University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Feasibility and Performance of RDBMS on top of DHTs

Francesco Luminati
Zürich, Switzerland
Student ID: 07-710-387

Supervisor: Dr. Thomas Bocek, Andri Lareida
Date of Submission: January 11, 2014

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Die jüngsten Entwicklungen im Datenbankbereich haben diverse NoSQL Systeme hervorgebracht, welche dank der hohen Skalierbarkeit in verteilter Art und Weise in der Lage sind, große Datenmengen zu verarbeiten. Die Nachteile solcher Systeme sind der Verzicht auf typische relationale Eigenschaften der Datenbankverwaltungssysteme (DBMS), wie z.B konkurrierende Zugriffe. Ein Beispiel solcher NoSQL Datenbanken ist die Distributed Hash Table (DHT), welche auf dem *key/value* Paradigma basiert. Relationale Datenbankverwaltungssysteme sind jedoch immer noch die bevorzugten Systeme von vielen Finanz-, Geschäfts- und Industrieanwendungen. Die vorliegende Arbeit entwirft, implementiert und evaluiert eine DBMS, welche das relationale Modell und die SQL Queries in *key/value* Paare und DHT Operationen übersetzt. Um das Konzept zu testen, sind verschiedene Experimente durchgeführt worden: INSERT, SELECT, JOIN und DELETE. Die Ergebnisse für die INSERT Operationen zeigen, dass die Indexstruktur mehr DHT Aufrufe in einer Grössenordnung von rund zehnmal mehr als ohne Index benötigt. Dieser Mehraufwand wird durch die Distributed Segment Tree (DST) Operationen verursacht. Die SELECT Operation funktioniert hingegen gut. Die Verwendung der Index Struktur führt zu schnelleren Resultaten, wenn im entsprechenden Experiment auf weniger als 75% der gesamten Tabelle zurückgegriffen wird. Der JOIN zweier Tabellen benötigt die doppelte Ausführungszeit einer SELECT Operation. Die Ergebnisse der Experimente sind vielversprechend. Jedoch fehlen gewisse Eigenschaften, welche entwickelt werden müssten, um daraus ein relationales DBMS zu erstellen.

Abstract

Recent developments in the context of databases have produced many NoSQL systems, capable of processing big data thanks to the high scalability in a distributed manner. The cost of this performance is the renunciation of typical relational DBMS properties, such as concurrency control. One example of NoSQL databases is the Distributed Hash Table (DHT), based on the *key/value* paradigm. Relational DBMSs are still preferred systems for many financial, business and industry applications. This thesis designs, implements, and evaluates a relational DBMS engine to translate the relational model and the SQL queries to *key/values* pairs and DHT operations. For the proof of concept, a series of experiments are executed: INSERT, SELECT, JOIN and DELETE. The results for the INSERT operations show that the indexing structure causes more DHT calls by an order of magnitude then without an index due to the Distributed Segment Tree (DST) operations. The SELECT operation, on the other hand, performs well. Using the index responds faster when the query selects less than 75% of the table for the executed experiment. Joining two tables is twice as slow as a SELECT operation. The results of the experiments are encouraging, however, missing features remain to be implemented in order to create a relational DBMS with its typical properties.

Acknowledgments

First of all, my most sincere thanks goes to my supervisors Thomas Bocek and Andri Lareida for the interesting topic proposed for my thesis and the constructive discussions we had. A special thanks goes to Thomas Bocek for the revision of the thesis and the support he gave me at any time when I was in need of help. Their experience contributed in a significant way to the success of the project and this thesis would not have been possible otherwise.

I am grateful to many members of the Communication System Group at the University of Zürich led by Prof Dr. Burkhard Stiller for giving me the opportunity to write this thesis and providing me with an excellent testing infrastructure.

Finally, I thank my friends and parents, who helped me during this work with precious advice; it is such as these little but constant contributions that make working easier and nicer.

Contents

1. Introduction

Peer-to-peer (P2P) are distributed networks with the main focus on resources sharing, such as CPU time, disk or memory. Important properties are scalability, because every connected member (peer) provides resources to the network and no single point of failure, given that there is no central server or hierarchical structure.

The most important application of P2P is file sharing, focused on sharing files between the connected peers such as music or movies. Every peer is acting as a downloader but at the same time also as uploader of the file. After the file sharing, also data sharing in form of distributed databases are an important field of P2P. One example is the Distributed Hash Table (DHT), a structure that permits to store *key/values* pairs between the peers.

Traditional databases, however, implement the relational paradigm; in other words the data are represented in tables of columns and rows. These database systems provide an attractive interface to access and manipulate the data and are very successful in many financial, business and Internet applications.

1.1. Motivation

On one hand, Distributed Hash Tables (DHT) ensure a number of interesting properties typical for P2P systems, such as self-organization, durability, fault tolerance and scalability. The data are stored in *key/value* pairs through a simple interface composed by a put(*key, value*) operation to store data, a get(*key*) operation to retrieve data and a remove(*key*) operation to remove data. The *key/value* construct permits to save schema-less data, in other words the key is derived with a hashing function and the value is the actual data, which can be completely unrelated to everything else in the database. However, for applications that involve elaborated data manipulation it may be a burden to rely on an interface that supports only data retrieval based on exact *key* match. A developer has to implement many functionalities on top of DHT that are usually supported by DBMSs to create such applications [1].

On the other hand a relational Database Management System (DBMS) implements the relational model, a structured way of representing data in form of tables. A new row inserted must follow the schema of the table. Structured data are then easy to query utilizing a query language such as the Structured Query Language (SQL) [2]. SQL permits very elaborated data manipulation like selecting only a small portion of the table using conditions or joining two tables over a column. The drawback is that the properties of P2P systems are not necessarily found also in relational DBMSs [3]:

- *Database systems are difficult to scale*: adding a new server to a relational database implies a lot of work in configuration, manual partitioning and data migration. Also the load balancing is not always automatic. In a P2P system connecting a new peer to the network is sufficient, everything else is self-organized.
- *Single point of failure*: in Client-Server architectures the server always represent a single point of failure, shutting it down makes the database unusable. Replication must be manually configured. In a DHT, on the other hand, removing a peer does not affect the network and replication is usually a standard functionality.
- *Peak provisioning leads to higher costs*: providing the infrastructure for peak times often leads to excessive resources in off-peak times. In a P2P system resources can be added and removed anytime and therefore be utilized for other tasks in off-peak periods.

The motivation of this thesis is to overcome the drawbacks of both approaches layering a relational DBMS over a DHT. In other words, a relational DBMS engine translates the relational model and the SQL queries in *key/value* pairs and DHT operations. The goals of the thesis are: design and implement a DBMS engine based on DHT and test the performances of the implementation.

1.2. Description of Work

This thesis is based on TomP2P [4], a DHT developed at the University of Zürich. The implementation of the DBMS engine, called TomDB, uses the API of TomP2P as the underlying storage engine. The main task of this thesis is to design and implement a DBMS for a subset of SQL commands utilizing only the standard DHT API. The implementation proposes a DB API similar to the Java DataBase Connectivity (JDBC) for INSERT, SELECT with or without indexes, JOIN, UPDATE and DELETE SQL commands. An indexing structure based on the Distributed Segment Tree (DST) [5] is implemented to permits indexing and range queries over DHT. The experiments are executed through the DB API in a separate application. The results are analyzed and graphically presented in the discussion.

1.3. Thesis Outline

Chapter 2 deepens the background technologies utilized in this thesis, in particular TomP2P, DBMSs in general and DST, and gives an overview of Related Works. In Chapter 3 the design of the application is explained in detail. Chapter 4 presents interesting aspects of the implementation. Chapter 5 shows the results of the experiments. In Chapter 6 the thesis is summarized and concluded. Finally, in Chapter 7 the Future Works are mentioned.

2. Background

In this Chapter the three fundamental building blocks of the thesis are presented. First of all, an introduction in DHT is done, as it is the underling network for the system. Then, an overview of relational DBMSs is given, as it compose the overlying structure of the system. The DST is used as overlying data structure for indexing.

2.1. Distributed Hash Table

Peer-to-peer (P2P) systems are decentralized, non-hierarchical networks to provide resource sharing [1]. Some characteristics are [4]:

- *Self-organizing*: peers organize themselves based on local observations.
- *Equality*: peers have similar rights; they are both client and server. Any peer can request, but also provide resources.
- *Decentralization*: they are self-organizing; there is no need of a central coordination.
- *Direct interaction*: peers communicate directly with other peers.
- *Resources sharing*: peers share resources with other peers such as storage or CPU.

After the unstructured and ad-hoc architectures of the first P2P-generation, systems arise with ordered structures, provable proprieties and good performances; one example is the Distributed Hash Table (DHT). It is also the fundamental concept of this thesis, on which the implemented application is based. Although various researches produced many different implementations of DHT, like Chord, Pastry or Kademlia the basic concepts remain the same: it is a decentralized system with similar lookup functions to Hash Tables with the difference of utilizing a consistent hashing function [6].

A Hash Table is based on associative arrays, a structure that maps keys to values. The keys are generated using various hashing algorithms. The hash is then reduced to the array size with the modulo operator ($key = hash \ \% \ array_{size}$). The *key* points directly to the data without the need to traverse the entire array. When the algorithm and the dimension of the array are such that every *key* receive a different memory slot (ideal conditions), Hash Tables perform the insert (put(*key, value*)) and retrieve (get(*key*)) operations with a constant cost independent of the number of stored elements. To keep the ideal condition, the array must be load under a certain threshold, for example 75%. When the threshold is reached, many implementations expand the table to fit within the parameter. The

resizing causes a rehashing of all the items to be distributed on the entire new table size, changing the modulo operator according to the new array size [7].

In DHT, on the other hand, the hashing function is consistent; this means that the *keys* newer change. Every connected node can get or put data based on *keys*. The responsibility for the *key/value* mapping is distributed among all the participants, in a way that joining or leaving the network will not affect the performance or the stored data. This design permits to handle an extremely large number of nodes. A DHT manages a keyspace, a set of all the possible *keys* in the network, usually derived by the maximal range of the hashing function. This keyspace is distributed across the participant nodes with various techniques. Each node is a *key* itself and it is responsible for the *keys* of the keyspace near to that *key* for a given distance. When a node leaves or enters the DHT, only a small amount of *keys* must be moved. To enable the lookups, every node maintains a list of links to other nodes. Given the sparse connection between nodes, a *key* lookup is passed from node to node until the node responsible for that *key* is found. The routing is driven utilizing the links that are closer to the *key*. In other words, every node passes the request to the node that is closer to the *key*, utilizing various distance metrics depending on the implementation [8].

In this thesis the selected DHT implementation is TomP2P [4], an extended DHT developed at the Zürich University. The main characteristics of this DHT are:

- Java 6 implementation with non-blocking IO based on Netty framework.
- XOR-based iterative routing with keyspace of 160-bit.
- Storage of multiple values for one *key*.
- Storage is memory-based or disk-based.
- Indirect and direct replication.
- Mesh-based distributed tracker.
- UDP communication.
- NAT traversal via UPNP and NAT-PMP.

To store multiple values for one *key*, which is called *location key* because it defines on which peer the *key* is saved, every peer creates a Hash Table for every *location key*. In other words, it is possible to save multiple *content keys* for every *location key* inside this Hash Table. Another important characteristic of the TomP2P API is the non-blocking communication: when a DHT operation is executed, a future object is immediately returned to keep track of future results. The operation can then be blocked to wait for the results, or a Listener can be added to get notified when the operation is completed. This key concept is used all over TomP2P API and permits to execute many operations in parallel and completely asynchronously [9].

The XOR-based iterative routing of TomP2P is similar to Kademlia [10]. Iterative routing means, in contrast to recursive routing, that a queried peer always sends a reply message back to the querying peer. In this way the querying peer can take control over the whole

Employees

Name	Surname	Department
Terrance	Golden	Marketing
Jeri	Fowkes	Sails
Felton	Stufflebeam	IT
Allen	Cail	IT

Departments

Department	Collaborators	Chief
Marketing	50	Archibald
Sails	100	Jeri
IT	150	Henri

Table 2.1.: Relational Model example based on a fictive employees database.

flow of the routing process. In Kademlia each node has a 160-bit node ID, usually a random construction. Every message sent by a node includes its ID, so that the receiver can save it if necessary. The *keys* in Kademlia are also 160-bit. In this way, the distance between two 160-bit identifiers can be calculated as the *bitwise exclusive OR* (XOR). Every node stores contact information about others nodes (IP address, UDP port, Node ID) in 159 bags with a capacity of 20. When a node receives a message, it updates the appropriate bag with the sender ID. If the bag is not full, the ID is added to the list, otherwise the system pings the least recent node ID from the list. If that node does not respond, it is removed from the bag and the new one is added, otherwise the new contact is discharged.

A *key/value* pair lookup proceeds as follows: the node queries α nodes in parallel from its bags that have the closest ID to the searched *key*. When the nodes get the request, they will return from their bags the k closest nodes to the *key*. The requester will then keep only the k closest nodes from the responses and repeat the request on them. This process continues until the nearest node returns the *key/value* pair [10].

The strengths of Kademlia are the continuous learning about neighbors and the facility to calculate the XOR distances, given *keys* of the same length. All this leads to high efficiency, the algorithm contacts only $O(log(n))$ nodes during a search out of n nodes in the system. The cost is a direct characteristic of the routing algorithm when the IDs are normally distributed on the entire keyspace. The hashing function is therefore a fundamental component for a performing DHT. These characteristics are reflected also in TomP2P.

2.2. Relational Databases

Edgar F. Codd proposed the Relational Model in 1970 revolutionizing the database world; several vendors were offering Relational Database Management Systems (rDBMS) a few years later supplanting the other database models. In the relational model, a database is a collection of relations and each relation is a table containing columns and rows. The tabular representation is simple to understand and visualize and permits to use high-level languages to query the data [11]. An example relational database is shown in Table 2.1, where the Department name is the relation between the tables.

A relational Database Management System (rDBMS), more generally a DBMS, is a set

of instruments to access and maintain one or more databases. All interactions with a database take place through the DBMS. The interactions fall into four groups [12]:

- *Data definition*: define, remove or modify the data structure of a database, i.e. table names and columns.
- *Data maintenance*: inserting, updating and deleting data (rows) from a database.
- *Data retrieval*: querying existing data.
- *Data control*: creating and monitoring users of the database, restricting access and monitor the performances.

The Structured Query Language (SQL) is the most widely used relational database language and was developed by IBM in the early 70s. SQL is a set of English-like commands used to communicate with a DBMS. The similarity to English of an SQL statement makes it easy to learn and understand, which determined its success [2]. For example, selecting all the employees that work in department "IT" in Table 2.1 is similar to an English sentence:

```
SELECT name, surname FROM Employees WHERE Department = 'IT'
```

SQL is a nonprocedural language, meaning that it is enough to tell SQL what data to be retrieved, rather than how to retrieve it. The DBMS takes care of locating and returning the information. SQL commands are classified in three types to cover all the functionalities of a DBMS: Data Definition Language, Data Manipulation Language and Data Control language.

SQL has a wide range of datatype (Strings, Integers, ...), used to define the type of a column, i.e. which type of data a column can take [2].

In summary, the advantages of a DBMS are:

- *Centralized data management*: it reduces redundancy and facilitates the management.
- *Data independency*: the information is isolated in terms of structure and storage. The DBMS always mediate as an abstraction layer between database and application.
- *Data consistency*: DBMS is designed to have data consistency, in other words, every copy of the same data has the same value and every transaction respects the rules and constraints of the DB.
- *Concurrency control*: the same data can be modified only by one instance at time.
- The properties above summarize to the ACID properties of databases. *Atomicity*: every transaction is "all or nothing"; *Consistency*: every copy of the data is the same; *Isolation*: transactions are submitted to concurrency control; *Durability*: a committed transaction remains constantly saved in the database.

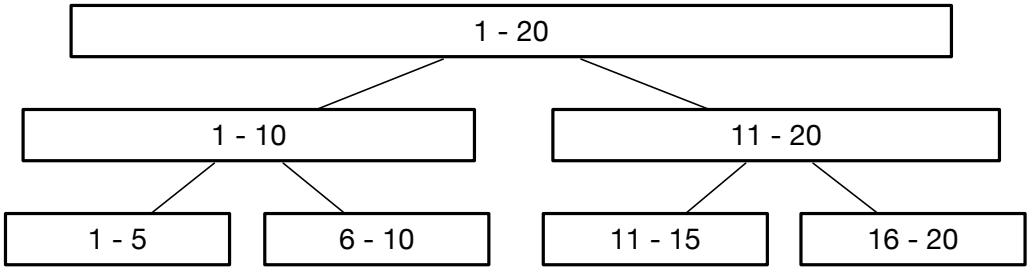

Figure 2.1.: Distributed Segment Tree example for a range from 1 to 20.

- *Performances*: DBMS stores directly on the disk and is optimized to give good performances.
- SQL as a standard query language.

A DBMS can usually be queried directly, using its own command line interface or it can be integrated inside an application using an API such as JDBC. Many DBSM are completely written in Java, like Apache Derby, H2 or HyperSQL. The most important commercial DBMS are Oracle, IBM DB2 and Microsoft SQL, the most widely used Open Source projects are MySQL and PostgreeSQL.

2.3. Distributed Segment Tree

Distributed Segment Tree (DST), proposed by Zheng et al. in [5], is the structure for indexing used in this thesis. DST represents a binary tree in which every node corresponds to a range of *keys* (segment). In other words, it is a data structure to support *range queries* over DHT. The root node corresponds to the entire range while every child nodes correspond to a sub range. The union of the child nodes on the same level returns the entire range. An example of a DST for a range from 1 to 20 is given in Figure 2.1.

The data structure behind the DST is the *segment tree* [13] with the properties listed below:

- The segment tree for a range of length L has a height $H = ln(L) + 1$.
- Each non leaf node has two children. The left child represents the range $[parent_{min}, \frac{parent}{2}]$, the right child represents $[\frac{parent}{2} + 1, parent_{max}]$. The union of the two child nodes returns the range of the parent node.

The particularity of the DST approach is that every node of the tree stores the *keys* it is responsible for. In other words, knowing the node or a union of nodes responsible for a given range is sufficient to locate the *keys*. This knowledge is reached with the *Range Splitting Algorithm* in Algorithm 1. It returns the minimal union of nodes necessary to retrieve the given range $[s, t]$ for the DST range $[lower, upper]$ (i.e. the range of the root node).

Algorithmus 1 : Range Splitting Algorithm [5]

Input : bounds of input segment s, t
bounds of current node interval $lower, upper$
Result : resulting union of node intervals ret

```
1 SplitSegment(s, t, lower, upper, ret)
2 if s ≤ lower AND upper ≤ t then
3 |   ret.add(interval(lower, upper));
4 |   return;
5 mid ← (lower + upper)/2;
6 if s ≤ mid then
7 |   SplitSegment(s, t, lower, mid, ret);
8 if t > mid then
9 |   SplitSegment(s, t, mid + 1, upper, ret);
```

The nodes of a DST are distributed on the DHT using the *key* hash(*[s,t]*) for the DHT operations put(*key, value*) and get(*key*). In other words, any node of the DST can be efficiently located using only the standard DHT interface. Every peer on the network can calculate *locally* the key of a DST node utilizing the *Range Splitting Algorithm*, as long as it knows the DST range. In addition, the DHT operations can be executed simultaneously and in parallel.

In a concrete case, a DST node cannot hold all the *keys* it is responsible for because of resources constraints. For these cases, a saturation point can be defined, which indicates the maximal number of *keys* a node hold. When this limit is not exceeded, a *key* is stored once on every level (high H) of the tree. However, when the limit is reached, a *key* is stored only on the children of the node. Accordingly, when a get(*key*) operation returns a saturated node, two more get(*key*) operations are necessary to ensure the retrieving of all the *keys* for the given range.

In summary, the advantages of this approach are:

- The *keys* are saved on every node of the tree.
- The static and regular tree structure can be calculated locally by every peer using the algorithm.
- DST is implemented using only the standard DHT interface.
- DHT operations can be executed simultaneously and in parallel, except in the case that a returned node is saturated.
- DHT operations minimized, if saturation is not reached, when querying the tree.

The disadvantages are:

- The static structure does not allow to dynamically changing the DST range once a tree is created.

NoSQL	
BigTable	Multi-dimensional map
Dynamo	*Key/Value*
Cassandra	Partitioned rows
TomP2P	*Key/Value*

SQL	
MySQL Cluster	Clustering
VoltDB	Horizontal partitioning, replication
Relation Cloud	DB as a Service

Hybrid	
Google F1	Replication, distributed SQL
[1]	MySQL engine for DHT
TomDB	DBMS over DHT

Table 2.2.: Summary of the Related Works.

- When inserting a new *key*, a DHT operation for every level (high H) of the tree must be executed.

- When the saturation point is reached, additional DHT operations are necessary to retrieve a *key*.

Although there are many approaches to support range queries over DHT, only a few are completely based on the standard DHT interface. A similar approach to DST is the Prefix Hash Tree (PHT) [14]. The PHT is a binary trie build over the data set, in which every node corresponds to a prefix that is calculated recursively: the left child is labeled *parent-prefix0* and the right *parent-prefix1*. The *keys* are stored on the leaf node whose label is a prefix of the *key*. When the leaf node reaches the saturation point, it splits in two child nodes, redistributing the *keys* on them. The DHT *key* for the put(*key, value*) and get(*key*) operations is hashed from the prefix. To query the trie, an algorithm retrieves the nodes with the prefix of the *key*, until a leaf node is reached.

Compared to DST, PHT stores *keys* only on leaf nodes and a client has no knowledge of the structure of the whole PHT. This means that more DHT operations are necessary to traverse the trie until a leaf node is reached. Moreover, the search is sequential. The advantage over DST is that PHT has a dynamic structure that adapts itself to the incoming *keys*, while in DST the structure is static.

2.4. Related Works

In the last decade the database world is divided in two major sections, the ”SQL” movement, embracing the relational model and the “NoSQL” flow, where the relational model is sacrificed to cope with big data and scalability.

NoSQL systems are designed to scale horizontally, i.e. resources are added attaching more nodes to the system. Some features are the ability to replicate and distribute data over many servers, a simple API interface, a weaker concurrency model compared to relational DBMSs and the ability to dynamically add new attributes to data records. The idea is to give up the ACID constraints (Atomicity, Consistency, Isolation and Durability), typical of relational DBMSs, and therefore achieve higher performance and scalability [15]. One important example is BigTable [16] by Google, a partitioned table that can scale to thousands of nodes and is the base of many Google applications, one over all the search engine. Amazon Dynamo [17] pioneered the idea of *eventual consistency*: the data fetched are not guaranteed to be up-to-date, but updates are guaranteed to be propagated to all nodes. Cassandra [18] is another arising example; open sourced by Facebook in 2008, it claims to be the marriage between Dynamo and BigTable. All the *key/value* pairs based storage systems, like TomP2P, are also important evidences of the NoSQL movement.

On the other hand there is the attempt to make relational DBMSs scalable. The approach usually consists of a clustering techniques, in other words the data are framed and replicated over multiple database servers. An overlay layer is developed to manage metadata, replication and query processing. Examples are MySQL Cluster [19] or VoltDB [20]. Recent improvements in this sector promise to give good per-node performances as well as scalability nearly comparable to NoSQL data stores keeping in account that operations like joins over many tables and in general transactions that involve many nodes do not scale and perform well [15]. Another idea is the relational cloud, proposed in [3], which introduces the term "Database as a Services". A user should have access on-demand to database functionalities without worrying about hardware and configuration, while providers should be able to easily administrate the cloud. One example is Google Cloud SQL [21], a relational database services backed by the Google infrastructures.

Another possibility is a hybrid approach, like Google F1 [22]. This system combines high availability and scalability of NoSQL databases with the consistency and usability of traditional SQL databases. F1 provides synchronous cross-datacenter replication to ensure consistency and a fully functional distributed SQL engine. The hybrid approach is still a new research topic and not many studies could be found. The study that comes closest to this thesis is "Layering a DBMS on a DHT-Based Storage Engine" by E.A. Ribas et al [1]. The paper propose an architecture for integrating DHTs and relational DBMSs, developing a prototype based on that architecture, along with an indexing structure for range queries based on DST. Experiments are conducted to show the impact of an index on query performances and to compare the horizontal and vertical fragmentation. The work consists of a new MySQL storage engine based on the Bamboo Distributed Hash Table. Bamboo utilizes a Pastry like routing protocol and communicates using TCP on HTTP. The experiment about table fragmentation shows that a horizontal fragmentation, when the number of selected columns grows, performs with a linear execution time, because always all the rows are retrieved. The execution time of the same experiment for a vertical fragmentation grows linearly. The entire relation of 2000 rows and 51 columns is retrieved in 45.24 seconds for the horizontal fragmentation and 1 minute and 29.50 seconds for the vertical, given a higher number of DHT operations. The vertical fragmentation performs better if less than half of the columns are selected. The experiment about indexes shows that to retrieve 25% of the table the index speedup 75% and to retrieve 13.3% of the table the speedup was of 81% compared to a full tablescan. The main difference

with the approach implemented in this thesis is the fragmentation. In [1] the horizontal fragmentation creates a *key/value* pair for every row and utilizes exclusively the standard DHT interface put(*key, value*), get(*key*) and remove(*key*). In this thesis, however, the extended interface of TomP2P is exploited to utilize *content keys* to save a list of n rows for every *key/values* pairs. Moreover, in this thesis the JOIN operation is implemented and evaluated.

3. Design

This chapter covers the design of TomDB, starting with a detailed analysis of the mapping between the relational model and *key/value* pairs. Then, the design of the application is explained.

3.1. Relational Model on Top of DHT

To provide the functionalities of a DBMS certain strategies are necessary to store relational data in a DHT. In other words the problem consists of mapping a table of information to *key/values* pairs so that a high efficiency is reached. The basic operations of a DBMS, insertion, deletion, retrieval and update need to be mapped to the basic API of a DHT, put(*key, values*), get(*key*) and remove(*key*). As a design principle for the implementation the extended API of TomP2P is used, which permits to store multiple values for one *key* (*x content keys* for one *location key*).

There are many ways of partitioning a relation into *key/values* pairs. This process is usually denoted as fragmentation. In a horizontal fragmentation each fragment is a subset of rows of the original table, in a vertical fragmentation each fragment is a column or a subset of columns of the original table [11]. Since the fragments are stored as *key/value* pairs in the DHT it is necessary to define how a fragment is mapped to a value and identified by a *key*. For that, an example relation is given in Table 3.1. The table is composed by 10 columns and 1000 rows and an internal *row ID* is given following the insertion order.

3.1.1. Vertical Fragmentation

The vertical fragmentation, or column-oriented approach, has its advantages when an application aggregates large numbers of similar data, i.e. a query selects many rows of a small subset of columns.

rowID	A	B	C	D	E	F	G	H	I	J
1	a-val	b-val	c-val	d-val	e-val	f-val	g-val	h-val	i-val	j-val
2	...									
...										

Table 3.1.: Relation of 10 columns and 1000 rows with an internal *row ID*.

The *key* representing the column name and the value as an array containing the entire column could compose the mapping to a *key/value* pairs. This could be further decomposed in blocks of n rows of a column when the table grows, to avoid big fragments. For the example given in Table 3.1 using the extended DHT API the mapping is as follows: the *location key* is composed by hash(*column-name + block number*) and the *content keys* by hash(*row ID*) for n rows. The value corresponds to the field of the *rows ID* for that column. Since that every fragment corresponds to a DHT operation, it is interesting to calculate how many operations are necessary to retrieve the entire table of the example. Setting $n = 100$, 100 *rows*/100 results in 10 blocks per column, for 10 columns it is 100 fragments distributed in the DHT. In other words, to retrieve the entire table mapped with the vertical approach, 100 DHT operations are necessary.

3.1.2. Horizontal Fragmentation

The horizontal fragmentation, or row-oriented approach, is the most diffused architecture for relational databases and is implemented by the major DBMSs. The advantages are given by the fact that many applications only need to retrieve information about a single or a few objects, like for example the contact information of a client. Moreover, it is more efficient for inserting new rows in the database.

The *key* representing the *row ID* and the value as the serialized row could compose the mapping to a *key/value* pairs. In the example given in Table 3.1 the mapping is as follows: the *location key* is composed by hash(*row ID*) and the *content keys* correspond to the hash(*column-name*). The value of every *content key* is then the field of the column for that row. The example table is composed by 1000 rows, i.e. 1000 fragments spread in the DHT. To retrieve the entire table mapped with the horizontal approach, 1000 DHT operations are necessary.

In this case an aggregation of n rows per fragment would reduce the number of operations necessary, which is also the approach of choice for this thesis, explained next.

3.1.3. Horizontal Fragmentation in Blocks

In the examples above the vertical fragmentation is the clear winner with only 100 DHT operations against the 1000 necessary for a horizontal fragmentation. In fact also in [1] it is proved that if a query selects less than 50 % of the columns, the column-oriented approach performs better. On the other hand, in a disk-based database, the sequential reading of many adjacent rows is very fast because the organization on the disk preserve the locality; this is another reason why the horizontal fragmentation is normally preferred by DBMSs. In DHT, however, the hashing algorithm do not preserve locality so that the cost of a DHT operation is always the same. To maintain a certain degree of locality also in DHT, every fragment could be formed by an aggregation of adjacent rows. In other words, the horizontal fragmentation could be organized in blocks of n rows.

This is also the design implemented in TomDB. The blocks of n rows are consecutively constructed following the insertion order of the incoming rows. In other words, when a

Location key	*Content key*	*Value*
hash(Block:tablename:[1..10])	hash(row ID 1)	Serialized row
	hash(row ID 2)	Serialized row
	...	...
	hash(row ID 10)	Serialized row

Table 3.2.: Example of the block [1..10].

new row is inserted, it is added to the last block utilizing a *content key* hashed from the *row ID*, until the block is full. When the blocks is full, a new block with a range of n is created. This assumes that the locality is given by the insertion order. The *location key* of the block is hashed from the string "*Block:tablename:[from..to]*", where "*from..to*" stands for the range that the block is responsible for. For instance, starting from 1 with $n = 10$, the first block would be "*Block:tablename:[1..10]*", the second "*Block:tablename:[11..20]*". An example of the block [1..10] is given in Table 3.2.

The structure of the blocks permits to identify the block responsible for a *row ID* or the blocks responsible for a range of consecutive *row IDs* with only n as an input in a simple algorithm. In other words, every instance of TomDB retrieves the metadata about n and is therefore able to calculate the hash of a block and get the block from the DHT for the searched row. This simple storage structure permits to organize the data without an overlay structure (e.g. DST); the information stored in the DHT are always retrieved with the minimal necessary number of DHT operations.

3.2. Approach

This section covers the approach followed for the implementation of TomDB. To implement an application based on a P2P system certain design principles should be applied to get the best performances; therefore, during the implementation the following rules have been kept in mind:

- Minimize the DHT operations.
- Execute the DHT operations simultaneously and in parallel.
- Avoid blocking operations, utilizing listeners and callback functions.

As a result of these principles, asynchronous operations are implemented everywhere the application uses the DHT API. In other words, the DHT operations are executed simultaneously and in parallel on different threads and the results are returned through a callback function to the same object to save them. This multithreads architecture is not trivial to design and leads often to faults or omissions, one over all concurrency problems difficult to debug. For example, when a thread is writing the results in a Collection and at the same time another thread is reading the Collection, it is possible that the reading process do not see the new results in time, leading to an inconsistent execution of the program.

The application is designed to support following SQL statements:

- Create Table (e.g. CREATE TABLE table_1 (column_a, column_b, ...)
- Insert (e.g. INSERT INTO table_1 VALUES (column_a_val, column_b_val, ...).
- Select with conditions (e.g. SELECT FROM table_1 WHERE column_a > 10). Multiple conditions with AND and OR are also supported.
- Select with joins (e.g. SELECT FROM table_1, table_2 WHERE table_1.column_a = table_2.column_a). The join can be further filtered with AND and OR conditions.
- Update (e.g. UPDATE table_1 SET column_a = 'new val' WHERE column_a > 100).
- Delete (e.g. DELETE FROM table_1 WHERE column_a < 10).

3.2.1. Main Classes and Metadata

The starting point of the application is the TomDB class, responsible for creating the DHT peer and the connection to the DB. The application can be started standalone as one or many DHT peers to add resources to the network or a connection to the DBMS can be established through the API to utilize the library inside a project. When a connection is established, the central class of the application, DBPeer, is initialized. This class is responsible to have a connection to the DHT, through which every DHT operation is executed and to manage the database metadata.

The metadata keep information about three aspects of the table, the column names, the rows and the indexes. The metadata about columns contains only a map of the column name and column id. The rows information contains the block size (n), the total number of rows saved in the database and the type of storage. The indexes metadata contains the name of the indexed columns, separate in normal or univocal indexes, the range of DST and the minimal and maximal value contained in an index.

In the DBPeer, the metadata are fetched from the DHT with a blocking operation, because it is crucial for the application that the metadata are actualized before everything else can be executed. The update, however, is done with a non-blocking operation.

3.2.2. Inverted Index

Conventional DBMSs usually come with indexing facilities to optimize the execution of a query, one example is the B^+tree. Thanks to these structures a query is directly addressed to the target, avoiding a scan of the entire table. An indexing mechanism becomes even more important in a DHT based DBMS, because it is not possible to scan through adjacent blocks like it happens in a disk-based DBMS; the locality of blocks is lost through the hashing function.

For the implementation, the concept of inverted index is utilized, a structure that maps the indexed content to the location of the content in the database. In a concrete example (Table 3.3), the indexed value, which corresponds to the value of a row of the indexed

Location key	*Content key*	*Value*
hash(DSTBlock:tablename:cloumnname:[1..1000])	hash(indexed val 1)	Row ID 1
	hash(indexed val 2)	Row ID 2
	...	...
	hash(indexed val 1000)	Row ID 1000

Table 3.3.: Example of an inverted index saved in DST blocks for the root DST block [1..1000].

column, is hashed as the *content key* and the *row ID* is saved as the value. The *location key* is calculated utilizing the DST overlay structure. DST is based on ranges and therefore, it supports the indexing of only Integers values. Ideally, the indexed column contains a range of entries that is smaller than the DST range. In this way a lookup in the DST performs optimally and in parallel. When the indexed range is bigger than the DST range, full blocks are encountered during the lookup process, causing the execution of more DHT operations to retrieve the child of the full blocks. An indexed value is assigned to a DST block by the DST Algorithm seen in Algorithm 1. The DST block is then hashed as the *location key*. This data structure permits then to efficiently retrieve an indexed value from the DHT utilizing the DST Algorithm. The input information for the DST Algorithm is covered by the metadata about indexes, especially the DST range and the minimum and maximum value in an indexed column, utilized to delimit the ranges.

3.2.3. Storage Type

In Subsection 3.1.3 it is explained that the table blocks are constructed following the insertion order of the new incoming rows. The blocks remain full if nothing is deleted from the table. When a DELETE operation removes rows from the table, however, it happens that blocks contain empty sectors, i.e. unassigned *row IDs*. If later new rows are inserted in the table following the insertion order, they are added to the last block without using the empty sectors. A select operation of the entire table needs now to get more blocks from the DHT and certain blocks are not completely full, deteriorating the performances. This is why, together with the insertion order storage, it was decided to implement also a full blocks storage.

The full blocks storage adds new rows first to the free sectors spread in the table, reutilizing the unassigned *row IDs*, and only when every free sector is occupied again, it adds to the last block, as seen in Figure 3.1. The information about free sectors is saved in the DHT as a metadata during the DELETE operations and accessed every time a new INSERT with full blocks storage is executed.

3.2.4. API

The Java Database Connectivity API (JDBC) [23] inspires TomDB API. Utilizing the static function *getConnection()* inside the TomDB class, it is possible to connect an application to the DBMS and the DHT. TomDB is implemented with the Singleton Pattern,

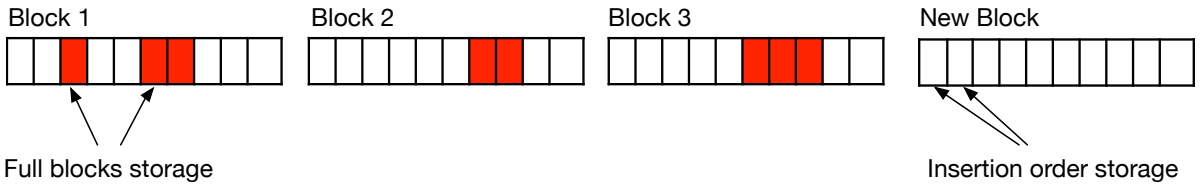

Figure 3.1.: Blocks of table with empty sectors in red: the full blocks storage adds new rows to the free sectors, the insertion order storage at the end of the table.

Listing 3.1: Execute API operation

```
1 public void executeStatement() {
2         Connection connection = TomDB.getConnection();
3         Statement stmt = connection.createStatement();
4
5         stmt.execute("INSERT INTO table_1 VALUES (row1_a, 1)");
6         stmt.execute("INSERT INTO table_1 VALUES (row2_a, 2)");
7         stmt.start();
8 }
```

creating only one DHT peer and one DBpeer object per execution of the program. TomDB offers also a function to create local DHT peers. These peers are set to a master peer and therefore all utilize the same port to avoid conflicts. Once a Connection object is instantiated, many Statement objects can be created with the *createStatement()* function. A statement serves to actually execute a SQL query. The *execute()* function (Listing 3.1) is utilized to execute queries that do not return a result, such as INSERT, UPDATE or DELETE. These queries can be buffered and the execution needs to be started with the *start()* method. The *executeQuery()* function (Listing 3.2) returns a ResultSet object containing the rows retrieved by the SELECT operation. The ResultSet is then utilized by the application inside a loop to elaborate the data coming from the database.

The ResultSet is implemented utilizing a Blocking Queue. The *next()* function returns *true* if a new row is arrived in the ResultSet, otherwise the Queue blocks the thread and waits until a new row asynchronously comes from the DHT. Once a row is set in the ResultSet, the values of a column can be read utilizing *getString()*, *getInt()* or *getDouble()*, depending on the type necessary for the application. These functions accept the column name as a String or the column ID as an Integer as parameter. When all the results are arrived from the DHT, the *next()* function returns *false* and the SELECT query is completed.

CREATE TABLE Statement

Adding a new table to the database consists of an update of the database metadata. The SQL parser interprets the information about columns, rows and indexes written in the SQL query. If the query is correct, the information is saved inside a CreateTable object, a sort of container of the parsed query. This kind of objects can be buffered inside the

Listing 3.2: Execute Query API operation

```
1  public void executeQueryStatement() {
2          Connection connection = TomDB.getConnection();
3          Statement stmt = connection.createStatement();
4
5          ResultSet results = stmt.executeQuery("SELECT * FROM
               table_1");
6
7          while(reults.next()) {
8                  System.out.println("Column_a: "
9                  + results.getString("column_a")
10                 + " Column_b: " + results.getInt(2));
11         }
12 }
```

Option	Value	
index	column name	Can be defined multiple times.
univocalindex	column name	Can be defined multiple times.
blocksize	integer from 1	Can not be changed.
dstrange	integer from 1	Can not be changed.
storage	insertionorder/fullblocks	

Table 3.4.: Options utilizable in a CREATE TABLE statement.

QueryEngine, responsible for the execution of the queries. When the *stmt.start()* API function is called, the buffer is executed. The information about the new table is first added to the metadata locally in the DBPeer object and then saved in the DHT, available for all the other users. The SQL of this statement presents a custom OPTIONS command used to set the parameters for the table: indexes and univocal indexes, blocks size, DST range and storage type (insertion order or full blocks). For every table multiple indexes can be added. With univocal it is intended the unique operator of SQL. The options are grouped in Table 3.4. An example for a CREATE TABLE statement is listed here:

```
CREATE TABLE table_1 (column_a, column_b, ...) OPTIONS (index
   :column_a, univocalindex:column_b, blocksize:100, dstrange
   :1000, storage:fullblocks)
```

INSERT Statement

The INSERT statement flow chart is shown in Figure 3.2. The parser interprets the query and creates an Insert object to be buffered inside the QueryEngine. The buffering is essential because the INSERT operation needs to update the local metadata from the DHT before the new row can be inserted and upload the actualized metadata to the DHT at the end. These operations, however, are done only once for the entire buffer.

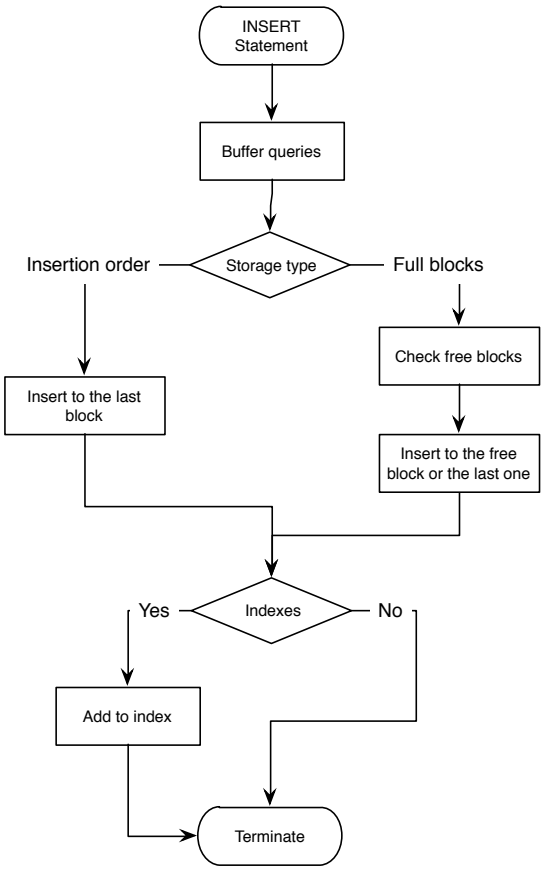

Figure 3.2.: INSERT statement flow chart.

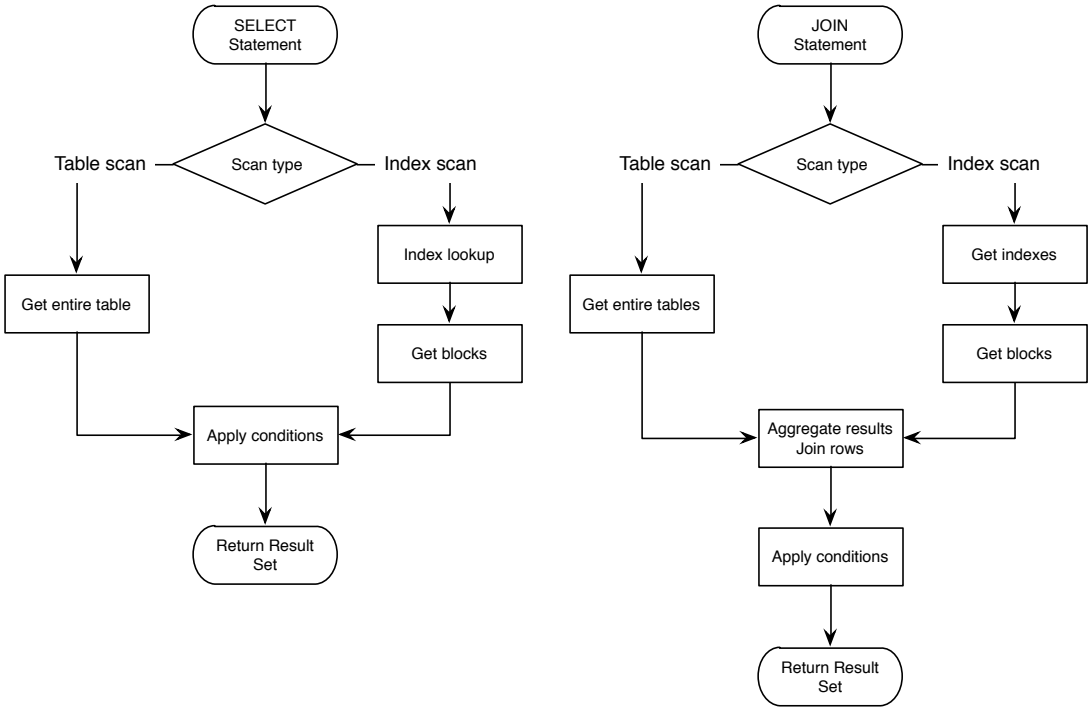

Figure 3.3.: SELECT and JOIN statements flow charts.

The INSERT operation depends on the storage type. For the insertion order storage, the row receives a new *row ID* calculated using the metadata about the number of rows saved in the table so far plus one. Thanks to the *row ID* and the block size, an algorithm can calculate the *key* of the last block of the table. The insertion order concludes with a put operation of the new row inside the DHT, adding the *row ID* to the last block. The full blocks storage, on the other hand, checks first on the DHT if free *row IDs* of precedent DELETE operations are available. If an unassigned *row ID* is found it is used for the new row; otherwise it performs an insertion order operation. Also here, using the *row ID* and the block size, the block responsible for the ID is calculated and a put operation is executed.

The last step is to update the indexes. For normal indexes the new *row ID* and indexed value is directly inserted in the index, because this kind of indexes can contain multiple entries of the same indexed value. For univocal indexes however, the index must be first queried to verify that the indexed value has not been inserted yet, otherwise the INSERT operation must be stopped and an error is thrown.

SELECT Statement

The flow charts for the SELECT and join SELECT operations are visualized in Figure 3.3. The query is parsed and a Select object is instantiated. In this case, however, the operation is executed immediately and an empty ResultSet object is returned to the application. The design renounces to update the table metadata for every query; the DBPeer updates them only every minute if it is necessary. The DBMS does not provide an automatic query optimizer, capable of deciding if a tablescan or an indexscan is faster to execute

the query. For that, an OPTION command is provided to define with which scan a query must be executed. The query is therefore extended to this:

```
SELECT * FROM table_1 WHERE id < 10 OPTIONS (tablescan/indexscan)
```

Tablescan means that the entire table is retrieved, i.e. all the blocks of a table are downloaded from the DHT. An indexscan, on the other hand, gets first the involved *row IDs* from the DST index and then downloads only the table blocks that are responsible for those *row IDs*. The indexscan works only if a WHERE condition is defined in the query. The column of the condition needs to be indexed, so that the indexscan can restrict the index lookup to the values that are included by the condition. In the query example above, the index lookup on column ID would search only for values smaller than 10. In this way, the number of DHT operations can be reduced.

In the last step, the ConditionsHandler applies the WHERE conditions, returning to the ResultSet only rows that match them. In a query it is possible to define which columns to select instead of select all (*). In these cases, the ConditionHandler is going to filter out the unneeded columns and returns a new row with only the selected columns.

In a SELECT statement, every DHT operations happen asynchronously. For instance, the index lookup starts a number of parallel operations to get the *row IDs* from the index. Once the result of an operation comes back, a get operation for the table block responsible for the *row ID* received from the index starts immediately. Similarly, once a table block arrives, the conditions are immediately applied and the rows are sent to the ResultSet. The same succeeds for a tablescan. This approach gives advantages in terms of performances and provides immediately results to the ResultSet, avoiding that the application has to wait until the operation is complete.

Even if the join SELECT reutilizes many aspects of the normal SELECT, it is to some extent quite different. A supported query is for example:

```
SELECT * FROM table_1, table_2 WHERE table_1.column_a =
   table_2.column_b OPTIONS (tablescan/indexscan)
```

Only two tables can be joined at once. The first equality condition is used by the system to know on which columns to join the tables and it is mandatory. Also here the distinctions in tablescan and indexscan is possible. In a tablescan, both tables are entirely downloaded from the DHT. In the indexscan, only the entire indexes of the joining columns are downloaded. The indexed values are then matched; if a match is found, the blocks of both tables responsible for the matching value are retrieved from the DHT. At this point, the system needs to wait until every DHT operation terminates. The rows of table A are then compared to the rows of table B, if a match is found, a new row containing both rows is created and sent to the ConditionHandler. The further AND and OR conditions defined in a query are applied here and the rows are returned to the ResultSet.

DELETE and UPDATE Statements

The flow charts for the UPDATE and DELETE operations are shown in Figure 3.4. Examples of supported queries are:

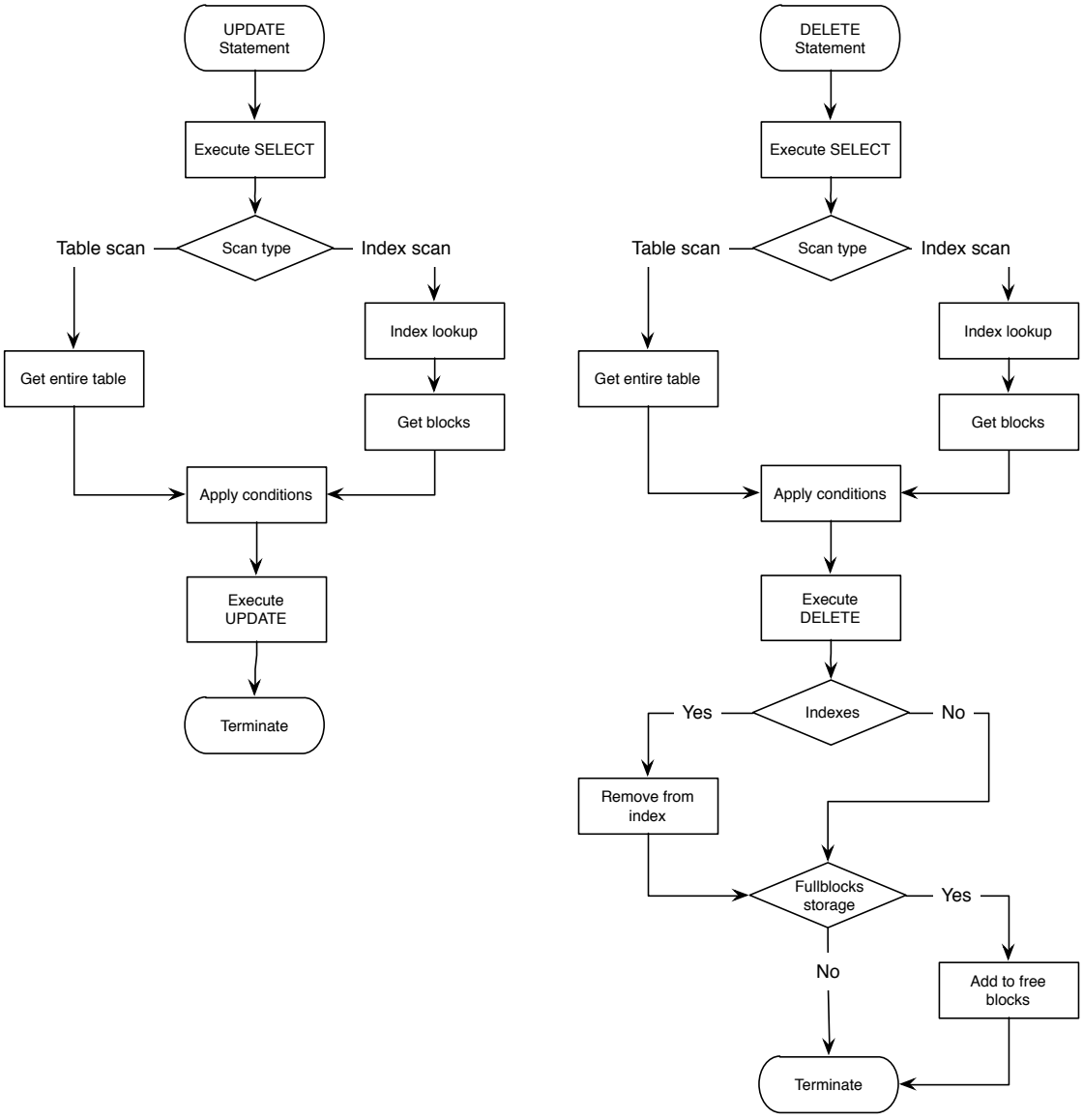

Figure 3.4.: DELETE and UPDATE statements flow charts.

```
UPDATE table_1 SET column_a = 'new value' WHERE column_a > 10
    OPTIONS (t.s./i.s.)
DELETE FROM table_1 WHERE column_a < 10 OPTIONS (t.s./i.s.)
```

Also for these operations the parser is going to create either an Update or a Delete object and the buffering of the queries is possible. The table metadata are actualized for every buffer because these operations are going to write in the table. The first operation for these statements is always a SELECT. In other words, a Select object is created and executed. In this case, however, the results are not returned to a ResultSet, but they come back to the Update or Delete objects. This design permits to reutilize the SELECT infrastructure, like the ConditionsHandler, tablescan and indexscan, to exactly identify the rows that need to be updated or deleted.

The last operation of an UPDATE consists of actualizing the rows coming from the SELECT and to put them back to the DHT. The DELETE, on the other hand, puts an empty row back to the DHT. When an index was defined, the DELETE needs also to delete the indexed value from the DST. For the full blocks storage, the DELETE operation produces unused *row IDs* that can be recycled by an INSERT operation. For that, the metadata about free blocks must be updated on the DHT.

4. Implementation

This Chapter describes in details important aspects of the implementation. First, details about the SQL parsers are given. Then, the DHT API is exposed, together with the callbacks architecture and the concurrency problems. Also the indexing system and the DST functions are explained in details. TomDB is composed by 44 classes divided in 10 packages and about 4000 lines of code.

4.1. SQL Parser

The SQL parser is implemented from scratch utilizing the tokenization technique. Tokenization means splitting a string of text into words, symbols or other elements called tokens. This is also the first process of the parser: the SQL query is divided into SQL specific words, like CREATE, SELECT etc. and punctuation marks in form of strings (COMMA, EQUAL, ...). An example of tokenization is given here:

```
CREATE TABLE table_1 (column_a, column_b)

becomes:

[CREATE, TABLE, table_1, POPEN, column_a, COMMA, column_b,
   PCLOSE]
```

To do that, the Tokenizer class utilizes the *String.split()* function with a sequence of regular expressions. The resulting tokens are saved in a list maintaining the order. Moreover, the Tokenizer object acts as an extended iterator through the list of tokens, implementing the ListIterator interface and providing the functionality to get the actual or the previous element, apart all the standard functionalities.

The SQL parser utilizes the iterator to first identify the type of query (CREATE, INSERT, SELECT, DELETE or UPDATE) and then to verify that the sequence and the syntax of the query is correct. When a problem is detected, a MalformedSQLQuery exception is thrown, where the problematic token is signalized. The parsing process consists of simple IF, SWITCH and WHILE constructs to check the tokens and move forwards the iterator. The values inserted in a query, like the table name, are saved in variables or lists. Once that the query is completely parsed and results to be correct, the specific operation object, for instance an Insert object, is created giving all the query variables and lists as parameter.

Listing 4.1: DHT GET operation with callback.

```
1  public void getFromDHT() {
2   FutureDHT future = peer.get(Number160.createHash("blockName").
       setAll().start();
3   future.addListener(new BaseFutureAdapter<FutureDHT>() {
4           @Override
5           public void operationComplete(FutureDHT future) throws
               Exception {
6                   if (future.isSuccess()) {
7                           callback(future.getDataMap());
8                   } else {
9                           System.err.println("GET failed!");
10                  }
11          });
12 }
```

4.2. DHT Operations and Asynchronous Callbacks

The DHT operations are consistent with the TomP2P API and utilize the Java Future and Listener interfaces. An example is given in the Listing 4.1 for a GET operation.

In the second line the future object is created and the entire set of content keys is retrieved. The future is then asynchronously started. To elaborate the answer from the asynchronous call, a listener is added in line 3. When the operation is completed, the *operationComplete()* function is called, returning the future object containing the results, in form of a Map.

The classical approach to elaborate the result is to define somewhere a callback function and to call it from the listener. In the example, the callback is inside the same object that executed the DHT operation. In most cases the application executes many DHT calls at once in parallel. The callback function is called once for every future object, asynchronously and often in a fraction of second. This can cause concurrency problems, because there is no control or locking on the callback. When the results are further elaborated and sent forwards without saving them, like it happens for example in a tablescan, where the results are directly sent to the ConditionsHandler and then to the ResultSet, the concurrency problem is not manifested, because Java can handle simultaneous execution of the same methods without problems. The concurrency problems arise when the callback function tries to write the results in a Collection and at the same time another function reads from the Collection. To overcome these problems, a Concurrent Collection is necessary, where the Collection Interface is implemented to be thread-safe. In other words, a Concurrent Collection permits the execution of read and write operations from many different threads. For TomDB, the ConcurrentHashMap is utilized. This Map implements a fine-grained locking, supporting multiple reader simultaneously while a portion of the Map is locked for the writing process.

4.3. Indexes and DST

The Distributed Segment Tree is designed to fit in the standard DHT API; nevertheless the implementation requires certain peculiarities. The Insert operation of a new value in the DST is simply reached executing a number of parallel DHT put operations equal to the height of the tree. This implementation does not check if the DST range is exceeded. For that, a custom StorageMemory is defined on every peer, overriding the *put()* method of TomP2P. The *put()* function is extended to check the Map size, if the size exceed the DST range, the incoming *content key* is not inserted in the given *location key*.

Querying a value from the DST, on the other hand, needs to be implemented with a recursive function. The recursion comes into play whenever a full DST block is returned. In that case, also both child blocks must be retrieved to guarantee that all the indexed values are found. To reach the recursion in Listing 4.1, when a full block is detected, i.e. the returned Map size is equal to the DST range, the callback function is substituted to call *getFromDHT()*, itself, for the two child blocks, until only non-full blocks are reached.

The IndexHandler class is used to insert and remove indexed values from the DST. Both operations need to first query the index to detect if the value is already there. When the response is positive for a univocal index, the entire Insert operation must be stopped. For a normal index the value is added to the same object. The checking operation is implemented to block the thread until the answer arrives, utilizing the *Thread.wait()* and *Thread.notify()* construct of Java. This breaks some of the rules prefixed for the design. The approach, however, simplified the implementation of IndexHandler.

5. Evaluation

This chapter focuses on the evaluation of TomDB. The goal is to run a series of tests and evaluate the main functionalities of the program, i.e. the INSERT, SELECT, JOIN and DELETE SQL statements. The performance results indicate if the implemented approach is a viable way for future developments and a real world use.

The evaluation is based on four experiments, one for each statement. Every scenario runs 10 times utilizing the same dataset.

5.1. Environment

The experiments are conducted on a test bed of 7 servers. The hardware specification is shown in Table 5.1. The servers are connected on a Gigabit Ethernet and isolated from the Internet. In this way the experiments are performed in a close environment to avoid external effects. All the Java applications are executed with the last Java JDK 7u45.

For every experiment, a TomP2P network is built up on 6 servers and on each server 167 local DHT peers are started, for a total of 1002 peers distributed on 6 machines. One of the server takes over the rule of bootstrapping peer and the other peers are bootstrapped to its address. The remaining server is utilized to execute the tests. The experiment application initializes only one DHT peer for the experiments and bootstraps it; in this way the communication over the DHT always needs to go through the Ethernet (worst case).

The dataset utilized by the experiments comes from the Ohloh.net webpage, an online community platform that collects statistics about Open Source softwares. Through the Ohloh.net API, information about 6000 Contributors and 1000 Projects are fetched and saved in text files formatted as Comma Separated Values (CSV). The Contributors table contains statistics about a person that committed code to a project. The Projects table contains information about Open Source projects, like names and web pages. The tables are further elaborated to add an unique ID starting from 1 and a random unique ID (rID)

	Servers Hardware
CPU	24 AMD Opteron 6180 SE Cores, 2.5 GHz
RAM	64 GB
OS	Ubuntu Server

Table 5.1.: Testbed hardware specification.

Contributors:

```
1,2788,11319,Gavin Sharp,44,C++,335,9789,21.083333333333332
2,3814,141602,Ed Morley,44,C++,355,7590,24.571428571428573
3,5805,48701,Mounir Lamouri (volkmar),44,C++,70,2056,27
4,3071,44171,Alex Surkov,52,Make,40,906,20
5,18,28554,Mossop,44,C++,162,3213,16
6,3393,52778,Vivien Nicolas,44,C++,117,2179,15.666666666666666
7,4747,796,sayrer,44,C++,33,723,20
8,5629,13506,Josh Aas,6,JavaScript,33,674,8
9,4789,6653,timeless,44,C++,472,7424,8.333333333333334
10,1072,485,Ben Goodger,6,JavaScript,25,641,13
```

Projects:

```
1,87,9,Mozilla Firefox,11781,4.43993,11874782,44,C++,8212026
2,942,72,Apache HTTP Server,8377,4.53223,11823084,42,C,1634476
3,11,1,Subversion,8334,4.22561,11818002,42,C,491569
4,563,4139,MySQL,8107,4.29966,11832516,44,C++,1423109
5,45,28,PHP,6864,4.27387,11911968,42,C,3782947
6,788,3141,Linux Kernel,6049,4.69384,11815032,42,C,15382092
7,728,4810,Firebug,5547,4.66753,11879863,6,JavaScript,474801
8,52,5120,Bash,5441,4.45063,11547047,42,C,188589
9,775,3745,Ubuntu,4586,4.45979,11875856,17,C#,974209
10,77,29,Apache OpenOffice,4453,3.97861,11840807,44,C++,19603114
```

Figure 5.1.: Extract of the Contributors and Projects dataset.

of the same range of the unique ID, randomly distributed on the entire table. An extract of the dataset is shown on Figure 5.1.

5.2. Experiment Design

The experiments are implemented in a separate application; in this way, TomDB is utilized as an external library exclusively through the API, directly from the compiled Jar with dependencies. The application is designed to execute from the command-line. To start the execution, the address of the bootstrapping peer must be given as parameter. After that, a Scanner reads the inputs from the keyboard, for example the name of the experiment to start or "exit" to shutdown the DHT peers and the experiment.

For every experiment the dataset is needed, i.e. it must be read from the CSV file and load in Java. The implementation utilizes a standard BufferedReader to load the file in memory, then, the *String.split(",")* function is used over every row to separate the columns delimited by a comma and returning an array for every row. This array of columns is saved in an ArrayList and an iterator is extracted. The iterator is then used in a loop to load the values in the INSERT queries.

5.2.1. Insert Experiment

The Insert experiment is subdivided in two scenarios, the first scenario inserts rows in a table without indexes, for the second scenario one index is defined. The experiments start with a CREATE TABLE statement to prepare the database for the inserts. The entire

Exp.	Inserted Rows
Ex1	1
Ex2	100
Ex3	200
Ex4	300
Ex5	400
Ex6	500
Ex7	600
Ex8	700
Ex9	800
Ex10	900
Ex11	1000

Table 5.2.: Insert experiment setup.

Contributors dataset of 6000 rows is utilized. For the first scenario, no OPTIONS parameter are given, utilizing the default settings (block size 100):

```
CREATE TABLE contributors (id, rid, account_id, account_name,
    main_language_id, main_language_name, man_months, commits
   , median_commits)
```

For the second scenario, the OPTIONS parameter defines the index on the ID column with a DST range of 10'000, block size remains by default:

```
CREATE TABLE contributors (id, rid, account_id, account_name,
    main_language_id, main_language_name, man_months, commits
   , median_commits) OPTIONS (univocalindex:id, dstrange
   :10000)
```

The experiments are organized in 11 sub experiments for each scenario, summarized in Table 5.2. Each sub experiment is separated by a 2 seconds break to send the DHT in an idle state. It starts inserting one row, then 100, 200, ..., to 1000; the "one row" experiment tries to simulate a real world situation. Very often a database is queried about just one or a few entries. The rest of the sub experiments simulate a growing load to the DBMS.

The query for the Insert operation is shown here:

```
INSERT INTO contributors VALUES (val[0], val[1], val[2], 'val
   [3]', val[4], 'val[5]', val[6], val[7], val[8])
```

The values are extracted from the columns arrays previously prepared in the setup phase. For the Ex2 to Ex11 a loop is used to create the desired number of queries. The queries are entirely buffered and the execution is started only when the buffering process is finished.

Exp.	% of table selected
Ex1	0.001% (1 row)
Ex2	10% (100 rows)
Ex3	20% (200 rows)
Ex4	30% (300 rows)
Ex5	40% (400 rows)
Ex6	50% (500 rows)
Ex7	60% (600 rows)
Ex8	70% (700 rows)
Ex9	80% (800 rows)
Ex10	90% (900 rows)
Ex11	100% (1000 rows)

Table 5.3.: Select experiment setup.

5.2.2. Select Experiment

The Select experiment is subdivided in three scenarios: the first scenario selects rows utilizing a tablescan, the second selects utilizing an indexscan on the univocal index on column ID and the third utilizes an indexscan on the random univocal index on column rID. The ID follows the insertion order of the rows from 1 to 1000; the rID has the same range but is randomly distributed over the entire table. The dataset is composed by the first 1000 rows of the Contributors table and the ID and rID columns are utilized for indexing. The first phase is the setup of the table, inserting the 1000 rows. This is very similar to Ex11 in the Insert experiment with the following query:

```
CREATE TABLE contributors (id, rid, account_id, account_name,
    main_language_id, main_language_name, man_months, commits
   , median_commits) OPTIONS (univocalindex:id, univocalindex
   :rid, dstrange:1000, blocksize:10)
```

The DST range is set to the dimension of the table, to minimize the number of DHT operations for DST; the block size is purposely small to augment the traffic of blocks. In this way, when more blocks are created, more DHT operations are needed.

The scenarios are organized in 11 sub experiments, summarized in Table 5.3. Each sub experiment is separated by a 2 seconds break. It starts by selecting one row from the table, like in the Insert experiment to simulate a real world situation. Then, 10%, 20%, ..., to 100% of the table is selected, utilizing a condition on the ID column. The select query is as follow:

```
SELECT * FROM contributors WHERE id <= numOfRows OPTIONS (
   tablescan/indexscan)
```

The WHERE condition is changed depending on the sub experiment, giving the number of rows to select. The tablescan option is used for the first scenario, the indexscan for the second and the indexscan on rID for the third.

Projets dataset for 100 matches:

```
94,932,3868,GNU Autoconf,749,3.31933,11848425,8,Perl,38351
95,566,3265,Amarok,744,4.54737,11817905,44,C++,264833
96,509,7278,GNU binutils,739,4.31532,11915386,42,C,2687113
97,99,4518,Postfix,719,4.43787,0,0,null,0
98,424,4145,Cygwin,715,4.05,11876081,42,C,1397863
99,953,14157,Facebook Plugin for Pidgin,715,3.94872,11664015,42,C,6037
100,787,155,LAME (Lame Ain t an MP3 Encoder),698,4.56098,11862169,42,C,116710
1101,1784,3867,GNU Automake,690,3.38281,11888629,11,shell script,111008
1102,1900,647,CVS: Concurrent Versions System,681,2.87234,11876003,42,C,267186
1103,1635,3437,Eclipse Web Tools Platform (WTP),676,3.97692,11808543,5,Java,10375577
1104,1120,3991,Gentoo Linux,672,4.67626,11334389,49,Ebuild,6623300
```

Figure 5.2.: Projects dataset for Ex2 with a selectivity of 100. The first two columns represent ID and rID. Only IDs lower or equal 100 will have a match to the joining table, all the other are set bigger than 1000.

The results coming inside the ResultSet of the *executeQuery()* statement are saved in a temporary Set. The size of the Set is then controlled at the end of each sub experiment. If the size corresponds to the number of selected rows of the given experiment, the query was successful executed.

5.2.3. Join Experiment

The Join experiment is similar to the Select experiment, except that in this case two tables are joined on a column. The experiment is also subdivided in three scenarios: tablescan, indexscan on ID and indexscan on rID. The Contributors dataset utilized in the Select experiment is the first table for the Join. Also in this case, the experiments are subdivided in 11 sub experiments, with the same selectivity as shown on Table 5.3. For a Join, however, the selectivity represent how many matching rows can be found in the second table, in other words it is given by the dataset. For example, Ex2 is going to return 100 joined rows and no more. Therefore, 11 Projects datasets had to be created with an appropriate ID and rID column that reflects the selectivity. For example, the Projects file of Ex2 presents the first 100 rows with an ID and rID below 1000, which can be found also in the Contributors table, resulting in 100 matches. The remaining IDs are bigger than 1000 and therefore not matchable. An extract of the Projects file for Ex2 is given on Figure 5.2.

The first phase of the Join experiment is therefore a long setup phase where the Contributors table and 11 Projects tables have to be loaded in the database, for a total of 12 different tables and 12'000 rows inserted. This is at the same time a good stress test for the implementation. The sub experiments are then executed, with a 2 seconds break between each, with the following query:

```
SELECT * FROM contributors, projectsX WHERE contributors.id =
    projectsX.id OPTIONS (tablescan/indexscan)
```

The projects table is changed depending on the experiment executed. For instance, projectsX would be projects100 for Ex2. The tablescan option is used for the first scenario,

the indexscan for the second and third. In the third scenario, the join is set on the rID column instead of the ID.

Similar to the Select experiment, the results coming inside the ResultSet are saved in a temporary Set. The size is then controlled at the end of each experiment to check if it corresponds to the expected joined rows number.

5.2.4. Delete Experiment

The Delete experiment is quite different from the others and is focused on testing the insertion order and full blocks storage types of TomDB. The insertion order storage type adds new incoming rows always at the end of the table, keeping the insertion order of the rows. The full blocks storage, however, reutilizes first empty sectors of previously deleted rows in the existing blocks. The performances of both storages are therefore different in different scenarios. The table used is the same Contributors table of the Select experiment, loaded in the database during the setup phase. The idea is to first delete half of the table, with the query:

```
DELETE FROM contributors WHERE rid < 500
```

The rID is used so that blocks with empty rows are spread all over the table. Then, 500 rows are reinserted in the table with the process seen in the Insert experiment. At the end, the entire table is retrieved with `SELECT * FROM contributors`.

This process is executed 10 times with the insertion order storage and 10 times with the full blocks storage, changing the OPTIONS setting for the storage in the CREATE TABLE statement; the block size is set to 10.

5.3. Results

The information are collected with many log messages inserted in the implementation, so that the execution time and the DHT operations utilized by a statement can be measured. The data are then aggregated in MySQL and saved in Excel sheets. For the execution time, a regression is calculated with the sheet, to be utilized as a trend line inside the charts. The elaborated information are saved in text files and the charts are created utilizing Gnuplot.

The experiments are executed 10 times separately for every scenario inside the test bed. The scenarios, comprehending the sub experiments, run automatically to the end. After every execution, the log file is saved and the DHT network is shut down and restarted to have a clean DHT, ready for the next execution.

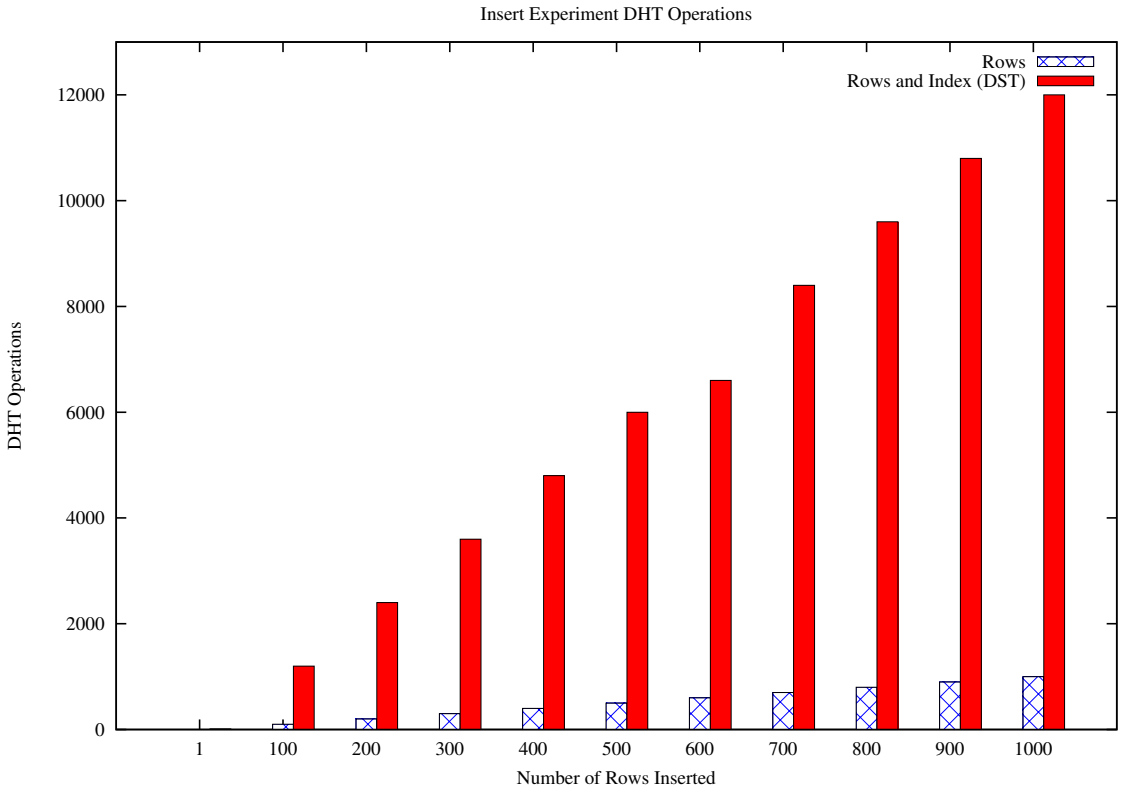

Figure 5.3.: DHT operations needed by the Insert experiment.

5.3.1. Insert Experiment

The Insert operation is an unavoidable function of a DBMS, for example to create the table for the other tests. In this thesis, however, the design focus was not on optimizing the Inserts, but rather the Selects. Even if the insert queries are buffered and executed all together, the DHT operations are not aggregated. In other words, every row inserted needs its set of DHT operations.

For the experiment without indexes, the Insert operation merely consists of one put operation per row, i.e. for example to insert 1000 rows, 1000 DHT operations are performed. This is evident also on Figure 5.3. The DST index in the second experiment, however, is causing a high overflow of operations, because every new indexed value needs to be put on each level of the tree. For example for a DST range of 10'000 this means a tree height of 11 ($ln(10'000) + 1$). The consequence of indexing for the Insert operation is therefore that, to insert for example 1000 rows, 1000 puts are executed for the DHT table and 11'000 puts for the DST tree, for a total of 12'000.

The DHT operations are the highest cost in this application because the routing algorithm goes through a number of DHT peers equals to $log(peers)$ to get or put a data. Given that many peers are not on the same machine, the DHT utilizes the network to communicate, which is much slower that the main memory. Therefore, the DHT operations are a good parameter to calculate the cost of an entire operation and they have high influence also on the execution time. It is no surprise that the execution time for the experiment with

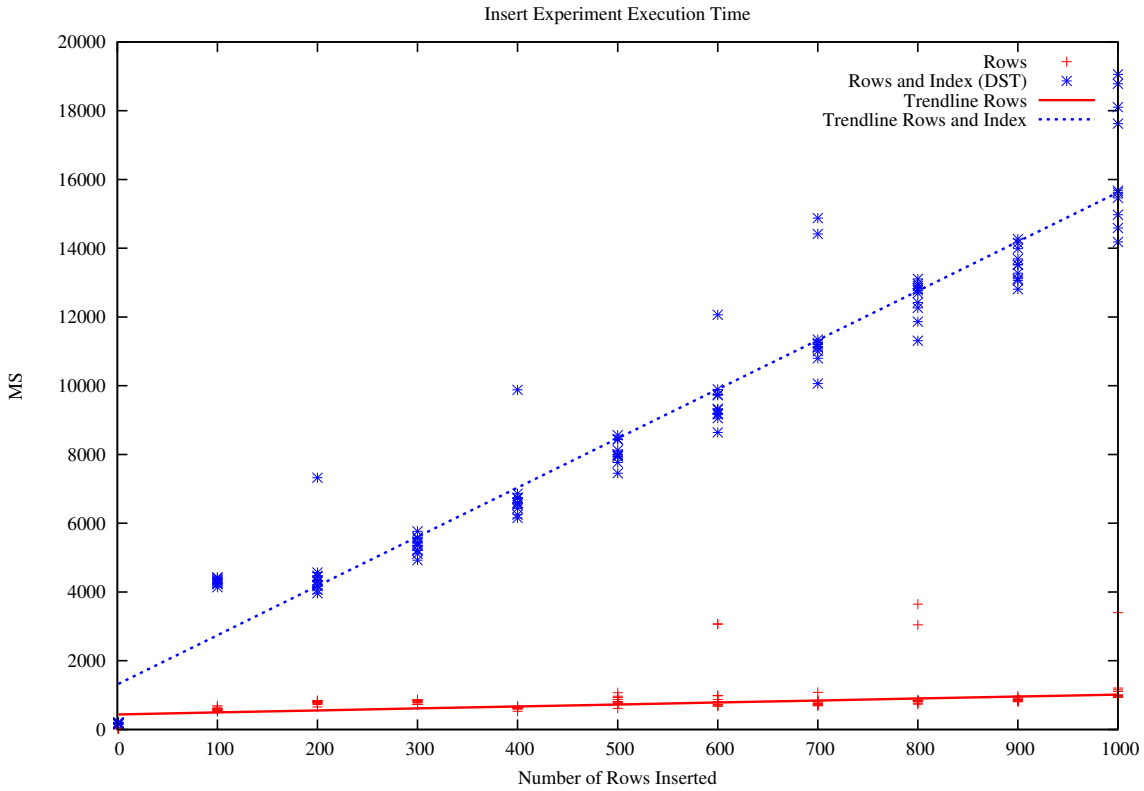

Figure 5.4.: Execution time of the Insert experiment.

index is much higher, as seen on Figure 5.4. For the experiment without index the median execution time goes from 37ms for inserting one row to 1s for inserting 1000 rows; by the experiment with index, the median execution time for inserting one row is 167ms and for inserting 1000 rows 15.7s.

5.3.2. Select Experiment

The Select operation is an important statement for many applications. Therefore, the focus on optimization was put on Select operations. This motivated also the choice of DST as an indexing structure, because it performs very efficient for Select operations, with the cost of a poor performance by Insert statements.

Figure 5.5 shows the DHT operations. A tablescan consists in retrieving the entire table, giving a constant result of 100 operations for the 1000 rows in the table of the experiment. This can be calculated dividing the table size by the block size, for instances here $1000/10 = 100$ *Blocks*, one block correspond to one DHT operation. The indexscan, on the other hand, first performs a lookup on the index of the indexed column to identify the interesting table blocks and then retrieves the blocks form the DHT. The experiment that performs the indexscan on the ID column starts with 2 operations to retrieve 1 row and grows linearly to 108 operations to retrieve the entire table. This is given because the ID column is set following the insertion order; to retrieve for example 100 rows the first 10 blocks of the table are retrieved plus some operations for the DST, to retrieve

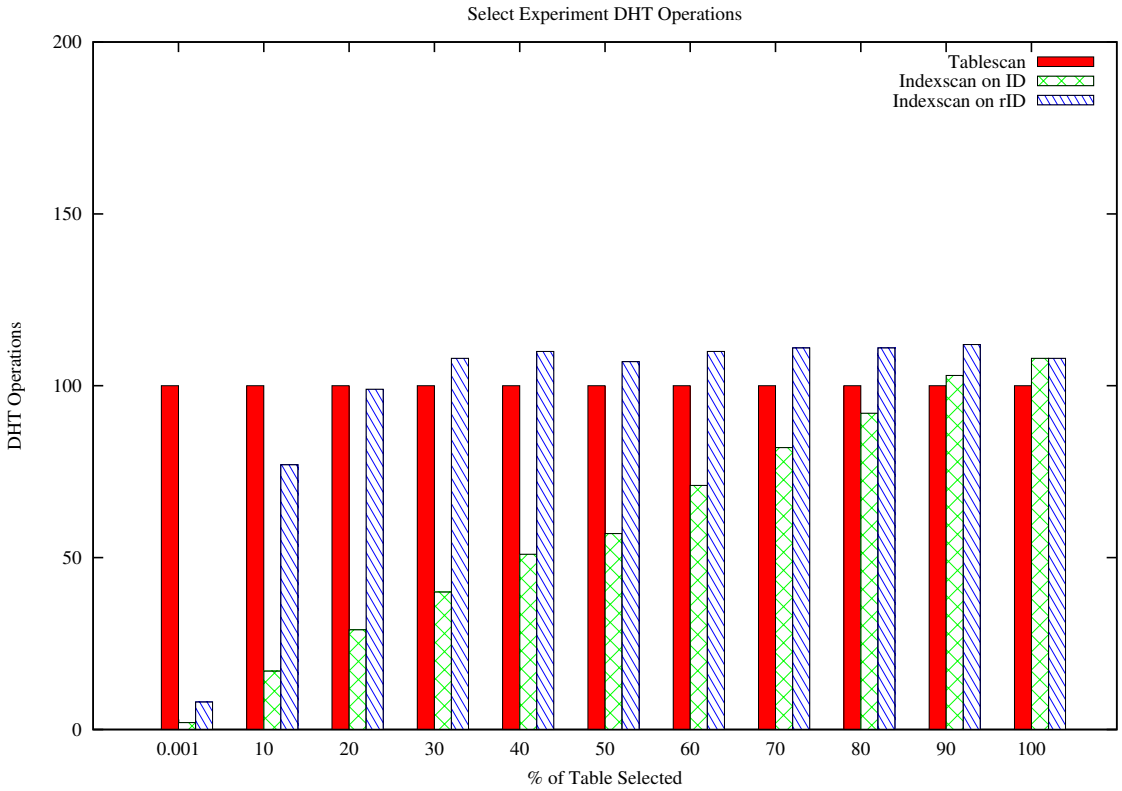

Figure 5.5.: DHT operations needed by the Select experiment.

200 rows the first 20 blocks are retrieved and so on. This corresponds to the best case. Completely different is the situation for the indexscan on the random distributed rID. Already for retrieving 300 rows, the operation needs 108 gets, meaning that the entire table is retrieved. This is also the expected behavior. In this case the rID do not follow the insertion order, however the IDs are randomly spread over the entire table. In other words, the first 300 rows that the query is looking for are not saved in consecutive blocks but distributed on the entire table. This was intended to simulate the worst case scenario for the indexing structure. Interesting to note is the overflow caused by the index lookup, namely the number of operations needed by the lookup, over the retrieving of the table blocks. This is calculated over all the executed experiments with a median value of 10 DHT operations for the index lookup.

The execution of the Select experiments encountered two types of problems; the first problem is a higher execution time of the first tablescan experiment. This is partially explained by the fact that this is also the all-first operation of the entire experiment and therefore, the table metadata are updated from the DHT, which is a blocking operation. The second problem is a much higher value for the execution time of certain sub experiments that happened occasionally. In other words, one or more DHT operations are blocked and respond only after a high delay, up to seconds. All this events have been flagged as outliers and excluded from the calculation of the regression.

In the chart on Figure 5.6 the execution times of all the three scenarios are shown, together with the trendlines. The first interesting aspect to be noted is the tablescan trendline,

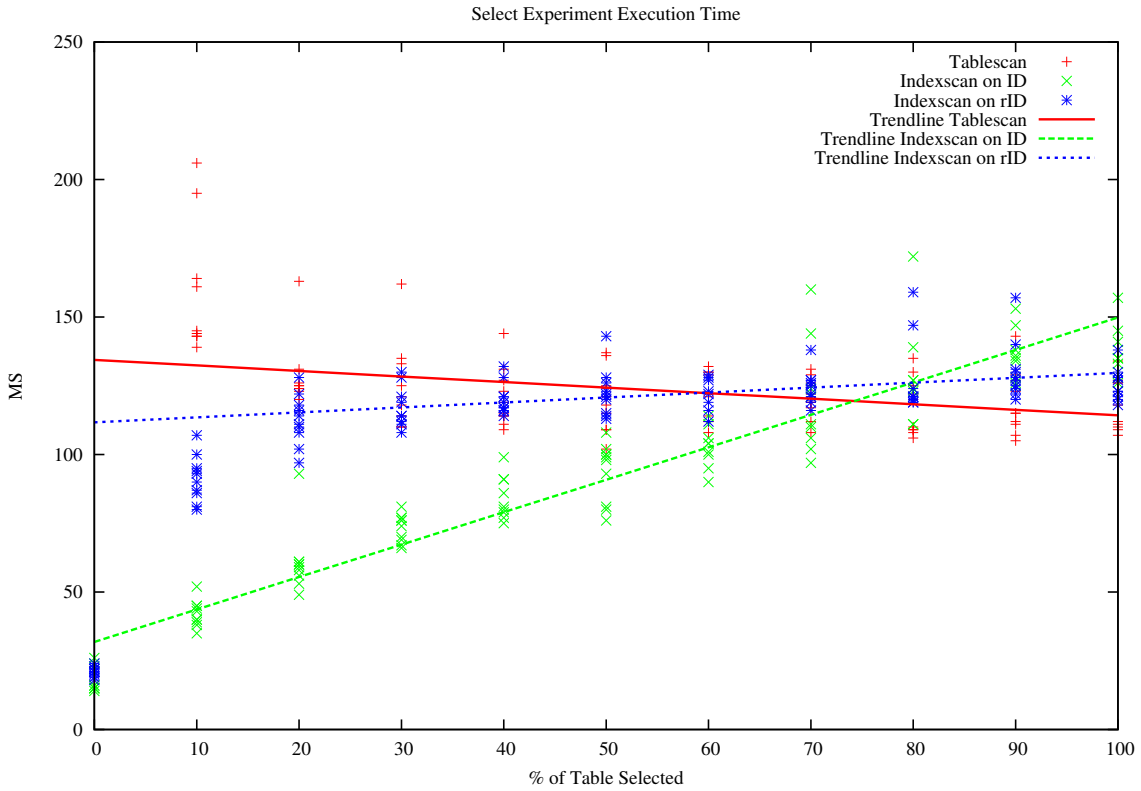

Figure 5.6.: Execution time of the Select experiment.

resulted by the first scenario experiment. Even if this test executes 11 times exactly the same operation, i.e. downloading the entire table, the line leans negatively. Given a median value of 124ms for the tablescan, the last sub experiments perform effectively faster. No definitive explanation could be given to this behavior, but it may lie in a caching or optimizing mechanisms of the DHT. The median execution time for the tablescan is influenced by the block size. For example, if the block size would be set to 100 for this experiment, only 10 blocks would be needed to retrieve the entire table. The highest cost in DHT is the routing to the interested peer. In other words, the setup of the connection to the responsible peer of a block costs; the transport of the block to the peer, on the other hand, is fast on the Ethernet. This suggests that bigger blocks are better, because fewer connections have to be created, which is to certain extent true. A problem could be, however, a congestion on peers that are responsible for highly requested blocks of the table, because bigger blocks are less distributed on the DHT.

The second scenario, the indexscan on ID, behaves as expected from the operations chart, growing linearly. It starts with a median value of 16ms to select one row and ends with a value of 138ms to select the entire table. The 16ms value confirms that indexes are a great instrument to optimize queries. The intersection point between the indexscan (green) and tablescan (red) trendlines in the chart denotes the selectivity %, about 75% of the table, where a tablescan performs better than the indexscan. In other words, when more than 75% of the table is selected by the query, a tablescan is probably faster. This can effectively happen, because the indexscan has an overflow, i.e. more DHT operations are executed to do the index lookup. This value, however, must be relativized seeing that

the median execution time to retrieve the entire table by an indexscan is 138ms, just 14ms more than the tablescan. The low overflow of DST is possible until the DST range remains bigger than the range of the indexed values. In this situation, the DHT operations for the index lookup are minimized. When the DST range is smaller than the indexed values range, however, full blocks are present in the index and the DHT operations increase to get the child blocks. This would mean a shift of the intersection point to the left.

To calculate the regression of the third scenario, the indexscan on rID, the first two sub experiments, Ex1 and Ex2, are excluded, because they perform with much less DHT operations compared to the other sub experiments, making the trendline too leaning for the scope of the experiment, which was intended to demonstrate the worst case scenario. The resulting trendline is nearly flat and, if the tablescan trendline would not manifest the descending behavior, they would be parallel. The execution times are similar to the tablescan. Indeed, to retrieve the entire table, more than 100 DHT operations are needed. The real intersection point is expected to be at 20%, where the DHT operations exceed the operations for tablescan. This worst case scenario was designed to show the limits of the indexing structure. The effectiveness of the indexing depends on the data indexed and does not show for every data the same behavior.

A comparison of the results to the experiments conducted in [1] is difficult, given completely different datasets. The speedup factor of the indexscan compared to the tablescan, which is relative, can however be compared. The execution time to select 25% of the table in [1] corresponds to a speedup of 75%; in the experiments conducted in this thesis, the same factor is 46%. The general impression is, however, that the tablescan of TomDB performs better; to select 1000 rows in this thesis, the median value is 124 milliseconds, to select 2000 rows in [1], the average is 12 seconds, which corresponds to 6 seconds for 1000 rows. This is probably given by the fragmentation of this thesis, where blocks of n rows are saved in the DHT. In [1], instead, every rows represents a *key/value* pair.

5.3.3. Join Experiment

The Join operation for the tablescan consists in downloading both tables completely. For the indexscan, both indexes are downloaded first and then the matching rows, found comparing the indexes, are downloaded from the DHT. This is reflected also on the chart in Figure 5.7, where the DHT operations are merely doubled compared to the Select experiment for the tablescan, resulting in 200. The indexscan on ID follows a growing linear trend, similar to the Select experiment, starting with 19 operations to join one row up to 216 to join the entire table. This is the best case scenario. The indexscan on rID, however, is quite different that in the Select experiment and shows a growing trend similar to the indexscan on ID. This is given by the Projects dataset, seen in Figure 5.2, where also the rIDs follow the insertion order. In other words, when the matching IDs are found through the comparison of the indexes of both tables, the retrieving of the rows from the Projects table is executed exactly like it happens for the indexscan on ID. Only the Contributors table shows a completely random ID and augments the needed DHT operations for the third scenario. With these datasets was therefore not possible to show the worst case for indexscan, like it happened in the Select experiment.

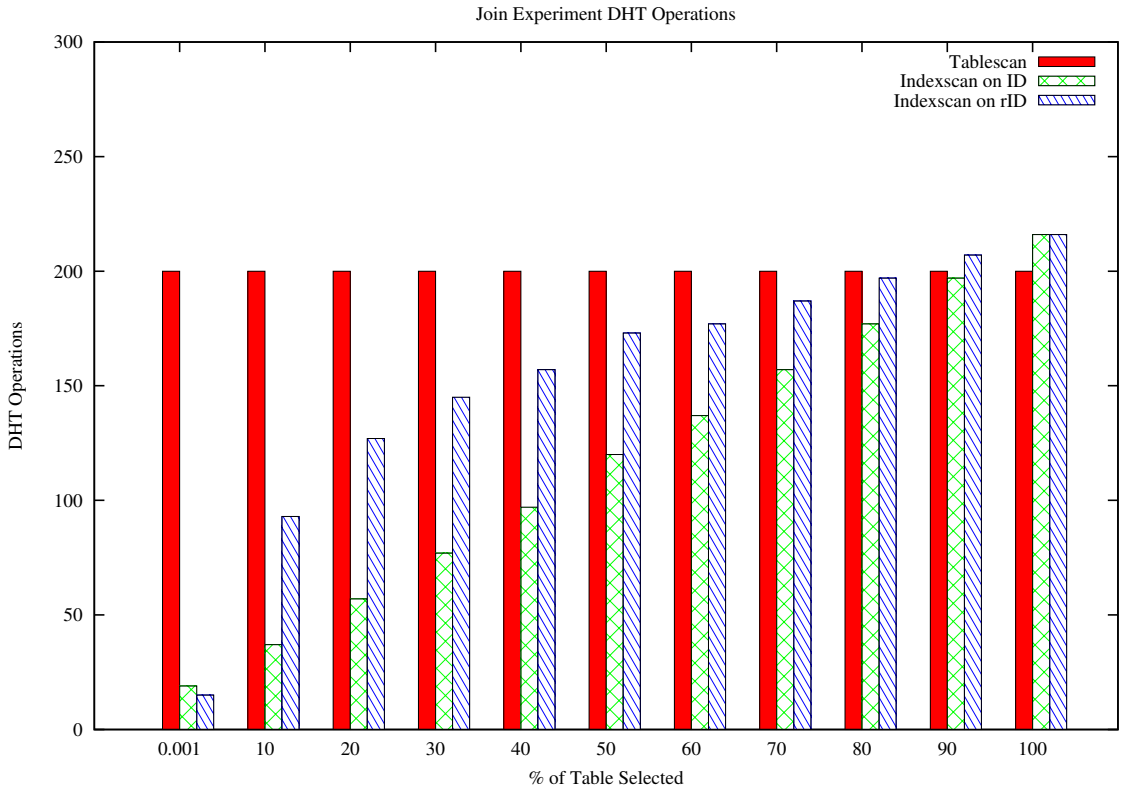

Figure 5.7.: Operations needed by the Join experiment.

Also in Figure 5.8 the data manifest the same two problems found in the Select experiment: higher values in the first sub experiment of tablescan and strange high values in the execution times occasionally, because of blocking calls. These outliers are excluded from the calculation of the regression. The trendline of the tablescan also shows a slightly descending trend as seen in the Select experiment. The median execution time of a tablescan is 280ms, 32ms higher than the doubled time of the Select operation. This is probably given by the fact that the Join implementation has to wait until every asynchronous get returns the results before the joining can begin, unable to fully exploit the advantages of asynchronous operations.

The trend for the second scenario, the indexscan on ID, grows linearly similar to the Select experiment. It starts with an execution time of 166ms for the first sub experiment and ends with a value of 302ms to join the entire table. This last value is 26ms higher than the doubled time of Select. The explanation is similar as above for the tablescan. It is also just 22ms higher than the tablescan. In fact, the intersection point in this experiment is about at 90% of selectivity. In other words, the indexscan approach is nearly always faster than the tablescan. This is, as explained in the Select, given by the ideal dataset utilized in the experiment. The trendline of the third scenario (blue) is parallel to the indexscan on ID (green), as already discussed in the paragraph above. The shift upward is given by the fact that the Contributors table needs to be retrieved completely, like it happened in the Select experiment, because the searched rows are spread over the entire table.

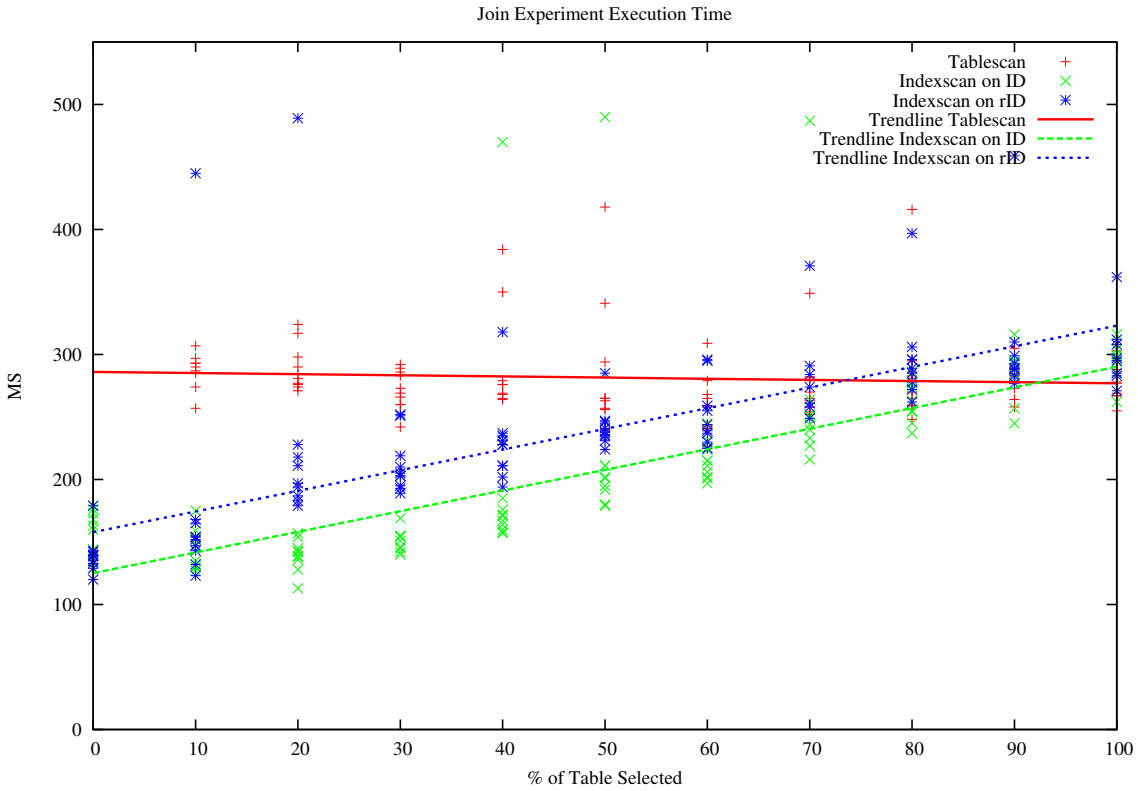

Figure 5.8.: Execution time of the Join experiment.

5.3.4. Delete Experiment

The Delete test was designed to verify the following scenario: when a DELETE query removes rows all over the table, every block on the DHT contains empty rows and unassigned *row IDs*. A SELECT query over this table is less efficient, because non-full blocks are transported, i.e. more blocks are needed to cover the same range of rows compared to full blocks. With the insertion order storage this problem is not solved because the new inserted rows are attached at the end of the table in new blocks. The full blocks storage, on the other hand, reutilize first the empty *row IDs* in the existing blocks. These scenario is reflected also in the results of the experiment on Table 5.4: after that 500 rows are deleted and then reinserted in the table, to retrieve the entire table the insertion order storage needs 150 DHT operations and the full blocks storage only 100. This is confirmed also by the execution times, where the insertion order takes 311ms compared to the 212ms of full blocks, as seen on Figure 5.9. Deleting with full blocks storage needs 601 operations, because it uses one operation to save the resulting free blocks on the DHT.

There is situations, however, where the insertion order storage performs better, because the insertion order of the new rows is maintained. For example in the scenario above, a query to SELECT only the last 500 new rows inserted would need only 50 operations with insertion order. This is given, because the Insert operation is going to create 50 new blocks at the end of the table. In full blocks storage, on the other hand, the 500 new rows are spread all over the existent blocks. To select the last 500 inserted rows, the entire table, i.e. 100 blocks/operations, is necessary. Also in this case it depends on the data

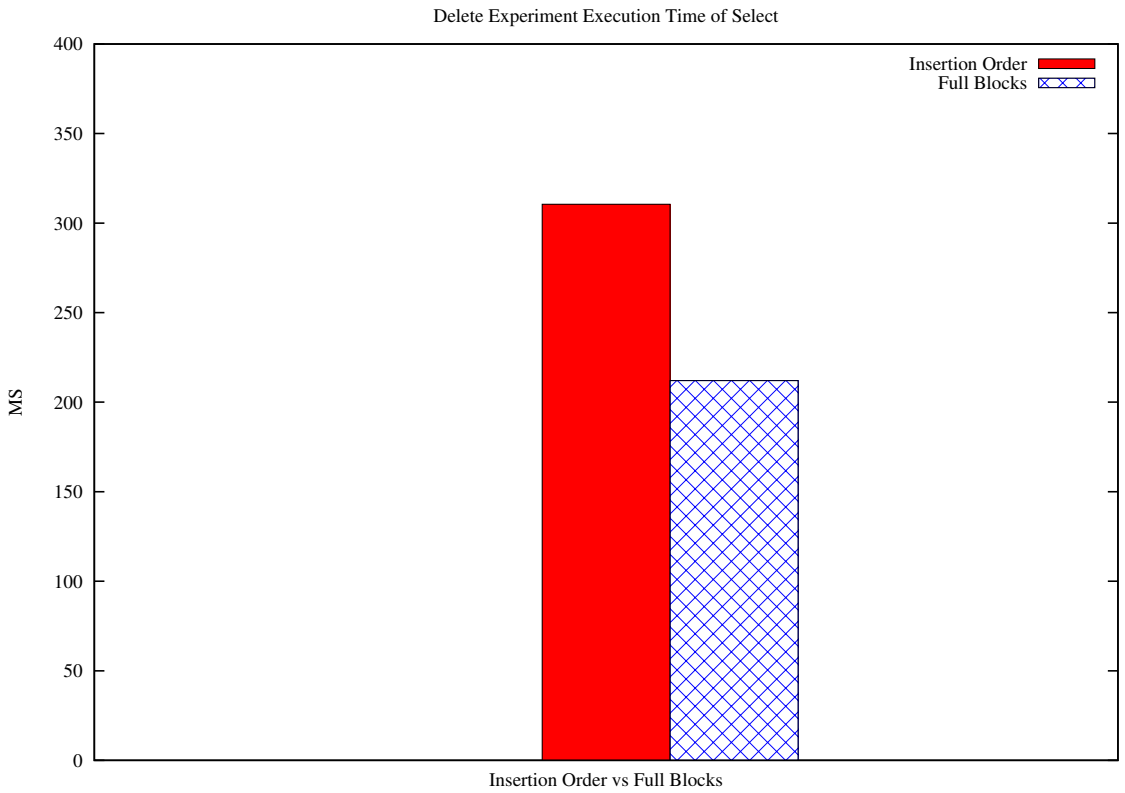

Figure 5.9.: Insertion Order vs Full Blocks Select execution time.

Insertion order	Delete	Insert	Select
ms	1252	582	311
operations	600	500	150

Full blocks	Delete	Insert	Select
ms	1158	689	212
operations	601	500	100

Table 5.4.: Delete experiment median results.

and the application to define a real winner between the two storage types.

6. Summary and Conclusions

6.1. Summary

NoSQL databases are arising as an important section of the database world. With a simple interface and the reject of typical relational DBMS properties, NoSQL approaches process big data with a high scalability and performance. One application is the *key/value* pair paradigm, used for example by DHTs. On the other hand, many business, financial and industries continue to prefer relational DBMSs for their applications.

TomDB was implemented as a relational DBMS engine capable of mapping the relational model and SQL queries to *key/values* pairs and DHT operations. TomP2P is used as the underlying DHT and is completely integrated inside the DBMS. The SQL parser supports the following commands: CREATE TABLE, INSERT, SELECT with condition and joins, UPDATE and DELETE. An indexing structure was implemented to meliorate the query performance, utilizing a Distributed Segment Tree. The application is designed as a standalone library, utilizable through a JDBC-like API.

The experiments evaluate the main functionalities of the DBMS: INSERT, SELECT, JOIN and DELETE. The data utilized for the execution of the queries are fetched from the Ohloh.net website and adapted to the experiments.

6.2. Conclusion

The proof of concept for TomDB was a success and returned encouraging results. With the given dataset it was possible to select 1000 rows in 124ms and 1 row, utilizing the index, in 16ms, on a DHT network of 6 machines and 1002 peers. Joining two tables performed with about the doubled time as a Select operation, which was expected. The drawback was given by the INSERT operation that is not completely optimized and by the indexing through the DST, which causes more DHT calls by an order of magnitude than without index.

The first goal, however, remains open: is TomDB really a relational DBMS on top of a DHT or is it just a SQL capable DHT? The ACID properties of DBMSs could be taken as an objective to define an application as a real relational DBMS. The answer could be therefore given analyzing the properties:

- *Durability*: This property is an important characteristic of DHT and P2P systems in general; when even just one peer remains alive on the network, the information are

kept. DHT networks, as self-organizing entities, handle adding and leaving of peers transparently, moving the data to the responsible peer and keeping replications, without the need of configuration. This characteristic was one of the drivers of this project.

- *Consistency*: On one hand, a *key/value* pair on a DHT is for every user consistent and exactly the same. Only the exact key match can retrieve that information. This guarantees the consistency of the *key/value* pairs on the DHT but it is not a promise that the saved data inside a *key/value* is correct. In the Future Works on Chapter 7 examples are given, where inconsistent data could be produced with TomDB.

- *Isolation*: This term is related to concurrency control. In P2P networks without a central authority, as stated in Chapter 7, a complete concurrency control is difficult to reach. Nevertheless, the execution of certain query, such as INSERT or UPDATE, must be guaranteed conflicts free, otherwise inconsistent data are saved in the database. On the other hand, for a SELECT query, the concept of *eventual consistency* could be applied. In other words, it is not absolutely necessary that always all the retrieved data are up-to-date. A concurrency control mechanism is, however, not part of TomDB.

- *Atomicity*: This property is concerned about the correct execution of transactions, "all or nothing", to avoid partial transactions to remain in the database. A reverse mechanism to keep trace of a transaction and cancel it if necessary has not been implemented.

The analysis gives a clear picture; TomDB is only an SQL capable DHT. Seeing that the ACID properties seem a distant goal for a DHT based relational DBMS, a new concept, more appropriate for NoSQL systems, could be analyzed; BASE:

- *Basic Availability*: The focus is on the availability of data, even in presence of failures. A DHT is explicitly designed for this, the self-organization and replication guarantees that the data are available even if peers leave the network.

- *Soft State*: the consistency requirements of ACID are abandoned. The state can change over time. In TomDB for example, a soft state is present during INSERT, DELETE and UPDATE operations. During these periods, the other users are not informed that changes are going on in the database.

- *Eventual Consistency*: The consistency is given at a certain point in the future but the data retrieved are not guaranteed to be up-to-date. In this thesis, however, a conflict between operations could lead to inconsistencies. A mechanism to guarantee the consistency, even if not in real time, could be a first interesting future development for TomDB.

The objectives of this thesis to proof the feasibility and performance of a relational DBMS on top of a DHT have been achieved. The results of the experiments show the potentiality of the proposed approach and is the starting point for future work. The proposed directions in this thesis are an inspiration to further develop this challenging topic.

7. Future Works

This thesis is a proof of concept and therefore the implementation is simplified to the minimal necessary functionalities to execute the experiments. To reach the capabilities of a relational DBMS, however, more functionalities and optimizations are required.

7.1. SQL Parser

The SQL parser supports only a small subset of the SQL standard; therefore, it should be extended to support all the language. The manual implementation chosen in this thesis shows its limits and becomes difficult to extend for more commands; the probability of making mistakes and oversights increase constantly with the complexity. Another approach should be choose, using for example a tool like ANTLR to compile a correct parser. The implementation of the SQL operations needs to be abstracted and extended to elaborate the entire set of SQL commands. For example the ConditionsHandler supports only the prioritization of AND and OR conditions from left to the right, without considering parenthesis.

7.2. Optimizations

Although it was a clear design principle from the beginning, there are still some blocking DHT operations inside TomDB, one over all the index lookup in IndexHandler. This last example revealed to be problematic when thousands of rows are consecutively inserted in the table, causing inconsistencies in the index. A clear architecture that eliminates all the blocking operations should be targeted in a future development. Also the design principle of minimizing the DHT operations has still potential for optimization. One example is the INSERT operation: every single new row utilizes a put operation. In TomP2P it would be possible to aggregate the new rows, i.e. the *content keys*, in a Map locally and put the entire Map, i.e. the *location key*, with only one operation.

7.3. Query Optimizer and Block Size

One of the most interesting and powerful capabilities of a traditional DBMS is the Query Optimizer, a function that is capable of deciding the path for executing a query in the most efficient way, for example choosing the type of scan (index or table). A possible

approach could be to create a quantity of statistical metadata about an indexed column, like the distribution of the data, and utilize this information to determine how much of the table a query is going to select. The selectivity of a query could determine the choice between indexscan and tablescan, calculating for example the number of DHT operations that the scan needs or based on empirical data.

Another variable in TomDB that has to be set manually is the size of a table block, n. For a table with many columns, this size should be small, the opposite for a table with a few columns. The network in use or the memory of a peer could also determine n. Moving a block on the Internet is probably slower than moving it on a LAN. An accurate study of these behaviors could produce an algorithm to automatically choose n, adding a further feature to the application.

7.4. DST

The DST range is another variable that is set manually in TomDB. In this case, however, it is difficult to predict how much a table grows, i.e. how large the DST range should be, to calculate it automatically. Moreover, the static structure of DST does not allow to dynamically changing the range at a later time. DST reveals to be efficient for SELECT statements, causing just a few DHT operations. By inserting new rows however, the overflow of DHT operations is very high, calculated in $ln(DSTrange) + 1$ for every new indexed value. In other words, DST may not be the best choice. It remains the best approach found in the literature that is entirely based on the standard DHT API, meaning that the research of alternatives should focus on a technology that is build inside the DHT and not on top of it.

7.5. Concurrent Users

One important characteristic of relational DBMS is *isolation*: the transactions on the database are submitted to concurrency control. In other words, if two users utilize a database simultaneously on the same data, an INSERT transaction, for instance, blocks the access to the part of table that is going to be written. In TomDB, however, a locking mechanism is not present. Problems could arise for example when two users try to insert new rows at the same time. The metadata are updated on both instances, the new *row ID* is extracted from the metadata, but it is the same for both users. When the put operations are started, the last operation to finish is going to overwrite the other row, because the *content key* is the same, causing inconsistent data in the table. Other problems arise when two users try to DELETE or UPDATE the table and at the same time the data are selected. If the SELECT operation happens before the UPDATE or DELETE, the received data are inconsistent because not actualized. The same could be said for a SELECT operation before a new INSERT is finished. Many other conflict scenarios could be listed.

Concurrency control is a great challenge in P2P systems like the DHT. There is no central control to drive a locking mechanism and broadcasting is not reliable and causes a huge

communication overflow. Also a distributed locking registry is probably not practicable because not efficient, it would produce an overflow of blocking operations inside the system. In a P2P based DBMS it is probably necessary to apply a more permissive approach like it happens by Amazon Dynamo [17]: *eventual consistency*. The data are guaranteed to be inserted, updated and deleted, but there is no guarantee that a SELECT operation returns the most updated information.

A feature that goes towards an *eventual consistency* in TomDB could be reached utilizing the metadata: the INSERT, UPDATE and DELETE operations always actualize the metadata before proceeding. The metadata could be flagged as locked for a given operation by a user and when the operation complete and the actualized metadata is uploaded in the DHT, the flag could be removed. In this way, when another user tries to execute a conflicting operation at the same time, it would be informed about the conflict before starting the process. The problem still remains when the metadata fetching operations happen so close to each other that the flagged metadata is not saved yet on the DHT.

7.6. Atomicity of data

Assuming an *eventual consistency* approach, the *atomicity* of transactions is even more important. The INSERT, DELETE and UPDATE operations must guarantee to produce consistent data. However, in TomDB there is no mechanism to revers an unsuccessful query. Especially when different operations are involved, like for example inserting in the table and the index or an update of the metadata, if one operation fails, the others operations are likely to be executed anyway. This produces inconsistent entries in the database.

To implement the *atomicity* of transactions, a registry of the operations should be kept until the entire process is finished correctly. When a DHT operation fails, a mechanism could be triggered to cancel all the other involved operations and bring the database state back to the beginning.

Bibliography

[1] Eduardo A Ribas, Roney Uba, Ana Paula Reinaldo, et al. Layering a dbms on a dht-based storage engine. *Journal of Information and Data Management*, 2(1):59, 2011.

[2] S. Sumathi and S. Esakkirajan. *Fundamentals of Relational Database Management Systems*. Studies in Computational Intelligence. Springer, 2007.

[3] Carlo Curino, Evan Jones, Yang Zhang, Eugene Wu, and Samuel Madden. Relational cloud: The case for a database service. Technical report, Massachusetts Institute of Technology, 2010.

[4] Thomas Bocek. Tomp2p. http://tomp2p.net/, last visited: December 2013.

[5] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed segment tree: Support of range query and cover query over dht. In *IPTPS*, 2006.

[6] Brandon Wiley. Distibuted hash tables. *Linux Journal*, (114), October 2003.

[7] The Wikipedia Authors. Hash table. http://en.wikipedia.org/wiki/Hash_table, last visited: December 2013.

[8] Aaron Kaluszka. Distributed hash tables. 2010.

[9] The Wikipedia Authors. Tomp2p. http://en.wikipedia.org/wiki/TomP2P, last visited: December 2013.

[10] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[11] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill higher education. McGraw-Hill Education, 2003.

[12] P. Beynon-Davies. *Database Systems*. Macmillan computer science series. Palgrave Macmillan Limited, 2004.

[13] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational geometry*. Springer, 2000.

[14] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M Hellerstein, and Scott Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, 2004.

[15] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, volume 7, pages 205–220, 2007.

[18] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[19] Oracle Corporation. Mysql cluster. http://www.mysql.com/products/cluster/, last visited: December 2013.

[20] LLC VoltDB. Voltdb technical overview, 2010.

[21] Google. Google cloud sql. https://developers.google.com/cloud-sql/, last visited: December 2013.

[22] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea Kyle Littlefield, David Menestrina, Stephan Ellner John Cieslewicz, Ian Rae, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11), 2013.

[23] Oracle Corporation. Java database connectivity api. http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/, last visited: December 2013.

List of Figures

List of Tables

A. Installation Guidelines

In order to utilize the TomDB library, Java JDK 7 is required.

The TomDB Jar can be executed as a standalone software from the command-line to act as a DHT peer with the following parameters:

- Address of the bootstrapping peer.
- Number of local peers to create.
- True/False if a random port should be choose instead of the default port 4000.

Otherwise, the library can be imported as an external library in a Java project and the JDBC-like API can be utilized inside the application. The self-contained Jar with dependencies do not require further libraries to be imported.

B. Contents of the CD

On the root level of the CD there is the thesis as PDF file, a german and english version of the abstract in plain text files and the presentation slides in PDF.

B.1. TomDB

This folder contains the compiled Jars including dependencies for TomDB (TomP2P 4.4 library): TomDB-1.0-jar-with-dependencies.jar and TomDB5 (TomP2P 5 Alpha library): TomDB5-1.0-jar-with-dependencies.jar

B.2. Data

This folder contains all the data produced with the experiments subdivided in the folders Insert, Select, Join and Delete. Every folder contains the raw log files, the Excel sheet with the analyzed data and the gnuplot files to generate the charts.

B.3. Experiments

This folder contains all files needed to run the experiments and the Eclipse project of the Experiments.

In order to execute the experiments, a TomDB network must be started. Then, the Experiments.jar file can be executed from the command-line, with the bootstrapping peer address as parameter. The Jar should be in the same directory as the dataset files.

B.4. Related Works

Contains all papers used for writing this thesis.

B.5. Sources

This folder contains the Maven projects for TomDB and TomDB5.

B.6. Thesis

Contains the LaTeX sources of this thesis including images.