
[5] http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf

components of the system. In future though, we are planning to use this component as a coordinator for the cluster. But this is future work.

#### TellDB

TellDB is one of the main components of Tell and basically resembles the functionality of Tell 1.0. TellDB is not a program, but a dynamic library which is supposed to be used by a middleware server. It exposes an API to the user that allows starting transactions, create tables and issue queries. TellDB is currently tightly glued to TellStore, so using another key-value store would be a major engineering investment. This was done mostly because of the unique execution model and creating a general API of the storage is nontrivial. In the end, it was mostly a time issue and running on top of another storage was not a feature we anticipated.

TellDB, while it does not support SQL, does provide a complete abstraction layer to the user. The user does not need to create indexes or care about the caching - all this is done by TellDB. One of the main differences to Tell 1.0 (apart from the execution model which is implemented in the context of Crossbow infinio) is that the interface is completely dynamic. This will help us in the future to interpret SQL without the need to recompile Tell. Furthermore, it is now much simpler to provide interfaces to other programming languages

#### Bd-Tree

This is the implementation of the Bd-Tree described in section 7.8. This Bd-Tree implementation allows to implement a simple API for any storage layer that supports atomic compare and swap operations. This implementation is used within TellDB to construct secondary indexes.

#### TellStore

TellStore is our key value store. Unlike most other key value stores, TellStore exposes a scan functionality: a user can scan tables and push down simple aggregations, selections, and projections. TellStore by itself is actually very modular, but this will be described in more detail in the next chapter.

#### TellJava

TellJava is a slim layer around TellDB that exposes the scan functionality of TellStore and the transaction capabilities of TellDB to Java. TellJava is very limited in terms of functionality and only exposes an interface for read-only access to the underlying storage. Also, we currently only allow for scans in Java programs to read data from the storage (i.e. get requests can not be issued from Java).

This was a design decision. The execution model of TellDB is asynchronous in

nature and TellJava provides a blocking interface on top of that. As a consequence, we are losing a lot of efficiency in the process. For scans this is a small overhead as response times are generally high. For get/put workloads though, this blocking interface would very soon become a bottleneck. Nonetheless, this functionality is still very useful to run analytical workloads.

#### TellSpark

TellSpark is written in Scala and uses TellJava to implement Sparks DataFrame API. TellSpark allows the use of SparkSQL to run analytical workloads on data that is stored within TellStore.

#### TellPresto

TellPresto is a plugin for the Presto SQL engine. Our plugin does not support updates, although these could be added later. It depends on TellJava to execute scans on partitions of data.

### 8.1.1 The execution model

The execution model is implemented as a part of Crossbow infinio. From a high level perspective, we implemented a simple event queue. The design of infinio was heavily influenced by Boost asio. A first design planned to implement an infiniband layer for Boost asio but it turned out that Boost asio is not well suited for RDMA, as its design is mostly tailored towards asynchronous socket programming.

Infinio runs several poll threads. It can be configured by the user how many threads it should start, but it should be no more than one per CPU core. The poll thread behaves differently on the server than on the client, therefore both are briefly discussed here.

On the server side, each poll thread maintains a set of connections. The Infiniband Verbs library does not provide any API calls to establish a connection to another host. We use the RDMA communications manager (rmda_cm)[6] which was implemented by OpenFabrics, to do that. At startup, the poll thread registers a set of memory blocks at the NIC and starts polling for new requests. Incoming connections are assigned to poll threads using round robin. Since the event queue is stored in the main memory of the process, we do not need to call into the kernel to read arriving events. Therefore the poll thread will busy-wait for new events for a certain amount of time. Busy-waiting has the advantage that it lowers the response time but it also produces a lot of CPU load without doing any work if there are no incoming requests. Therefore, after a few thousand poll iterations,

[6] http://linux.die.net/man/7/rdma_cm

we use call *epoll_wait*[7]. This is a system call that instructs the kernel to wait for new events and wake up the process as soon it receives an interrupt from the NIC. This behavior gives us the best of both worlds: short response times when the system is under high load and minimal resource wastage under low load.

A network request consists of a set of commands to be executed from the storage. This batching is done to improve network throughput (more on batching will be described later when I talk about the client side). The poll thread will then execute all commands synchronously. This works fine, as these commands are inexpensive to execute. More than one thread can be used to poll the event queue and all polling threads can execute incoming requests in parallel. While executing the commands, the poll thread writes the answers back to preallocated memory blocks and puts them on the send queue. It is important to note that this two-sided protocol is only used for get and put operations. Scan operations use one-sided RDMA, this will be described in the next chapter.

This execution model on the server side is very simple and fast, as no context switches are done and no callback functionality is used. On the client side though, things get a bit more complicated. The client needs to be able to run many tasks in parallel and we can not afford to have one thread per task. There are two common solutions to this problem: use callback functions for asynchronous operations or use user-level threads. We decided to implement user level threads (we call them fibers), as it simplifies the usage of the library.

On the client side, the user can configure the number of operating system threads to use. In practice, no more than one per CPU core should be started to minimize scheduling overhead. Each of these threads will open a connection to each storage node. This is affordable with Infiniband, as the maintenance of connections is much cheaper than with a TPC/IP stack. It will then run a very simple loop:

1. Poll from the task queue for new fibers.

2. Call into each fiber by making a context switch. The fiber will execute without getting preempted. Whenever it makes a request, it will just put the request into a block of memory. Within the user code that runs in a fiber, every function which involve communication to storage returns a future object. If the network buffer is full, the fiber will flush it to the NIC and get a new one. A fiber can decide to wait for a result at any point in time. To do so, it will call into the main loop.

[7] http://man7.org/linux/man-pages/man2/epoll_wait.2.html

3. The main loop will then put all requests to the send queue of the corresponding Infiniband connection.

4. It will then check whether there are new messages on the receive queue. If there are, it will set the future objects from the corresponding fibers to point to the request within the received block and call back into the fibers, one after another (again, by doing a context switch).

5. Go back to step item 1.

This execution model has several advantages. It is very easy to use, as a user can program tasks in the same way that he would program single threaded executions of task. The only difference is that calls the storage are not blocking but return a future object. Calling get or wait on this future object will potentially block.

The biggest advantage is that the client can now do efficient batching of requests. For every connection, the client will keep one network buffer it tries to fill up with requests before it sends the requests to storage. As a consequence, batching happens at two levels: each task can issue several requests without sending anything over the network which results in task batching (or transaction batching in case of TellDB) and several parallel tasks batch requests automatically together. Therefore, more concurrency will result in better batching behavior. Without this batching, it would not be possible to saturate the network throughput with simple get and put requests.

Please keep in mind that this RPC mechanism is only used for get and put, scans use this RPC facility only at start and end. One-sided RDMA is used to ship scan results. For Simple get/put operations, one-sided communication is not beneficial, as only small amounts of data are shipped for each request.

Due to its execution model, a TellDB client can potentially run into these bottlenecks:

- *CPU load*: Depending on the calculations a transaction needs to do, it could run into a CPU bottleneck. By adding more cores to this layer, more transactions can be executed in parallel, but the transactions won't finish earlier. The user however, is free to provision CPU cores to execute parts of a transaction in parallel. In practice, OLTP workloads to issue transactions that are not CPU heavy, therefore this should not be a big issue.

- *Network Throughput*: We did micro benchmarks that showed that we can saturate the network throughput if all packets sent over the network are larger than 4 kilobytes. With our batching mechanism, this packet size

should always be reached. Reaching the network throughput with an OLTP workload will happen only for extremely high loads: with 1 Million TpmC in the TPC-C workload, we sent around 400MB of data over the network. Therefore, in all our experiments, the network throughput was never a bottleneck.

In practice, the bottleneck is usually the CPU - either in storage or in the processing layer. This can be solved by adding more CPUs (which in our case meant adding more machines). It is expected that at one point either the commit manager or the network will become a bottleneck. The commit manager can scale as well and for the network it would be possible to add an additional NIC to each machine. Since our cluster is pretty small (12 machines), we could not run experiments to find these limits. We would expect Tell to scale well to one or two racks of machines. After this point, the network load might become too high for the network switches to handle.

## 8.2 Experimental Results

There are two main differences between Tell 1.0 and Tell 2.0: we use a new storage layer (TellStore instead of RAMCloud) and we implemented a new execution model based on user level threads and asynchronous execution.

TellStore is not significantly faster than RAMCloud for get and put operations (and get/put are the only operations we issue while executing an OLTP workload). Nonetheless, when running Tell on top of the key value store, TellStore performs significantly better than RAMCloud. With six RAMCloud storage nodes, Tell 1.0 reaches a maximal throughput of slightly more than 1.1 million TpmC (New Order Transactions per Minute). With six TellStore servers however, Tell 2.0 reaches more than 1.7 million TpmC. This is an improvement by more than a factor of 1.5. Furthermore, the abort rate of New Order transactions decreases from 3 percent for Tell 1.0 on RAMCloud to 1.1 percent on Tell 2.0 on TellStore. 1% of New Order transactions have to be aborted according to the benchmark specification. Therefore, around 2% of the New Order transactions abort on Tell 1.0, and around 0.1% abort on Tell 2.0 are due to conflicts.

There are several reasons for these improvements. Probably the most important ones are our execution model, native multi version concurrency control within the storage, and the way we do batching.

While RAMCloud supports batching, it can only batch together certain types of requests. For example, atomic operations can not be batched with read requests. Therefore there are still some requests that need to be executed one by one. This

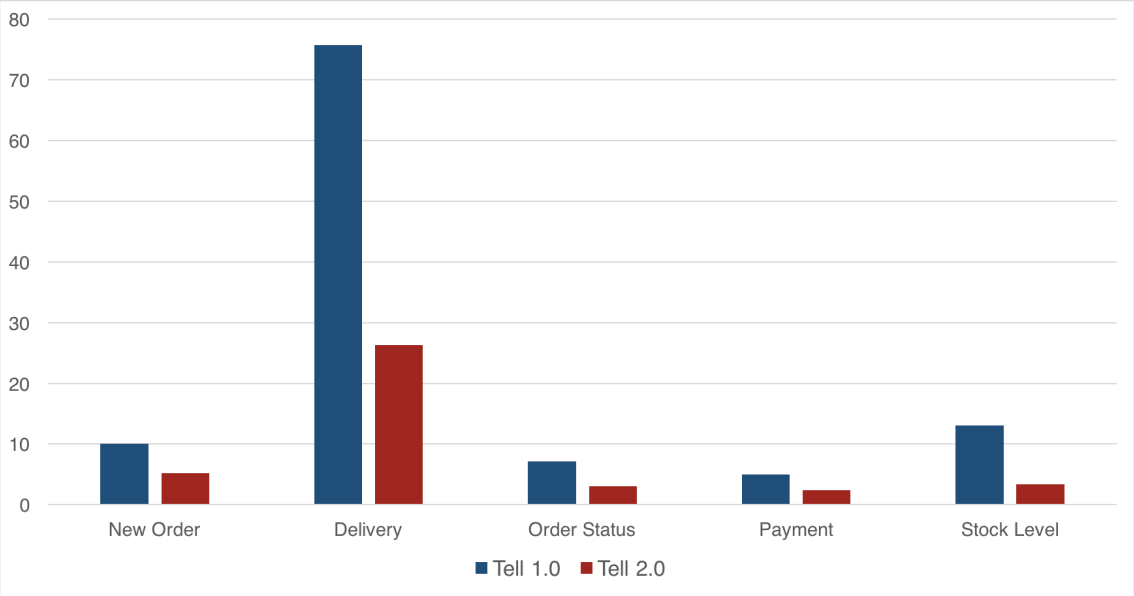



Figure 8.1: Response on maximum load with 6 storage nodes for the TPC-C queries (in milliseconds)

means that when a transaction tries to commit, it needs to send all its write requests to storage and wait for an acknowledgment. This slows down commit time.

Furthermore, TellStore provides native support for versioning. While with RAMCloud, TellDB always has to fetch all valid versions of a record (and it even has to do garbage collection which is now done within TellStore), TellStore will only deliver the newest version of the record that is in the read set of the transaction.

But the biggest difference comes from our new execution model. As described before, due to our blocking interface on top of a non-blocking API, a huge amount of time is spent within the OS kernel. While Tell 2.0 still has some overhead for transaction scheduling, the overhead is much smaller (a context switch is done in less than 100 cycles - but of course there is usually some time needed to rewarm the caches). The main effect of this is that the response times under high load got significantly lower. This is illustrated in fig. 8.1. The response times for Tell 1.0 on RAMCloud are, depending on the transaction, between a factor of 2 and 4 higher than for Tell 2.0 on TellStore. The lower response time also explains the lower abort rate: since transactions run for a shorter amount of time, there are fewer transactions that might read and write the same data.

It is important to note here that this is not a comparison between RAMCloud and TellStore. Most of the optimizations we did within Tell 2.0 would be possible to do within Tell 1.0 as well. Therefore, the conclusion of this experiment is not that TellStore outperforms RAMCloud but Tell 1.0 outperforms Tell 2.0.

# 9

# Mixed Workloads

While it was already shown that Tell can execute OLTP workloads, I did not yet show, how the different key value stores perform under different workloads. It is expected that the row store and the log structured storage will perform well for OLTP workloads while the column map storage engine will perform well for analytical queries.

This section will therefore mostly compare our three key value store against each other. As a baseline, we use Kudu [Lip+15], a key value store from Cloudera that tries to bring fast scans to key value stores.

## 9.1 Experimental Setup

To run our experiments, we used a cluster of 12 machines with the following specification:

- Two Intel Xeon E5-2609 CPUs. Each of the CPUs has four cores and runs at 2.4GHz.
- Each machines has 128GB of main memory, divided into two NUMA regions (64GB each).
- One Infiniband card, Mellanox ConnectX-3, installed at NUMA region 0.
- 250GB SSD.

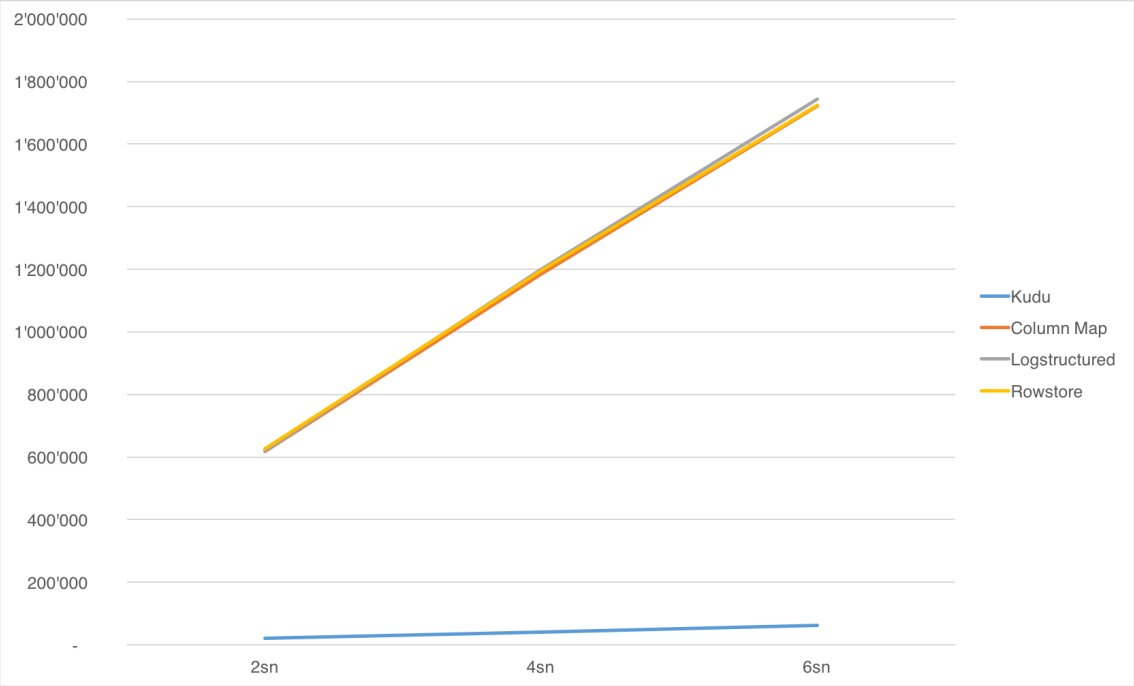



Figure 9.1: Maximal Throughput in TpmC, scaling number of storage nodes

As our system under test was the storage, we always use the number of storage nodes as a basis. Since the process for a storage is always pinned to one NUMA-node (as this achieves higher performance), there are at most two storage nodes running on one machine.

## 9.2 OLTP Workload

In order to test how well the different storage engines perform under an OLTP workload, we ran the TPC-C benchmark. The throughput result is presented in fig. 9.1.

As the system under test is the storage, and not like before TellDB, we are now scaling the number of storage nodes (again: one node is one NUMA node which means that for two storage nodes, one machine is needed). The setup is the following: there is one client process running on a dedicated machine that generates load and measures response time and throughput. Then, there are a number of middleware processes which link against TellDB (or the Kudu client library). These middleware servers execute the transactions. We made sure that the middleware servers never become a bottleneck. In this experiment, we always have twice as many middleware nodes than storage nodes (the middleware is pinned to one NUMA region as well). For example: for the experiment with six storage nodes, we use three machines to run TellStore and six machines to run

| | 2 storage nodes | 4 storage nodes | 6 storage nodes |
|---|---|---|---|
| **Kudu** | 20'686 | 40'556 | 61'179 |
| **Logstructured** | 619'037 | 1'199'167 | 1'743'504 |
| **Rowstore** | 625'478 | 1'194'296 | 1'722'880 |
| **Column Map** | 617'280 | 1'183'379 | 1'720'880 |

Table 9.1: Throughput numbers for TPC-C (in TpmC)

the middleware service (TellDB), one machine for the client and one machine to run the commit manager (11 machines in total). With Kudu, we implemented the transactions in a similar way. Kudu, however, does not support transactions But this benefits Kudu so we just accepted that Kudu might commit conflicting transactions.

There are several things we can observe from fig. 9.1: all systems scale linearly. One can also see immediately that all TellStore variants outperform Kudu. This has sever reasons:

- While TellStore is an in-memory storage, Kudu writes to disk. In this benchmark, we made sure that Kudu can hold all data in it's cache, but it will still need to write to disk. This makes the comparison to Kudu a bit unfair.
- Kudu does not provide an Infiniband interface. Therefore, it sends all messages over 10G Ethernet, while TellStore can utilize the Infiniband NIC.
- Kudu can only efficiently batch write request within one transaction. TellStore on the other hand, makes inter-processing batching.
- Tells execution model allows for more parallelism: Tell performs best with 8 concurrent transaction per CPU core - for Kudu we got the best results with two concurrent transactions per core.

The more interesting part, however, is the comparison between the different TellStore variants. The interesting results here is that there is virtually no difference between the TellStore variants. The differences between the variants are more readable in table 9.1. The column map performs surprisingly well. This has two reasons: whenever a client is writing, the update will be written into the write-optimized delta. For reading, our optimized materialization makes sure that the whole tuple can be efficiently read from the main (if one reads from the delta, the data is in row-format anyway).

## 9.3 Mixed Workload

### 9.3.1 The Huawei-AIM benchmark

The Huawei-AIM benchmark is a benchmark developed within our group in collaboration with Huawei. The benchmark models a use-case from the telecommunication industry. It is described in [Bra+15].

This benchmark operates on a single wide table with a large number of attributes (more than 500). Each record within this table holds the information for one customer. Storage needs to handle 20'000 customers per machine - this means that we store 10'000 customers per storage node (as there are two storage nodes per machine).

These customers will generate 10'000 events per second - an event is typically a phone call or a network interaction. For each event, the system needs to read this record, update a few attributes and write it back to storage. Therefore, within Tell, each event corresponds to one get and one put request to storage. The events will access the tuples randomly.

**Query 1**

```
SELECT AVG (a_25)
FROM wide_table
WHERE a_35 > [ALPHA]
```

**Query 2**

```
SELECT MAX (a_24)
FROM wide_table
WHERE a_28 > [BETA]
```

**Query 3**

```
SELECT (SUM (a_22)) / (SUM (a_25)) as cost_ratio
FROM wide_table
GROUP BY a_28
```

**Query 4**

```
SELECT city_zip, AVG(a_35), SUM(a_32)
FROM wide_table
WHERE a_35 > [GAMMA] AND a_32 > [DELTA]
GROUP BY city_zip
```

**Query 5**

```
SELECT region_id, SUM(a_29), SUM(a_36)
FROM wide_table
WHERE subscription_type_id = [T] AND category_id = [CAT]
GROUP BY region_id
```

**Query 6**

```
(
  SELECT entity_id
  FROM wide_table
  WHERE a_34 = (
            SELECT MAX(a_34) FROM wide_table WHERE country_id=[CTY]
        )
  LIMIT 1
)
UNION
(
  SELECT entity_id
  FROM wide_table
  WHERE a_41 = (
            SELECT MAX(a_41) FROM wide_table WHERE country_id=[CTY]
        )
  LIMIT 1
)
UNION
(
  SELECT entity_id
  FROM wide_table
  WHERE a_13 = (
            SELECT MAX(a_13) FROM wide_table WHERE country_id=[CTY]
        )
  LIMIT 1
)
UNION
(
  SELECT entity_id
  FROM wide_table
  WHERE a_20 = (
            SELECT MAX(a_20) FROM wide_table WHERE country_id=[CTY]
        )
  LIMIT 1
)
```

**Query 7**

```
SELECT MAX(a_22 / a_25)
FROM wide_table
WHERE value_type_id=[V]
```

Table 9.2: Analytical queries from the Huawey-AIM benchmark

While these events are processed, another system will issue analytical queries to the system (for example to generate advertisements or give some benefits to

| | |
|---|---|
| ALPHA | uniform in [0,2] |
| BETA | uniform in [2,5] |
| GAMMA | uniform in [2,10] |
| DELTA | uniform in [20,150] |
| T | uniform in [0,3] |
| CAT | uniform in [0,2] |
| CTY | uniform in [0,3] |
| V | uniform in [0,3] |

Table 9.3: Parameter generation for the queries in the Huawei-AIM benchmark

users). The benchmark defines seven queries which have to be run. The system will randomly (using a uniform distribution) select one query and execute it. The queries can be found in table 9.2. The parameters are defined in table 9.3.

### 9.3.2 Experiment Setup

There are some important differences between the AIM system (for which this benchmark was originally designed) and Tell. The AIM system can execute updates directly on storage, while Tell has a simple get/put interface. This means that to process an event in Tell, the client needs to first fetch the record and then write it back to storage, while in AIM the request can be sent directly to the corresponding storage node. The same is true for the queries: TellStore can only execute scans. Within the AIM system, these queries are implemented in C++ within the storage. This means that AIM is only able to execute exactly this workload (unless the storage layer is changed) while Tell can execute all kind of workloads.

To implement this workload, we implemented the components illustrated in fig. 9.2. For all experiments, we run four storage nodes. These storage nodes hold 40'000 records in total. We then have two different kinds of middleware servers: event processors and query processors. There are always as many event processors as storage nodes, but we scale the number of query processors. Query processors answer queries - each query processor can execute up to two queries in parallel (this is not a sharp limitation, but as our system under test is the storage layer, we wanted to make sure that the middleware does not become a bottleneck). The query processor issues scan requests to all storage nodes and uses the result to execute the query (doing the joins, group by etc). For each query, the query processor will issue up to eleven scan requests to each storage nodes. To make

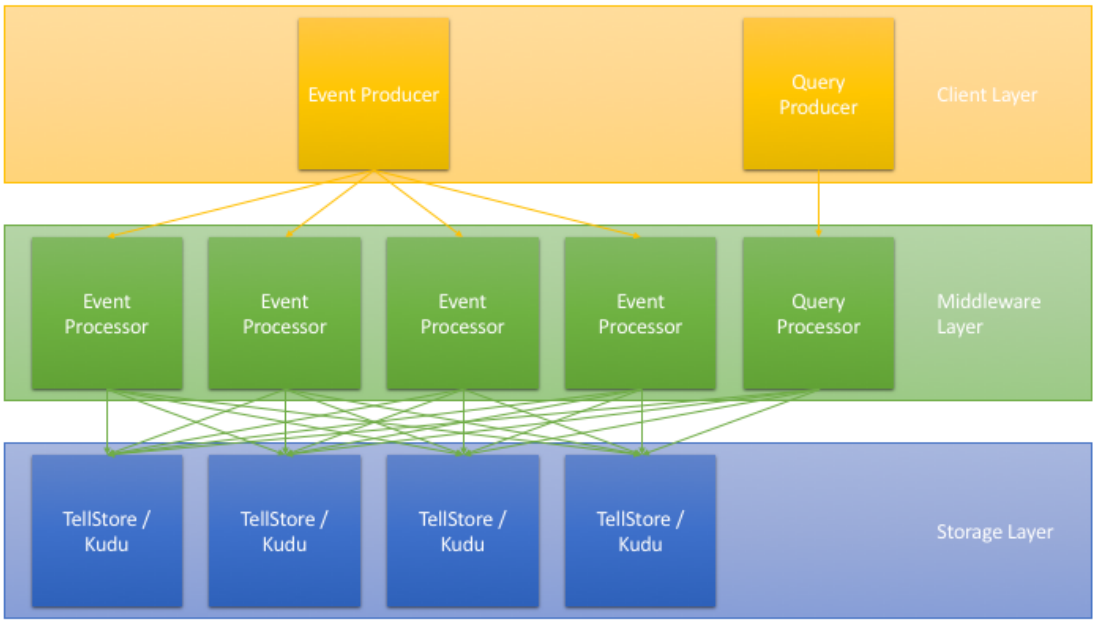



Figure 9.2: The setup for the Huawei-AIM benchmark

efficient use of sharing within the scans, all scan requests are issued in parallel.

On the top layer, there are two clients: an event producer and a query producer. The event producer generates 20'000 events per second and sends them to the event processors. The query producer generates queries and waits for the answer from the query processor.

### 9.3.3 Metrics

For this benchmark, we measure throughput and response time of the queries. For the events we either issue 20'000 events per second (as the benchmark required), or none. We do this to see the influence event processing has on the query response time and throughput. We measure at the client side, this means that the query response time is not the same as the scan time. We don't show any results for the event processing, as all storage variants can process the given number of events without any problems.

### 9.3.4 Results

#### Running with a single client

To analyze the response times, we need to run the system with one query client. Due to sharing within the scan, queries will start to slow down each other, as one heavy query will issue scan requests at the same time as a concurrently running light query.

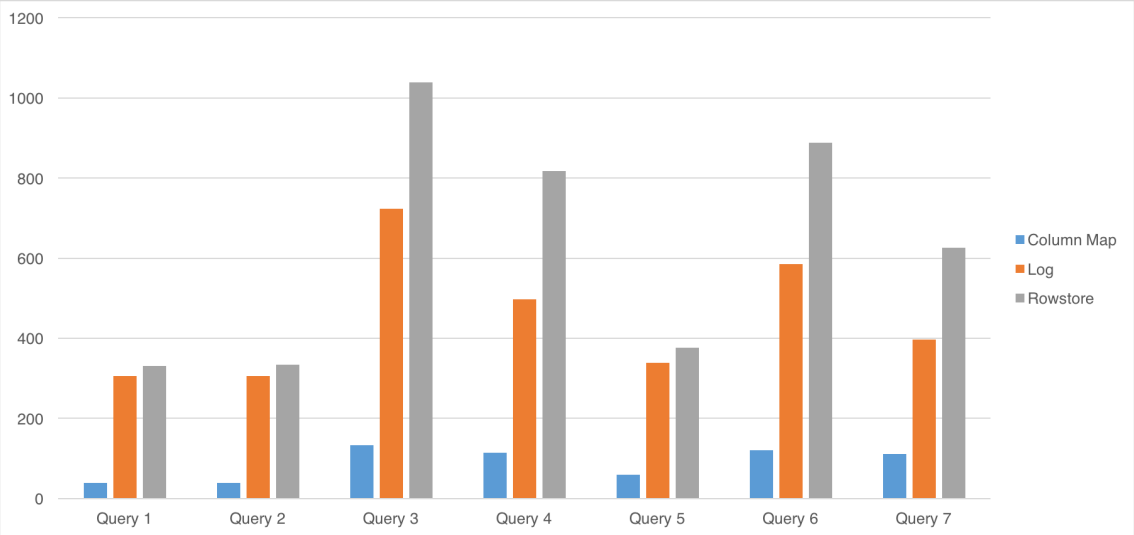



Figure 9.3: Response times for all approaches with one query at a time (in milliseconds)

| | TP without events | TP with events | Slowdown |
|---|---|---|---|
| **Column Map** | 11.57 | 10.02 | 13.4% |
| **Log** | 2.22 | 2.08 | 6.3% |
| **Row** | 1.58 | 1.39 | 12.4% |

Table 9.4: Query throughput with one client

For one query client, we measured the response time for each query. The results for this experiments can be found in fig. 9.3. As one can easily see, the column map wins against the other variants by nearly one order of magnitude. This is not very surprising, as the columnar layout helps to speed up scans. However, it might surprise the reader that the log structured storage outperforms the row store in this experiment. This is because the log structured storage does not have a dedicated garbage collection thread but it does garbage collection while scanning. Since this workload is read-only, the scan does not need to do any garbage collection. This gives storage the advantage of having one core more at disposal to execute scans. Therefore each scan thread has to scan less data. We also ran the benchmark on Kudu, but we did not include the results in fig. 9.3. The Kudu response times are so high that the figure would become unreadable. Kudu needs between 1.8 (query 2) and 9.6 (query 6) seconds for each query. This is surprising as Kudu is a column store and it should cache all tuples (according to the authors, Kudu tries to use all available memory to cache the data).

By looking at the throughput we can see that the column map is significantly

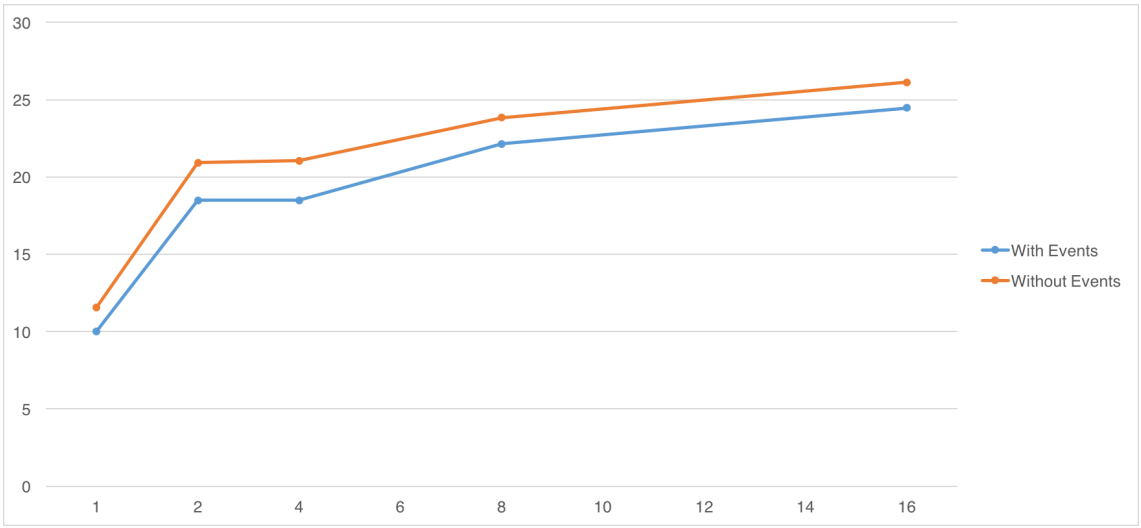



Figure 9.4: Query Throughput scaled with the number of clients)

faster than the other variants. Again, this is an expected result. In table 9.4 we show the throughput numbers for the queries with concurrent updates (with the event producer running) and without concurrent updates (read-only mode). As one can see, TellStore experiences a slow down of roughly 10% when it is run with concurrent updates. We consider this to be acceptable. The log structured variant experiences a smaller slow down than the two delta main variants. This is an expected result, as the delta main approaches have to follow newest-pointers from the main to into the delta whenever the scan reads a tuple that was updated but not yet merged into the main table by the garbage collector.

#### Scale-Out

In another experiment, we scaled the number of clients issuing query requests. We executed this experiment only for the column map, as it is clear that the other variants will not perform well on this benchmark. We did execute the benchmark for Kudu, but the numbers are not presented here (Kudu can not reach a throughput of 0.5 queries per second). These results can be found in fig. 9.4. There are two main observations from this experiments: the storage can scale very well for up to two clients and the slow down gets smaller as we scale out.

The scaling for two clients shows the effects of the shared scan. From one to two clients, the throughput nearly doubles. As the storage needs to execute up to 22 queries in parallel for two clients (as each client issues up to 11 concurrent scan requests), this result shows that query sharing works quite well. For more than 2 two clients, TellStore does still scale, but not so fast anymore. The reason is that at this point the predicate evaluation is becoming a bottleneck - that means

that we are now CPU bound.

# 10

# Future Work

Tell is quite a big project and there is still a lot to do.

On the storage side replication, durability, and elasticity are the most important missing features. These features, however, should not be too difficult to implement as there is enough existing work on how to solve. The next big issue is the scan memory manager which should do remote memory management.

An interesting question would be, what the correct way is to do query processing on a shared-data system for analytical workloads. As every processing node has a full view over all data, it could freely decide how to partition the data. Currently, data is always partitioned by primary key - but this might not always be the most efficient solution.

Furthermore it might be interesting to evaluate, whether it would be possible and beneficial to do query indexing for the shared scans. This could potentially make scan sharing even more efficient.

TellDB ships without a SQL processor. It is not clear how a SQL optimizer for Tell should work. Tell does not produce histograms and automatically decide whether to use table scan or index scan is non-trivial.

Intuitively, TellStore would be a great candidate to run on non-volatile memory. But NVRAM is currently not on the market and the hardware architecture is not yet defined. Nonetheless, NVRAM will bring new, exciting challenges and processing data on future hardware is an interesting field.

The fact that Tell uses snapshot isolation makes it a good candidate as a temporal database system. Supporting time-travel should be trivial and it should

also be possible to do temporal aggregations on data stored in TellStore. [Pil+16] describes ParTime, a parallel temporal aggregation algorithm that would be straight forward to implement on top of Tell.

As Tell decouples storage and computation it could be an interesting candidate for multi tenancy. With the rise of cloud computing I expect that multi tenancy will get more attention from industry and research.

# 11

# Conclusions

This dissertation presented Tell - a distributed system that can execute mixed workloads. Tell is scalable, elastic and efficient.

Any distributed system has a well-defined architecture. Choosing the right architecture for the job is not only difficult, the decision for a certain architecture has consequences to each part of the design.

Storing and retrieving data has to be efficient in any database management system. A system that optimizes for mixed workloads has to be able to deliver large amounts of data quickly. Additionally, queries on small ranges and point queries have to execute fast in order to be able to execute OLTP workloads. While most existing storage solution can either provide a high data throughput or a high query throughput, a storage for Tell has to deliver both.

Most interesting OLTP workloads contain queries on small ranges. In order to execute these queries efficiently some form of index is needed. Indexing data in a distributed is difficult. The user should be able to decide which dimension of the data should be indexed.

One of the hardest problems in computer science is concurrency. Probably one of the most successful solutions which helps users to reason about concurrency is transactional processing. Transactions give some guarantees, that some form of data races do not happen. However, transactions just push this complexity down to the database layer. For a distributed database we have to find an efficient way to implement transactions.

Each problem imposed some requirements:

The chosen architecture has to scale well and be elastic. In order to be suitable for mixed workloads we have to separate compute and storage. Otherwise, concurrent queries will heavily fight for resources. The architecture has to be flexible so that a system administrator can react adequately to new workload requirements and resource contention.

The storage system has to provide get, put, and scan functionality. To minimize slowdown of concurrent operations on the same data, it needs to provide a feature to version data. Simple operations have to be executed directly on storage. This helps us to minimize network load. Furthermore, the storage has to be elastic and it has to scale well.

An index over data distributed to several machines imposes several requirements. It has to be able to execute range queries, therefore it has to be a ordered index. To store the index on a distributed storage it has to be easy to break it into small pieces and store those pieces on several storage instances. At the same time, the overhead to access and modify an index should be minimal. In particular, we want to minimize network traffic due to index processing.

Short-running and long-running transactions should impact each other only minimally. Therefore we need a concurrency control algorithm where read and write operations do not block each other. Furthermore, the system has to be able to handle a lot of short-running as well as a small number of long-running transactions.

This thesis presents a definition for the shared-data architecture. This architecture separates storage and compute which greatly simplifies elasticity and scales well on computer and storage. We found that this architecture is well suited for mixed workloads.

We explored the design space for distributed key value stores supporting a scan operation and version. As a result we built three variants for such a storage and conducted experiments with different kind of workloads.

This thesis also presents the Bd-tree. This data structure executes all operations lock-free, utilizing the LL/SC functionality of its underlying storage. The proposed caching scheme allows to minimize network traffic. Furthermore we provide invariants and contracts for all operations and presented a sketch of a proof.

For transaction processing, TellDB uses a variant of snapshot isolation. By defining three classes of transactions, TellDB can scale to thousands of concurrently short-running transactions several several long-running transactions.

All these ideas have been successfully integrated into Tell. Tell is a collection of

programs and software libraries. Tell's complete source code has been published to github under a liberal open source license.

Our experiments have shown that Tell can run a variety of workloads. It performs exceptionally well on pure OLTP workloads, on analytical workloads, and on mixed workloads.

# Appendices

# A

# Bd-Tree Pseudo-Code

This chapter presents C++-like pseudo-code for a Bd-tree implementation. The original implementation can be found on [1].

The real code of the Bd-tree consists of several thousand lines of code. This pseudo-code leaves out certain details of the original implementation and can only be used to better understand the workings of the data structure.

One important feature of the Bd-Tree implementation omitted in this pseudo code is range queries. The actual implementation uses an iterator model to execute range queries on an index.

For simplicity, this Bd-Tree does not use the cache described in section 6.5. However, adding the cache to a working Bd-tree should be trivial.

Furthermore, this Bd-Tree does not support a concurrent split and merge operation. This operation would be used if the resulting node from a merge would be larger than the max size of a tree node. Instead, this Bd-Tree allows for each tree node to grow twice the maximal defined size.

## A.1 Storage Interface

The Bd-tree stores its data in a storage which has to provide get/put functionality as well as a Load-Link/Store-Conditional (LL/SC) function. The following defines such an interface. For simplicity, this interface assumes blocking operations to storage.

[1] https://github.com/tellproject/bdtree

```
using table_t = uint64_t;
using key_t = uint64_t;
using version_t = uint64_t;

constexpr version_t infinity = std::numeric_limits<uint64_t>::max();

struct Value {
  std::unique_ptr<char[]> buffer = nullptr;
  size_t size = 0;
  // The version is used for the put command.
  // The following semantic is used:
  //   - If version == infinity, only write the
  //     value if the tuple does not exist
  //   - Otherwise: only write the value if
  //     remote(version) == version
  version_t version = infinity;
};

struct Storage {
  // Reads the value of (t, k) into
  // value. It also writes the version
  // of the tuple into value, which can later
  // be used to do an LL/SC operation on this
  // key
  bool get(table_t t, key_t k, Value& value);
  // Write the value to storage if the version
  // does not conflict (see above). Basically,
  // get and put provide an LL/SC interface.
  // As a side-effect, put will write the current
  // version into value.
  bool put(table_t t, key_t k, Value& value);
  void erase(table_t t, key_t k);
};

// For simplicity, we assume that the storage interface
// is accessible over a global variable.
Storage storage;
```

## A.2 Data Structures

The Bd-tree uses several data structures which is stores in-memory. These are defined here.

```
// Serialization methods not shown. We assume
```

```
// the appropriate specializations exist.
template<class T>
void serialize(const T&, Value&);

template<class T>
void deserialize(T&, const Value&);

using logical_ptr = uint64_t;
using physical_ptr = uint64_t;

constexpr uint64_t invalid = std::numeric_limits<uint64_t>::max();

struct IndexKey {
  static const IndexKey& max();
  static const IndexKey& min();
  bool operator<  (const IndexKey& rhs) const;
  bool operator>  (const IndexKey& rhs) const;
  bool operator<= (const IndexKey& rhs) const;
  bool operator>= (const IndexKey& rhs) const;
  bool operator== (const IndexKey& rhs) const;
  bool operator!= (const IndexKey& rhs) const;
  //...
  IndexKey() {}
  template<class T>
  IndexKey(T v) {
    // ...
  }
};

struct TreeNode {
  // the largestKey is equal to the largest
  // key stored within this tree node
  IndexKey largestKey;
  // The lowest key is smaller than the smallest
  // key stored within this tree node. It is equal
  // to the largest key of its left sibling.
  // Therefore all keys within the range
  // (lowestKey, largestKey] are stored in the
  // subtree with this tree node as root.
  IndexKey smallestKey;
  uint64_t logicalNextPointer;
};

struct InnerNode : TreeNode {
  using ContainerType = std::vector<std::pair<IndexKey, logical_ptr>>;
```

```
  ContainerType pointers;
};

struct LeafNode : TreeNode {
  using ContainerType = std::vector<std::pair<IndexKey, key_t>>;
  ContainerType pointers;
};

// We want to be able to use std::lower_bound
template<class T>
bool operator< (const std::pair<IndexKey, T>& rhs,
                const IndexKey& lhs) {
  return lhs < rhs.first;
}

template<class T>
bool operator< (const IndexKey& lhs,
                const std::pair<IndexKey, T>& rhs) {
  return lhs < rhs.first;
}

template<class T>
bool operator< (const std::pair<IndexKey, T>& lhs,
                const std::pair<IndexKey, T>& rhs) {
  return lhs.first < rhs.first;
}

namespace std {
template<class T>
struct less<std::pair<IndexKey, T>>
{
  using type = std::pair<IndexKey, T>;
  bool operator() (const type& lhs, const type& rhs) const {
    return lhs.first < rhs.first;
  }
};
}

struct SplitNode {
  physical_ptr left;
  logical_ptr right;
};

struct DeletionNode {
  physical_ptr toDelete;
```

```
};

struct MergeNode {
  physical_ptr left;
  physical_ptr right;
};

using Node = boost::variant<InnerNode,
                            LeafNode,
                            SplitNode,
                            DeletionNode,
                            MergeNode>;

struct BdTree {
  // These values have to be
  // initialized by the user
  const table_t pointerTable;
  const table_t nodeTable;
  const table_t counterTable;
  // The storage must allow to write
  // maxNodeSize*2 value sizes. The
  // reason is: we want to be able to
  // merge full nodes in case of a
  // merge of two nodes with different
  // parents.
  const size_t maxNodeSize;
  // We merge a node with its sibling
  // if #keys <= fillFactor*maxNodeSize
  // 0 <= fillFactor < 1
  const double fillFactor;
};
```

## A.3 Helper Functions

We use some helper functions for repetitive tasks. These are listed here.

```
// error codes
constexpr int SUCCESS = 0;
constexpr int LOGICAL_PTR_DOESNT_EXIST = 1;
constexpr int EMPTY_TREE = 2;
constexpr int KEY_NOT_FOUND = 3;
constexpr int KEY_EXISTS = 4;
constexpr int CONFLICT = 5;

// A helper struct that basically holds
```

```
// the deserialized node and its versions
struct NodeValue {
  Node node;
  version_t ptrVersion;
  version_t nodeVersion;
  logical_ptr lptr;
  physical_ptr pptr;
};

// Get the tree node associated
// with the provided logical_ptr
int getNode(const BdTree& tree,
            NodeValue& result)
{
  // This function can starve. Lock-freeness does
  // not guarantee that all processes make progress.
  for (;;) {
    Value v;
    if (!storage.get(tree.pointerTable, result.lptr,
                     v)) {
      return LOGICAL_PTR_DOESNT_EXIST;
    }
    deserialize(result.pptr, v);
    result.ptrVersion = v.version;
    // the next get will not always succeed. It can
    // happen that another process rewrote the node
    // with an LL/SC operation and we read the old
    // physical pointer. This race is resolved by
    // simply reading the pointer again.
    if (storage.get(tree.nodeTable, result.pptr, v)) {
      deserialize(result.node, v);
      result.nodeVersion = v.version;
      return SUCCESS;
    }
  }
}

// Walks up the tree until the top node in the
// stack reflects the correct ranges from the
// parent and the key is in keyRange
//
// In the actual implementation this function (and
// code that calls into this function) is significantly
// more complex. The presented function will often fetch
// one node from storage unnecessarily which is
```

```
// inefficient. This is better optimized in the
// actual code.
void fixStack(const BdTree& tree,
              std::stack<NodeValue>& stack,
              std::pair<IndexKey, IndexKey> keyRange,
              const IndexKey& key)
{
  while (!stack.empty()) {
    // we fetch again the current node. A call to
    // fixStack indicates, that something is out
    // of date.
    auto& top = stack.top();
    NodeValue curr;
    curr.lptr = top.lptr;
    stack.pop();
    if (getNode(tree, curr) == SUCCESS) {
      if (auto leaf = boost::get<LeafNode>(&curr.node)) {
        if (leaf->smallestKey < key && key < leaf->largestKey) {
          stack.push(std::move(curr));
          // stack is clean (enough)
          keyRange.first = leaf->smallestKey;
          keyRange.second = leaf->largestKey;
          return;
        }
      } else if (auto inner = boost::get<InnerNode>(&curr.node)) {
        if (inner->smallestKey < key && key < inner->largestKey) {
          stack.push(std::move(curr));
          // stack is clean (enough)
          keyRange.first = inner->smallestKey;
          keyRange.second = inner->largestKey;
          return;
        }
      }
    }
  }
  // we poped everything from the stack or fixStack
  // was called with an empty stack. In this case we
  // fetch the root of the tree
  NodeValue root;
  root.lptr = 0;
  auto r = getNode(tree, root);
  if (r != SUCCESS) {
    return;
  }
  // We need to handle a corner case here:
```

```
  // It might happen that a root node split crashed
  // before it could write the new root node. In that
  // case we will finish it here
  if (auto p = boost::get<InnerNode>(&root.node)) {
    if (p->pointers.size() == 1) {
      // We need to finish a root node split here
      for (;;) {
        Value v;
        v.version = root.ptrVersion;
        serialize(root.pptr, v);
        if (storage.put(tree.pointerTable,
                        key_t(0), v)) {
          // SUCCESS
          storage.erase(tree.pointerTable,
                        key_t(root.pptr));
          // fetch root again
          getNode(tree, root);
          break;
        } else {
          // we'll retry
          if (getNode(tree, root) != SUCCESS) {
            assert(false);
          }
        }
      }
    }
  }
  stack.push(std::move(root));
  // The ranges in the root node might be wrong,
  // we therefore ignore them
  keyRange.first = IndexKey::min();
  keyRange.second = IndexKey::max();
}

// Walks up the tree until key is in the key
// range
void fixKeyRange(const BdTree& tree,
                 std::stack<NodeValue>& stack,
                 std::pair<IndexKey, IndexKey>& keyRange,
                 const IndexKey& key)
{
  for (;;) {
    if (stack.empty()) {
      fixStack(tree, stack, keyRange, key);
      break;
```

```
    }
    auto& node = stack.top();
    if (auto* inner = boost::get<InnerNode>(&node.node)) {
      if (inner->smallestKey < key || inner->largestKey >= key) {
        keyRange.first = inner->smallestKey;
        keyRange.second = inner->largestKey;
        break;
      }
      stack.pop();
    } else if (auto* leaf = boost::get<LeafNode>(&node.node)) {
      if (leaf->smallestKey < key || leaf->largestKey >= key) {
        keyRange.first = leaf->smallestKey;
        keyRange.second = leaf->largestKey;
        break;
      }
      stack.pop();
    } else {
      stack.pop();
    }
  }
}
```

## A.4 ID Generation

```
// we use the same function to generate physical
// and logical pointers. The reason for this is,
// that the IDs do not need to strictly increment,
// they only need to be unique
template<class T> // either physical_ptr or logical_ptr
T createId(const BdTree& tree) {
  key_t counter_key = (key_t) tree.nodeTable;
  Value lastId;
  serialize(uint64_t(1), lastId);
  auto get = [&]() {
    return storage.get(tree.counterTable, counter_key, lastId);
  };
  // on the first call, it might be that the counter does not
  // exist.
  while (!get()) {
    if (storage.put(tree.counterTable,
                    counter_key, lastId)) {
       break;
    }
  }
```

```
  T res;
  deserialize(res, lastId);
  for (;;) {
    serialize(res + 1, lastId);
    if (storage.put(tree.counterTable, counter_key, lastId)) {
      break;
    }
    get();
  }
  return res;
}
```

## A.5 Find

```
// lower_bound returns the value where its
// key is the largest key in the tree that
// is not smaller than the given key
//
// This function fills the provided stack with
// the path to a tree node which does contain
// the key-value pair if it exists (checking
// whether it does is the responsibility of
// the caller). This is usefull as we can use
// this function for other operations as well.
//
// The top element in stack will always be a
// LeafNode if lower_bound returns with SUCCESS
int lower_bound(const BdTree& tree,
                const IndexKey& key,
                std::stack<NodeValue>& stack)
{
  std::pair<IndexKey, IndexKey> keyRange(IndexKey::min(),
                                         IndexKey::max());
  if (stack.empty()) {
    // In a first step we get the root node.
    // fixStack either returns with something
    // in the stack or the tree itself is empty;
    fixStack(tree, stack, keyRange, key);
    if (stack.empty()) return EMPTY_TREE;
  } else {
    // we continue from a non-empty stack. Therefore we first need
    // to find the current keyRange
    // we do this by poping nodes from the stack until we find either
    // an inner or a leaf node. This is not very efficient but simple
    // and correct.
    for (;;) {
```

```
      auto& node = stack.top().node;
      if (auto* inner = boost::get<InnerNode>(&node)) {
        keyRange.first = inner->smallestKey;
        keyRange.second = inner->largestKey;
      } else if (auto* leaf = boost::get<LeafNode>(&node)) {
        keyRange.first = leaf->smallestKey;
        keyRange.second = leaf->largestKey;
      } else {
        stack.pop();
        continue;
      }
      break;
    }
  }
  for (;;) {
    auto& node = stack.top();
    if (auto* inner = boost::get<InnerNode>(&node.node)) {
      // we first check the range of the keys in this node.
      // If this range is not what we expect it to be, we will
      // retry
      // If this inner-node, however, is the root node, we expect
      // the range to be min -> max
      if (stack.size() == 1) {
        keyRange.first = IndexKey::min();
        keyRange.first = IndexKey::max();
      } else if (key < inner->smallestKey ||
                 keyRange.second != inner->largestKey) {
        fixStack(tree, stack, keyRange, key);
        continue;
      }
      keyRange.first = inner->smallestKey;
      // we hit an inner node
      auto iter = std::lower_bound(inner->pointers.begin(),
                                   inner->pointers.end(),
                                   key);
      // we know that this iterator will be valid, as it is in
      // the range of the node (we checked for this above)
      NodeValue next;
      next.lptr = iter->second;
      getNode(tree, next);
      stack.push(std::move(next));
      // the highest key in the new node has to be equal to
      // the key we found. Otherwise we will detect the race later
      keyRange.second = iter->first;
    } else if (auto* leaf = boost::get<LeafNode>(&node.node)) {
```

```
      if (stack.size() == 1) {
        keyRange.first = IndexKey::min();
        keyRange.first = IndexKey::max();
      } else if (key < inner->smallestKey ||
          keyRange.second != inner->largestKey) {
        fixStack(tree, stack, keyRange, key);
        continue;
      }
      keyRange.first = inner->smallestKey;
      return SUCCESS;
    } else if (boost::get<SplitNode>(&node.node)) {
      stack.push(std::move(node));
      split(tree, stack);
      fixKeyRange(tree, stack, keyRange, key);
    } else if (boost::get<DeletionNode>(&node.node)) {
      stack.push(std::move(node));
      merge(tree, stack);
      fixKeyRange(tree, stack, keyRange, key);
    } else if (boost::get<MergeNode>(&node.node)) {
      stack.push(std::move(node));
      merge(tree, stack);
      fixKeyRange(tree, stack, keyRange, key);
    }
  }
}

int lower_bound(const BdTree& tree,
                const IndexKey& key,
                key_t& result)
{
  std::stack<NodeValue> stack;
  auto res = lower_bound(tree, key, stack);
  if (res != SUCCESS) return res;
  auto& leaf = boost::get<LeafNode>(stack.top().node);
  auto iter = std::lower_bound(leaf.pointers.begin(),
                               leaf.pointers.end(),
                               key);
  if (iter == leaf.pointers.end()) {
    return KEY_NOT_FOUND;
  } else {
    result = iter->second;
    return SUCCESS;
  }
}
```

## A.6 Insert, Update and Delete

```
enum class Operation {
  INSERT, UPDATE, DELETE
};

// Executes a write operation on the Bd-Tree.
// For Operation::DELETE, value will be ignored
int write(const BdTree& tree,
          const IndexKey& key,
          std::stack<NodeValue>& stack,
          Operation op,
          key_t value)
{
  int res;
  for (;;) {
    res = lower_bound(tree, key, stack);
    if (res != SUCCESS) return res;
    // we copy the leaf to create a new one which we
    // can safely modify
    auto n = stack.top();
    auto leaf = boost::get<LeafNode>(n.node);
    switch (op) {
    case Operation::INSERT:
      if (leaf.pointers.size() >= tree.maxNodeSize) {
        // We can not insert into this node
        split(tree, stack);
        continue;
      }
    case Operation::DELETE:
      if (stack.size() > 1 &&
          leaf.pointers.size() <=
          tree.fillFactor * tree.maxNodeSize) {
        // We can not erase from this node, as it would
        // violate the size invariant
        merge(tree, stack);
        continue;
      }
    case Operation::UPDATE:
      // we don't need to do anything in this case
      break;
    }
    // find position for operation
    auto iter = std::lower_bound(leaf.pointers.begin(),
                                 leaf.pointers.end(),
```

```
                                key);
  if (iter == leaf.pointers.end()) {
    switch (op) {
    case Operation::INSERT:
      // This key is smaller than the smallest key in the
      // nodes array. Therefore we will it as the first
      // element in the array
      leaf.pointers.emplace(leaf.pointers.begin(), key, value);
      break;
    case Operation::DELETE:
      return KEY_NOT_FOUND;
    case Operation::UPDATE:
      return KEY_NOT_FOUND;
    }
  } else if (iter->first == key) {
    switch (op) {
    case Operation::INSERT:
      return KEY_EXISTS;
    case Operation::DELETE:
      leaf.pointers.erase(iter);
      break;
    case Operation::UPDATE:
      iter->second = value;
    }
  } else {
    switch (op) {
    case Operation::INSERT:
      // std::vector::emplace inserts the element before
      // the iterator
      ++iter;
      leaf.pointers.emplace(iter, key, value);
    case Operation::DELETE:
      return KEY_NOT_FOUND;
    case Operation::UPDATE:
      return KEY_NOT_FOUND;
    }
  }
  // try to write back the new node

  // we plan to remove the current leaf
  stack.pop();
  // in a first step, we write the new leaf node
  // to storage. This will never fail
  Value val;
  serialize(Node(std::move(leaf)), val);
```

```
    auto pptr = createId<physical_ptr>(tree);
    // val.version is already infinity
    // this put operation cannot fail, as the
    // generated Id is guarnteed to be unique.
    storage.put(tree.nodeTable,
                key_t(pptr),
                val);
    // now we do the SC part of the LL/SC operation
    // if this fails, the tree changed and we retry
    serialize(pptr, val);
    val.version = n.ptrVersion;
    if (storage.put(tree.pointerTable,
                    key_t(n.lptr),
                    val)) {
      // we can now safely erase the old node from storage
      storage.erase(tree.nodeTable, key_t(n.pptr));
      return SUCCESS;
    }
    // we pickup our trash before we continue
    storage.erase(tree.nodeTable, key_t(pptr));
  }
}

int insert(const BdTree& tree,
           const IndexKey& key,
           key_t value)
{
  std::stack<NodeValue> stack;
  return write(tree, key, stack,
               Operation::INSERT, value);
}

int update(const BdTree& tree,
           const IndexKey& key,
           key_t value)
{
  std::stack<NodeValue> stack;
  return write(tree, key, stack,
               Operation::UPDATE, value);
}

int erase(const BdTree& tree,
          const IndexKey& key)
{
  std::stack<NodeValue> stack;
```

```
  // the value will be ignored, therefore we can pass
  // anything we want
  return write(tree, key, stack,
               Operation::DELETE, key_t(0));
}
```

## A.7 Split

```
// We use a visitor that executes the
// first step of the split operation:
// - if the node is a leaf or an inner
//   node, it will create the split node
//   and node Q
// - Otherwise, it will fetch P and Q
struct GenericSplitter : boost::static_visitor<>
{
  const BdTree& tree;
  std::stack<NodeValue>& stack;
  NodeValue initial;
  NodeValue P;
  NodeValue Q;
  NodeValue splitNode;

  GenericSplitter(const BdTree& tree,
                  std::stack<NodeValue>& stack,
                  const NodeValue& initial)
    : tree(tree)
    , stack(stack)
    , initial(initial)
    {}

  template<class T>
  void startSplit(T& node) {
    // step 1: insert new node
    // This step will never fail
    P = initial;

    // we create a split node and node Q
    T qN;
    auto iter = node.pointers.begin();
    std::advance(iter, node.pointers.size() / 2);
    qN.largestKey = node.largestKey;
    qN.smallestKey = iter->first;
    std::advance(iter, 1);
    // Now we fill the new Node
    std::copy(iter, node.pointers.end(),
```

```
            std::back_insert_iterator<typename T::ContainerType>(
              qN.pointers));
  qN.logicalNextPointer = node.logicalNextPointer;
  Q.node = qN;
  Q.pptr = createId<logical_ptr>(tree);
  Value v;
  serialize(Q.node, v);
  // we don't need to check the result of this put, as
  // we know that this will succeed.
  storage.put(tree.nodeTable, key_t(Q.pptr), v);
  Q.nodeVersion = v.version;
  Q.lptr = createId<logical_ptr>(tree);
  v.version = infinity;
  serialize(Q.pptr, v);
  storage.put(tree.pointerTable, key_t(Q.lptr), v);
  Q.ptrVersion = v.version;
  // now, the split node needs to be created
  SplitNode split;
  split.left = P.pptr;
  split.right = Q.lptr;
  splitNode.node = split;
  splitNode.pptr = createId<physical_ptr>(tree);
  v.version = infinity;
  serialize(split, v);
  storage.put(tree.nodeTable, key_t(splitNode.pptr), v);
  splitNode.ptrVersion = v.version;

  // now, all nodes are created and written to storage
  if (!cas()) {
    // if the CAS operation fails, we will just pick
    // up the trash and exit. The caller will retry if
    // necessary
    storage.erase(tree.nodeTable, key_t(Q.pptr));
    storage.erase(tree.pointerTable, key_t(Q.lptr));
    storage.erase(tree.nodeTable, key_t(splitNode.pptr));
    return;
  }
  // From here on, the split is guaranteed to finish eventually
  if (!updateParent()) {
    return;
  }
  consolidateP();
}

const IndexKey& qMin() {
```

```
    if (auto q = boost::get<InnerNode>(&Q.node)) {
      return q->smallestKey;
    }
    return boost::get<LeafNode>(Q.node).smallestKey;
  }

  const IndexKey& qMax() {
    if (auto q = boost::get<InnerNode>(&Q.node)) {
      return q->largestKey;
    }
    return boost::get<LeafNode>(Q.node).largestKey;
  }

  // in the real implementation, this function does some
  // complicated dancing through the tree to minimize
  // retries. This function is much more optimistic:
  // if anything fails, it will just fail.
  // The calling operation will then retry its operation
  // again and, possibly, retry to do a split.
  bool cas() {
    auto parentN = stack.top();
    auto parent = boost::get<InnerNode>(&parentN.node);
    if (parent == nullptr) {
      // there already is a split or merge operation
      return false;
    }
    const auto& splitKey = qMax();
    auto iter = std::lower_bound(parent->pointers.begin(),
                                 parent->pointers.end(),
                                 splitKey);
    if (iter == parent->pointers.end() ||
        iter->first != splitKey) {
      // in this case something is horribly out
      // of date - so we abort and let the client
      // retry
      // before we do so, we make sure that the top
      // of our stack is up to date
      fixStack(tree, stack,
               std::make_pair(parent->smallestKey,
                              parent->largestKey),
               splitKey);
      return false;
    }
    // now we do the compare and swap
    // this is actually a compare and swap
```

```
    // but we do not have the ABA problem,
    // as each Id is generated at most once.
    Value v;
    if (!storage.get(tree.pointerTable,
                     iter->second,
                     v)) {
      // again, something is out of date
      return false;
    }
    physical_ptr currPtr;
    deserialize(currPtr, v);
    if (currPtr == splitNode.pptr) {
      // The split was done already...
      // This function can be called from
      // within several stages.
      return true;
    }
    if (currPtr != P.pptr) {
      // CONFLICT
      return false;
    }
    serialize(splitNode.pptr, v);
    if (!storage.put(tree.pointerTable,
                     iter->second,
                     v)) {
      // CONFLICT
      return false;
    }
    return true;
  }

  bool updateParent() {
    // we know that the parent is an inner node
    // this statement copies it
    auto oldParent = stack.top();
    stack.pop();
    auto inner = boost::get<InnerNode>(oldParent.node);
    auto q = qMax();
    auto iter = std::lower_bound(
      inner.pointers.begin(),
      inner.pointers.end(),
      q);
    // we used this node to walk to P, therefore
    // this entry has to exist
    assert(iter == inner.pointers.end());
```

```
    assert(iter->first != q);
    iter->second = Q.lptr;
    // now we insert the new P
    auto p = qMin(); // qMin() == pMax()
    // we insert the new P before Q
    inner.pointers.insert(
      iter,
      std::make_pair(
        p, splitNode.lptr));
    NodeValue newParent;
    newParent.lptr = oldParent.lptr;
    newParent.pptr = createId<physical_ptr>(tree);
    newParent.node = std::move(inner);
    // write node to storage
    Value v;
    serialize(inner, v);
    storage.put(tree.nodeTable,
                key_t(newParent.pptr), v);
    // do the conditional write
    v.version = oldParent.ptrVersion;
    serialize(newParent.pptr, v);
    if (storage.put(
          tree.pointerTable,
          key_t(newParent.lptr), v)) {
      // SUCCESS
      // remove old parent
      storage.erase(tree.nodeTable,
                    key_t(oldParent.pptr));
      return true;
    } else {
      // conflict
      storage.erase(tree.nodeTable,
                    key_t(newParent.pptr));
      return false;
    }
  }


  template<class T>
  void consolidatePT(T node) { // node is a copy
    // first, we write a new version of P

    // delete all entries which belong to Q
    node.largestKey = boost::get<T>(Q.node).smallestKey;
    node.pointers.erase(
```

```
      std::upper_bound(
        node.pointers.begin(),
        node.pointers.end(),
        node.largestKey),
      node.pointers.end());
    node.logicalNextPointer = Q.lptr;
    auto pptr = createId<physical_ptr>(tree);
    Value v;
    serialize(node, v);
    // this can never fail
    storage.put(tree.nodeTable,
                key_t(pptr), v);
    // then we try to install this update
    v.version = P.ptrVersion;
    serialize(pptr, v);
    if (!storage.put(tree.pointerTable,
                     key_t(P.lptr),
                     v)) {
      // take the trash and leave
      // someone will retry
      storage.erase(tree.nodeTable, key_t(pptr));
    } else {
      // success, now we need to delete the old delta
      // and the old P node
      storage.erase(tree.nodeTable, key_t(splitNode.pptr));
      storage.erase(tree.nodeTable, key_t(P.pptr));
    }
  }

  // if this method fails, it usually means that
  // another process already did the consolidation.
  // In the rarer case (usually concurrent update
  // operations) we pay the cost to restore the
  // state and retry
  void consolidateP() {
    if (auto node = boost::get<InnerNode>(&P.node)) {
      consolidatePT(*node);
    } else {
      consolidatePT(boost::get<LeafNode>(P.node));
    }
  }

  void operator() (LeafNode& leaf) {
    startSplit(leaf);
  }
```

```
void operator() (InnerNode& inner) {
  startSplit(inner);
}

void operator() (SplitNode& split) {
  splitNode = initial;
  Q.lptr = split.right;
  int res = getNode(tree, Q);
  if (res != SUCCESS) {
    // something changed already
    stack.pop();
    return;
  }
  auto parent = boost::get<InnerNode>(&stack.top().node);
  if (!parent) {
    // we hold old state
    return;
  }
  auto iter = std::lower_bound(
    parent->pointers.begin(),
    parent->pointers.end(),
    qMin());
  if (iter->first == qMin()) {
    // someone already installed
    // the split and the Q
    P.lptr = iter->second;
    getNode(tree, P);
    consolidateP();
  } else {
    P.lptr = iter->second;
    getNode(tree, P);
    if (updateParent()) consolidateP();
  }
}

void operator() (DeletionNode&) {
  // must never happen
  assert(false);
}

void operator() (MergeNode&) {
  // must never happen
  assert(false);
}
```

```
};

// Splits two nodes (either inner)
// or outer and writes the results
// to left and right.
template<class T>
void execSplit(const T& toSplit,
               NodeValue& left,
               NodeValue& right,
               InnerNode& root)
{
  T l;
  T r;
  auto iter = toSplit.pointers.begin();
  std::advance(iter, toSplit.pointers.size() / 2 + 1);
  std::copy(
    toSplit.pointers.begin(),
    iter,
    std::back_insert_iterator<typename T::ContainerType>(l.pointers));
  std::copy(
    iter,
    toSplit.pointers.end(),
    std::back_insert_iterator<typename T::ContainerType>(l.pointers));
  // The ranges in a root key are allowed to be wrong.
  // We ignore them
  l.smallestKey = IndexKey::min();
  l.largestKey = l.pointers.rbegin()->first;
  r.smallestKey = l.largestKey;
  r.largestKey = IndexKey::max();
  l.logicalNextPointer = right.lptr;
  r.logicalNextPointer = invalid;

  root.pointers.push_back(
    std::make_pair(l.largestKey, left.lptr));
  root.pointers.push_back(
    std::make_pair(r.largestKey, right.lptr));

  left.node = std::move(l);
  right.node = std::move(r);
}

// here we handle the special case of a root
// node split. Instead of creating a split
// node, we just replace the current root
// with the finished tree.
```

```
void rootSplit(const BdTree& tree,
               std::stack<NodeValue>& stack)
{
  assert(stack.size() == 1);

  // First step: Create new root node
  // and the two children
  auto& oldRoot = stack.top();
  stack.pop();
  NodeValue newRoot;
  newRoot.lptr = 0;
  newRoot.pptr = createId<physical_ptr>(tree);
  InnerNode root;
  NodeValue leftChild;
  NodeValue rightChild;
  leftChild.pptr = createId<physical_ptr>(tree);
  leftChild.lptr = createId<logical_ptr>(tree);
  rightChild.pptr = createId<physical_ptr>(tree);
  rightChild.lptr = createId<logical_ptr>(tree);
  if (auto leaf = boost::get<LeafNode>(&oldRoot.node)) {
    execSplit(*leaf, leftChild, rightChild, root);
  } else if (auto inner = boost::get<InnerNode>(&oldRoot.node)) {
    execSplit(*inner, leftChild, rightChild, root);
  }
  newRoot.node = root;

  // Second step: write back new tree
  // write nodes
  Value v;
  serialize(leftChild.node, v);
  storage.put(tree.nodeTable,
              key_t(leftChild.pptr),
              v);
  v.version = infinity;

  serialize(rightChild.node, v);
  storage.put(tree.nodeTable,
              key_t(rightChild.pptr),
              v);
  v.version = infinity;

  serialize(newRoot.node, v);
  storage.put(tree.nodeTable,
              key_t(newRoot.pptr),
              v);
```

```
  v.version = infinity;

  serialize(leftChild.pptr, v);
  storage.put(tree.pointerTable,
              key_t(leftChild.lptr),
              v);
  v.version = infinity;

  serialize(rightChild.pptr, v);
  storage.put(tree.pointerTable,
              key_t(rightChild.lptr),
              v);
  v.version = infinity;

  // last step: do the compare and swap
  serialize(newRoot.pptr, v);
  v.version = oldRoot.ptrVersion;
  if (storage.put(tree.pointerTable,
                  key_t(newRoot.lptr),
                  v)) {
    // Split succeeded
    newRoot.ptrVersion = v.version;
    stack.push(newRoot);
  } else {
    // Split failed. Collect garbage
    storage.erase(tree.nodeTable,
                  key_t(leftChild.pptr));
    storage.erase(tree.nodeTable,
                  key_t(rightChild.pptr));
    storage.erase(tree.nodeTable,
                  key_t(newRoot.pptr));
    storage.erase(tree.pointerTable,
                  key_t(leftChild.lptr));
    storage.erase(tree.pointerTable,
                  key_t(rightChild.lptr));
  }
}


// Tries to execute a split on the top node
// of the stack. This function will fail
// silently, as the caller will automatically
// retry if something fails and the node was
// not splitted by another process.
//
```

```
// To help the reader, we are using the same
// names for nodes as in the textual description:
//   - P is the node that should be splitted
//   - R is the initial right sibling of P
//   - Q is the newly inserted node
void split(const BdTree& tree,
           std::stack<NodeValue>& stack)
{
  // this function can be called with a
  // split node, with a inner node, or
  // with a leaf node...

  if (stack.size() == 1) {
    rootSplit(tree, stack);
    return;
  }

  GenericSplitter splitter(tree, stack, stack.top());
  stack.pop();
  boost::apply_visitor(splitter, splitter.initial.node);
}
```

## A.8 Merge

```
struct GenericMerger : boost::static_visitor<>
{
  const BdTree& tree;
  std::stack<NodeValue> stack;
  NodeValue initial, P, Q, delNode, mergeNode, parent;

  GenericMerger(const BdTree& tree,
                std::stack<NodeValue>& stack,
                const NodeValue& initial)
    : tree(tree)
    , stack(stack)
    , initial(initial)
    { assert(stack.size() > 1); }

  bool installDelNode() {
    DeletionNode node;
    node.toDelete = Q.pptr;
    delNode.lptr = Q.lptr;
    delNode.pptr = createId<physical_ptr>(tree);

    // write the deletion node
    Value v;
```

```
    serialize(node, v);
    delNode.node = std::move(node);
    storage.put(tree.nodeTable,
                key_t(delNode.pptr), v);

    // Try to install the deletion node
    v.version = Q.ptrVersion;
    serialize(delNode.pptr, v);
    if (storage.put(
          tree.pointerTable,
          key_t(Q.lptr),
          v)) {
      delNode.ptrVersion = v.version;
      return true;
    }
    // The conditional store failed.
    // Collect garbage
    storage.erase(tree.nodeTable,
                  key_t(delNode.pptr));
    storage.erase(tree.pointerTable,
                  key_t(Q.lptr));
    return false;
  }

  bool getLeftSibling(NodeValue& result,
                      const NodeValue& of,
                      const NodeValue& parent) {
    IndexKey index;
    if (auto node = boost::get<InnerNode>(&of.node)) {
      index = node->largestKey;
    } else {
      index = boost::get<InnerNode>(of.node).largestKey;
    }
    const InnerNode& p = boost::get<InnerNode>(parent.node);
    auto iter = std::lower_bound(p.pointers.begin(),
                                 p.pointers.end(), index);
    // iter now points to the value containing "of"
    if (iter == p.pointers.begin()) {
      // nodes to merge point to different
      // parents. We first merge them and then retry
      merge(tree, stack);
      return false;
    }
    --iter;
    return getNode(tree, result) == SUCCESS;
```

```
}

// get a node with only its physical
// pointer defined.
bool getNodeP(NodeValue& result) {
  Value v;
  return storage.get(tree.nodeTable,
                     key_t(result.pptr), v);
}

bool installMergeNode() {
  stack.pop();
  parent = stack.top();
  NodeValue l;
  if (!getLeftSibling(l, Q, parent)) {
    // parent node is out of date
    stack.pop();
    return false;
  }
  if (auto m = boost::get<MergeNode>(&l.node)) {
    // someone else already installed the
    // merge node
    if (m->right != Q.pptr) {
      // Something is out of date
      return false;
    }
    mergeNode = l;
    P.pptr = m->left;
    return getNodeP(P);
  }
  MergeNode n;
  n.left = P.pptr;
  n.right = Q.pptr;
  mergeNode.pptr = createId<physical_ptr>(tree);

  // write mergeNode to storage
  Value v;
  serialize(n, v);
  storage.put(tree.nodeTable,
              key_t(mergeNode.pptr), v);
  mergeNode.node = std::move(n);
  mergeNode.lptr = P.lptr;
  // Conditional write
  serialize(mergeNode.pptr, v);
  v.version = P.ptrVersion;
```

```
    if (storage.put(
          tree.pointerTable,
          key_t(mergeNode.lptr), v)) {
      mergeNode.ptrVersion = v.version;
      return true;
    }
    // CONFLICT
    storage.erase(tree.nodeTable,
                  key_t(mergeNode.pptr));
    return false;
  }

  template<class T>
  bool getNodeAt(NodeValue res,
                 const T& node,
                 const IndexKey& idx) {
  }

  template<class AssertedType, class T>
  bool getNextNode(NodeValue& result,
                   const T& node) {
    result.lptr = node.logicalNextPointer;
    if (getNode(tree, result) == SUCCESS) {
      if (boost::get<AssertedType>(&result.node))
        return true;
    }
    return true;
  }

  bool findDelNode() {
    // we set the parent
    stack.pop();
    parent = stack.top();
    // We can not be sure that the parent still
    // points to the deletion node (it might be
    // that the parent was already updated).
    // Therefore we walk to it through P.
    if (auto p = boost::get<LeafNode>(&P.node)) {
      return getNextNode<DeletionNode>(delNode, *p);
    } else {
      return getNextNode<DeletionNode>(
        delNode,
        boost::get<InnerNode>(P.node));
    }
  }
```

```
bool updateParent() {
  // when we reach this step, we know that
  // all previous step succeeded and that
  // P, Q, delNode, mergeNode, and parent
  // are all set.
  IndexKey qIdx;
  if (auto q = boost::get<LeafNode>(&Q.node)) {
    qIdx = q->largestKey;
  } else {
    qIdx = boost::get<LeafNode>(Q.node).largestKey;
  }
  auto p = boost::get<InnerNode>(parent.node);
  auto iter = std::lower_bound(
    p.pointers.begin(),
    p.pointers.end(), qIdx);
  if (iter == p.pointers.end() ||
      iter->first != qIdx) {
    // This step was already executed
    return true;
  }
  // we know that the current parent will be out of
  // date when this function returns
  stack.pop();
  p.pointers.erase(iter);
  NodeValue newParent;
  newParent.lptr = parent.lptr;
  newParent.pptr = createId<physical_ptr>(tree);
  newParent.node = std::move(p);

  // write back node
  Value v;
  serialize(newParent.node, v);
  storage.put(tree.nodeTable,
              key_t(newParent.pptr),
              v);
  // conditional write
  v.version = parent.ptrVersion;
  serialize(newParent.pptr, v);
  if (storage.put(
        tree.pointerTable,
        key_t(newParent.lptr), v)) {
    stack.push(newParent);
    parent = std::move(newParent);
    return true;
```

```
    }
    // collect garbage
    storage.erase(tree.nodeTable,
                  key_t(newParent.pptr));
    return false;
  }

  template<class T>
  void mergeNodes(NodeValue& res,
                  const T& left,
                  const T& right) {
    T r;
    std::copy(
      left.pointers.begin(),
      left.pointers.end(),
      std::back_insert_iterator<typename T::ContainerType>(
        r.pointers));
    std::copy(
      left.pointers.begin(),
      left.pointers.end(),
      std::back_insert_iterator<typename T::ContainerType>(
        r.pointers));
    r.smallestKey = left.smallestKey;
    r.largestKey = right.largestKey;
    r.logicalNextPointer = right.logicalNextPointer;
    res.node = r;
  }

  bool isRootMerge() {
    if (stack.size() != 1) return false;
    auto p = boost::get<InnerNode>(parent.node);
    if (p.pointers.size() > 1) return false;
    return true;
  }

  void consolidate() {
    // now we can write the new P
    NodeValue newP;
    if (auto p = boost::get<LeafNode>(&P.node)) {
      mergeNodes(newP, *p, boost::get<LeafNode>(Q.node));
    } else {
      mergeNodes(newP,
                 boost::get<InnerNode>(P.node),
                 boost::get<InnerNode>(Q.node));
    }
```

```
    newP.pptr = createId<physical_ptr>(tree);
    newP.lptr = P.lptr;
    // write back consolidated node
    Value v;
    serialize(newP.node, v);
    storage.put(tree.nodeTable,
                key_t(newP.pptr), v);
    // conditional write
    v.version = P.ptrVersion;
    serialize(newP.pptr, v);
    if (!storage.put(tree.pointerTable,
                     key_t(newP.lptr),
                     v)) {
      // we had a conflict
      storage.erase(tree.nodeTable,
                     key_t(newP.pptr));
    }
    P = newP;
    if (isRootMerge()) {
      // in this case, we can now try to replace the
      // old root with P
      v.version = parent.ptrVersion;
      serialize(P.pptr, v);
      bool s = storage.put(
        tree.pointerTable,
        key_t(0), v);
      if (s) {
        storage.erase(
          tree.pointerTable, key_t(P.lptr));
        stack.pop();
      }
    }
    // The split did successfully complete
    stack.push(P);
  }

  void operator() (LeafNode& leaf) {
    Q = initial;
    if (!installDelNode()) {
      stack.pop();
      return;
    }
    if (!installMergeNode()) {
      stack.pop();
      return;
```

```
    }
    if (!updateParent()) return;
    consolidate();
  }

  void operator() (InnerNode& inner) {
    Q = initial;
    if (!installDelNode()) {
      stack.pop();
      return;
    }
    if (!installMergeNode()) {
      stack.pop();
      return;
    }
    if (!updateParent()) return;
    consolidate();
  }

  void operator() (DeletionNode& del) {
    delNode = initial;
    // get Node Q
    Q.pptr = del.toDelete;
    if (!getNodeP(Q)) {
      // something is out of date
      // remove current node and
      // parent to force re-read
      // from storage
      stack.pop();
      stack.pop();
      return;
    }
    if (!installMergeNode()) {
      return;
    }
    if (!updateParent()) return;
    consolidate();
  }

  void operator() (MergeNode& m) {
    mergeNode = initial;
    P.pptr = m.left;
    Q.pptr = m.right;
    if (!getNodeP(Q)) {
      stack.pop();
```

```
      stack.pop();
      return;
    }
    if (!getNodeP(P)) {
      stack.pop();
      stack.pop();
      return;
    }
    if (!findDelNode()) {
      stack.pop();
      return;
    }
    if (!updateParent()) return;
    consolidate();
  }

  void operator() (SplitNode&) {
    assert(false);
  }
};

void merge(const BdTree& tree,
           std::stack<NodeValue>& stack)
{
  GenericMerger merger(tree, stack, stack.top());
  stack.pop();
  boost::apply_visitor(merger, merger.initial.node);
}
```

# B

# Transaction Processing

This chapter lists a simple transaction class in C++-like pseudo code. While it is a strong simplification of the one in TellDB it is still useful as a reference.

The pseudo-code shows a simpler (and, unlike the original, blocking) interface for storage and commit manager. It then presents a transaction class that can be used by clients to run transactions against TellStore.

The main components missing from this code are range queries and scans. Range queries and scan both need a log of code which has to do bookkeeping of data from the cache and data fetched from storage. However, this code is not very interesting and writing it should be an easy exercise.

```
#include <unordered_map>
#include <cstdint>
#include <memory>
#include <boost/functional/hash.hpp>

namespace telldb {

// This class defines an interface of the Snapshot Descriptor
// defined in the thesis. The interface itself is not shown
// here, as the client will be oblivious to the actual
// interface.
class Snapshot {
public:
  uint64_t version();
  // rest not shown
```

```
};

// The three transaction classes supported by Tell
enum class TransactionMode { READ_WRITE, READ_ONLY, ANALYTICAL };

class CommitManager {
public:
  Snapshot getNewSnapshot(TransactionMode mode);
  void finishTransaction(Snapshot &snapshot);
  // The commit manager assignes an undo table to each
  // processing node. This table is passed to another
  // processing node in case of failures and can be used
  // to rollback all in-flight transactions.
  uint64_t getMachineId() const;
};

// TellStore can hold a schema. The implementation
// of such a row is not shown here, to keep the code
// simple
class RowType {};

// TellStore only allows uint64_t as primary keys. All
// other indexes are implemented via Bd-tree (not shown
// here)
using table_t = uint64_t;
using key_t = uint64_t;

class Conflict : std::exception {};
class KeyDoesNotExist : std::exception {};

// The interface to TellStore. The actual interface is
// more complex than this one.
struct TellStore {
  bool get(Snapshot, table_t, key_t, RowType &, bool &isNewest);
  bool put(Snapshot, table_t, key_t, const RowType &);
  bool put(Snapshot, table_t, key_t, const char*, size_t);
  bool erase(Snapshot, table_t, key_t);
  // Reverts the operation executed by the transaction with the
  // given version. This can also be safely called if the transaction
  // did not write anything to the provided key.
  // Returns true if something was reverted
  bool revert(uint64_t version, table_t, key_t);
};

// serializes data into a buffer. resizes buffer if it is too small and
```

```
// resets the size.
template<class T>
void serialize(const T& value, std::unique_ptr<char[]>& buffer, size_t&
    size);

class Transaction {
  enum class Operation { READ, UPDATE, INSERT, DELETE };
  struct CacheEntry {
    RowType row;
    // indicates whether this tuple was the newest version
    // at the time it was read from storage
    bool isNewest = true;
    Operation op = Operation::READ;
  };

  CommitManager &commitManager;
  TellStore &storage;
  TransactionMode mode;
  Snapshot snapshot;
  bool finished = false;
  // The transaction cache.
  // The actual implementation also caches updated index entries
  using global_key = std::pair<table_t, key_t>;
  using hash = boost::hash<global_key>;
  std::unordered_map<global_key, CacheEntry, hash> cache;

public:
  Transaction(CommitManager &commitManager, TellStore &storage,
              TransactionMode mode = TransactionMode::READ_WRITE)
      : commitManager(commitManager), storage(storage), mode(mode),
        snapshot(commitManager.getNewSnapshot(mode)) {}

  ~Transaction() {
    // We simply abort a transaction that gets destroyed
    // before it was commited or aborted by the user.
    if (!finished) {
      abort();
    }
  }

  void abort() {
    // aborting is as simple as giving the snapshot
    // back to the commit manager. When this method
    // is called, no writes to storage were executed
    commitManager.finishTransaction(snapshot);
```

```
    finished = true;
  }

  void commit() {
    // In a first step, we serialize the undo log
    std::unique_ptr<char[]> buf;
    size_t sz = 0;
    // we write the key of each tuple we are planning
    // to touch into the undo log. This together with
    // the version of the transaction is enough to
    // roll back a transaction
    for (auto& p : cache) {
      if (p.second.op == Operation::READ) continue;
      serialize(p.first, buf, sz);
    }
    // Next we would write all Bd-Tree updates into the
    // undo log. However, this part is omitted for
    // brevity.
    //
    // Then we simply write it into the table with the
    // same id as our process node id
    storage.put(snapshot,
                table_t(commitManager.getMachineId()),
                key_t(snapshot.version()),
                buf.get(), sz);
    // Now we can try to commit
    bool conflict = false;
    for (auto& p : cache) {
      switch (p.second.op) {
      case Operation::READ:
        continue;
      case Operation::DELETE:
        conflict = storage.erase(
          snapshot,
          p.first.first, p.first.second);
        break;
      case Operation::UPDATE:
      case Operation::INSERT:
        conflict = storage.put(
          snapshot, p.first.first, p.first.second, p.second.row);
        break;
      }
      if (conflict) {
        // TellStore reported a conflict. We need to roll back
        break;
```

```
    }
  }
  if (!conflict) return;
  // rollback
  for (auto& p : cache) {
    if (p.second.op == Operation::READ) continue;
    if (!storage.revert(snapshot.version(),
                        p.first.first, p.first.second)) {
      // The order of the unordered_map does not change
      // if nothing is written to it. Therefore we can simply
      // rollback changes in the same order which we used to
      // apply them. As soon as a revert fails, we know that the
      // corresponding tuple was not written by this transaction.
      // Therefore no other tuple was written.
      break;
    }
  }
  // Next we would roll back the Bd-tree changes
  throw Conflict();
}

const RowType &read(table_t table, key_t key) {
  global_key k = {table, key};
  auto iter = cache.find(k);
  if (iter != cache.end()) {
    if (iter->second.op == Operation::DELETE)
      throw KeyDoesNotExist();
    return iter->second.row;
  }
  CacheEntry c;
  if (!storage.get(snapshot, table, key, c.row, c.isNewest)) {
    throw KeyDoesNotExist();
  }
  CacheEntry &res = cache[k];
  res = std::move(c);
  return res.row;
}

void write(table_t table, key_t key, const RowType &row) {
  global_key k = {table, key};
  // we first make sure that the row is in cache if
  // it exists on storage
  try {
    read(table, key);
  } catch (KeyDoesNotExist&) {}
```

```
    auto p = cache.emplace(k, CacheEntry());
    auto iter = p.first;
    if (!iter->second.isNewest)
      throw Conflict();
    if (p.second || iter->second.op == Operation::INSERT) {
      // this is an insert operation
      iter->second.op = Operation::INSERT;
    } else {
      iter->second.op = Operation::UPDATE;
    }
    iter->second.row = row;
  }

  void erase(table_t table, key_t key) {
    global_key k = {table, key};
    auto iter = cache.find(k);
    if (iter == cache.end()) {
      read(table, key);
      iter = cache.find(k);
    }
    if (!iter->second.isNewest) throw Conflict();
    switch (iter->second.op) {
    case Operation::INSERT:
      cache.erase(iter);
      break;
    case Operation::READ:
    case Operation::UPDATE:
      iter->second.op = Operation::DELETE;
      break;
    case Operation::DELETE:
      throw KeyDoesNotExist();
    }
  }
};
}
```

# List of Tables

# List of Figures

# Bibliography


[ABH09] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. “Column-oriented Database Systems”. In: *Proceedings of the VLDB Endowment* 2.2 (2009), pages 1664–1665 (cited on page 47).

[AMH08] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. “Column-Stores vs. Row-Stores: How Different Are They Really?” In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pages 967–980 (cited on page 47).

[Ail+01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. “Weaving Relations for Cache Performance”. In: *VLDB*. 2001, pages 169–180 (cited on pages 47, 57).

[BG81] Philip A Bernstein and Nathan Goodman. “Concurrency Control in Distributed Database Systems”. In: *ACM Computing Surveys (CSUR)* 13.2 (1981), pages 185–221 (cited on page 113).

[BRD11] Philip A Bernstein, Colin W Reid, and Sudipto Das. “Hyder - A Transactional Record Manager for Shared Flash”. In: *CIDR*. Volume 11. 2011, pages 9–20 (cited on page 25).

[BMK99] Peter A Boncz, Stefan Manegold, and Martin L Kersten. “Database Architecture Optimized for the new Bottleneck: Memory Access”. In: *VLDB*. Volume 99. 1999, pages 54–65 (cited on page 47).

[Bon+06] Peter Boncz, Torsten Grust, Maurice Van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. “MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine”. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM. 2006, pages 479–490 (cited on pages 47, 58).

[Bra+15] Lucas Braun, Thomas Etter, Georgios Gasparis, Martin Kaufmann, Donald Kossmann, Daniel Widmer, Aharon Avitzur, Anthony Iliopoulos, Eliezer Levy, and Ning Liang. “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, pages 251–264. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742783. URL: http://doi.acm.org/10.1145/2723372.2742783 (cited on pages 16, 57, 58, 140).

[CRF08] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. “Serializable Isolation for Snapshot Databases”. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’08. Vancouver, Canada: ACM, 2008, pages 729–738. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376690. URL: http://doi.acm.org/10.1145/1376616.1376690 (cited on page 114).

[16a] *Cassandra Datastax Client*. https://github.com/ datastax/java-driver. May. 02. 2016 (cited on page 73).

[CO15] Tatsuhiro Chiba and Tamiya Onodera. “Workload Characterization and Optimization of TPC-H Queries on Apache Spark”. In: (2015) (cited on page 124).

[Coo+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking Cloud Serving Systems with YCSB”. In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC ’10. Indianapolis, Indiana, USA: ACM, 2010, pages 143–154. ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152. URL: http://doi.acm.org/10.1145/1807128.1807152 (cited on page 74).

[Cou16] Transaction Processing Council. *TPC-H*. http://www.tpc.org/tpch. May. 02. 2016 (cited on page 75).

[DG08] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51.1 (2008), pages 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492 (cited on page 22).

[Dia+13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. “Hekaton: SQL Server’s Memory-Optimized OLTP Engine”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pages 1243–1254 (cited on page 90).

[Fac16] Facebook. *Presto | Distributed SQL Query Engine for Big Data*. 2016. URL: https://prestodb.io/ (visited on 02/02/2016) (cited on pages 33, 123).

[Fär12] Franz Färber et al. “The SAP HANA Database – An Architecture Overview”. In: *IEEE Data Eng. Bull.* 35.1 (2012) (cited on page 58).

[Fär+12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. “SAP HANA Database: Data Management for Modern Business Applications”. In: *SIGMOD Rec.* 40.4 (2012), pages 45–51. ISSN: 0163-5808. DOI: 10.1145/2094114.2094126. URL: http://doi.acm.org/10.1145/2094114.2094126 (cited on page 46).

[Fou13] Foundationdb. *FoundationDB – a NoSQL Database with ACID Transactions*. 2013. URL: http://archive.is/NS07L (visited on 04/10/2013) (cited on page 33).

[Geo11] Lars George. *HBase: The Definitive Guide*. ” O’Reilly Media, Inc.”, 2011 (cited on page 41).

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google File System”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP ’03. Bolton Landing, NY, USA: ACM, 2003, pages 29–43. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945450. URL: http://doi.acm.org/10.1145/945445.945450 (cited on page 22).

[GAK12] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. “SharedDB: Killing One Thousand Queries with One Stone”. In: *Proc. VLDB Endow.* 5.6 (2012), pages 526–537. ISSN: 2150-8097. DOI: 10.14778/2168651.2168654. URL: http://dx.doi.org/10.14778/2168651.2168654 (cited on page 44).

[GP87] Jim Gray and Franco Putzolu. “The 5 Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time”. In: *Proceedings of the 1987 ACM SIGMOD International*

*Conference on Management of Data*. SIGMOD '87. San Francisco, California, USA: ACM, 1987, pages 395–398. ISBN: 0-89791-236-5. DOI: 10.1145/38713.38755. URL: http://doi.acm.org/10.1145/38713.38755 (cited on page 48).

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufıann Publishers, 1993 (cited on page 48).

[Hal+06] Alan Halverson, Jennifer L Beckmann, Jeffrey F Naughton, and David J Dewitt. "A Comparison of C-Store and Row-Store in a Common Framework". In: *University of Wisconsin-Madison, Tech. Rep. TR1570* (2006) (cited on page 47).

[16b] *HBase Native Client*. https://github.com/ apache/hbase/tree/master/hbase-native-client. May. 02. 2016 (cited on page 73).

[Kal+08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. "H-store: A High-Performance, Distributed Main Memory Transaction Processing System". In: *Proc. VLDB Endow.* 1.2 (2008), pages 1496–1499. ISSN: 2150-8097. DOI: 10.14778/1454159.1454211. URL: http://dx.doi.org/10.14778/1454159.1454211 (cited on pages 16, 21).

[KN11] Alfons Kemper and Thomas Neumann. "HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots". In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE. 2011, pages 195–206 (cited on page 16).

[Klo+14] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. "Building Efficient Query Engines in a High-level Language". In: *Proc. VLDB Endow.* 7.10 (2014), pages 853–864. ISSN: 2150-8097. DOI: 10.14778/2732951.2732959. URL: http://dx.doi.org/10.14778/2732951.2732959 (cited on page 44).

[16c] *Kudu Client Interface*. https://github.com/ cloudera/kudu/blob/master/src/kudu/client/client.h. May. 02. 2016 (cited on page 73).

[KL80] HT Kung and Philip L Lehman. "Concurrent Manipulation of Binary Search Trees". In: *ACM Transactions on Database Systems (TODS)* 5.3 (1980), pages 354–382 (cited on pages 89, 129).

[LM10] Avinash Lakshman and Prashant Malik. “Cassandra: a Decentralized Structured Storage System”. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pages 35–40 (cited on page 41).

[Lam78] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM* 21.7 (1978), pages 558–565 (cited on page 21).

[LS79] Butler Lampson and Howard Sturgis. *Crash recovery in a distributed data storage system*. Xerox Palo Alto Research Center Palo Alto, California, 1979 (cited on page 21).

[LY81] Philip L. Lehman and s. Bing Yao. “Efficient Locking for Concurrent Operations on B-trees”. In: *ACM Trans. Database Syst.* 6.4 (1981), pages 650–670. ISSN: 0362-5915. DOI: 10.1145/319628.319663. URL: http://doi.acm.org/10.1145/319628.319663 (cited on page 90).

[LLS13] Justin J Levandoski, David B Lomet, and Sabyasachi Sengupta. “The Bw-tree: A B-tree for new hardware platforms”. In: *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE. 2013, pages 302–313 (cited on pages 17, 86, 90, 98).

[Lip+15] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, and Andrew Wang. *Kudu: Storage for Fast Analytics on Fast Data*. Technical report. Cloudera, Inc., 2015 (cited on page 137).

[Loe15] Simon Manfred Loesing. “Architectures for Elastic Database Services”. PhD thesis. Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 22441, 2015 (cited on pages 28, 36, 109, 110, 113, 124).

[Loe+13] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. *On the Design and Scalability of Distributed Shared-Memory Databases*. Technical report. ETH Zürich, 2013 (cited on page 110).

[Loe+15] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. “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, pages 663–676. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2751519. URL: http://doi.acm.org/10.1145/2723372.2751519 (cited on pages 34, 36, 42, 110, 113).

[Mic04] Maged M Michael. “Hazard pointers: safe memory reclamation for lock-free objects”. In: *Parallel and Distributed Systems, IEEE Transactions on* 15.6 (2004), pages 491–504 (cited on page 89).

[Neu11] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: *Proc. VLDB Endow.* 4.9 (2011), pages 539–550. ISSN: 2150-8097. DOI: 10.14778/2002938.2002940. URL: http://dx.doi.org/10.14778/2002938.2002940 (cited on pages 44, 58).

[ONe+96] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. “The log-structured merge-tree (LSM-tree)”. In: *Acta Informatica* 33.4 (1996), pages 351–385 (cited on pages 46, 86).

[Ous+11] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. “The Case for RAMCloud”. In: *Commun. ACM* 54.7 (2011), pages 121–130. ISSN: 0001-0782. DOI: 10.1145/1965724.1965751. URL: http://doi.acm.org/10.1145/1965724.1965751 (cited on page 46).

[PR04] Rasmus Pagh and Flemming Friche Rodler. “Cuckoo Hashing”. In: *Journal of Algorithms* 51.2 (2004), pages 122–144. ISSN: 0196-6774. DOI: http://dx.doi.org/10.1016/j.jalgor.2003.12.002. URL: http://www.sciencedirect.com/science/article/pii/S0196677403001925 (cited on page 53).

[Pil+16] Markus Pilman, Martin Kaufmann, Florian Köhl, Donald Kossmann, and Damien Profeta. “ParTime: Parallel Temporal Aggregation”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: ACM, 2016, pages 999–1010. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2903732. URL: http://doi.acm.org/10.1145/2882903.2903732 (cited on page 150).

[Ram+13] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. “DB2 with BLU Acceleration: So Much More than Just a Column Store”. In: *Proceedings of the VLDB Endowment* 6.11 (2013), pages 1080–1091 (cited on page 58).

[16d] *RAMCloud Client Interface*. https://github.com/ PlatformLab/RAMCloud/blob/-master/src/RamCloud.h. May. 02. 2016 (cited on page 73).

[RO92] Mendel Rosenblum and John K. Ousterhout. “The Design and Implementation of a Log-structured File System”. In: *ACM Trans. Comput. Syst.* 10.1 (1992), pages 26–52. ISSN: 0734-2071. DOI: 10.1145/146941.146943. URL: http://doi.acm.org/10.1145/146941.146943 (cited on page 46).

[RKO14] 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, pages 1–16. ISBN: ISBN 978-1-931971-08-9. URL: https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble (cited on pages 41, 46, 69).

[Shv+10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. “The Hadoop Distributed File System”. In: *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE. 2010, pages 1–10 (cited on page 22).

[Smi03] Gordon Smith. “Oracle RAC 10g Overview”. In: *An Oracle White Paper, Oracle Corporation* (2003), pages 1–15 (cited on page 22).

[Sto86] Michael Stonebraker. “The Case for Shared Nothing”. In: *IEEE Database Eng. Bull.* 9.1 (1986), pages 4–9 (cited on page 20).

[SÇ05] Michael Stonebraker and Uğur Çetintemel. “”One Size Fits All”: An Idea Whose Time Has Come and Gone”. In: *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. IEEE. 2005, pages 2–11 (cited on page 23).

[Sto+05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O’Neil, Pat O’Neil, Alex Rasin, Nga Tran, and Stan Zdonik. “C-Store: A Column-oriented DBMS”. In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB ’05. Trondheim, Norway: VLDB Endowment, 2005, pages 553–564. ISBN: 1-59593-154-6. URL: http://dl.acm.org/citation.cfm?id=1083592.1083658 (cited on pages 37, 46, 47).

[TP14] Shahar Timnat and Erez Petrank. “A Practical Wait-free Simulation for Lock-free Data Structures”. In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’14. Orlando, Florida, USA: ACM, 2014, pages 357–368. ISBN: 978-1-4503-2656-8. DOI: 10.1145/2555243.2555261. URL: http://doi.acm.org/10.1145/2555243.2555261 (cited on pages 104, 106).

[Unt+09] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. “Predictable Performance for Unpredictable Workloads”. In: *Proc. VLDB Endow.* 2.1 (2009), pages 706–717. ISSN: 2150-8097. DOI: 10.14778/1687627.1687707. URL: http://dx.doi.org/10.14778/1687627.1687707 (cited on page 41).

[Wil+09] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. “SIMD-Scan: Ultra Fast In-Memory Table Scan using on-Chip Vector Processing Units”. In: *PVLDB* 2.1 (2009), pages 385–394 (cited on page 47).

[Zah+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. 2010, pages 10–17 (cited on pages 22, 123).

[ZR02] Jingren Zhou and Kenneth A. Ross. “Implementing Database Operations Using SIMD Instructions”. In: *SIGMOD*. 2002, pages 145–156 (cited on page 47).