INFORMATIK
BERICHTE

377 – 05/2018

BBoxDB - A Distributed and Highly Available Key-Bounding-Box-Value Store

Jan Kristof Nidzwetzki
Ralf Hartmut Güting

FernUniversität in Hagen

Fakultät für Mathematik und Informatik
D-58084 Hagen

BBoxDB - A Distributed and Highly Available Key-Bounding-Box-Value Store

Jan Kristof Nidzwetzki
Ralf Hartmut Güting
Faculty of Mathematics and Computer Science
FernUniversität Hagen
58084 Hagen, Germany
{jan.nidzwetzki@studium.,rhg@}fernuni-hagen.de

May 2, 2018

Abstract

BBoxDB is a distributed and highly available key-bounding-box-value store, which is designed to handle multi-dimensional big data. The software splits data into multi-dimensional shards and spreads them across a cluster of nodes. In contrast to existing key-value stores, BBoxDB stores each value together with an n-dimensional axis parallel bounding box. The bounding box describes the spatial location of the value in an n-dimensional space. A space partitioner (e.g., a K-D Tree, a Quad-Tree or a Grid) is used to split up the n-dimensional space into disjoint regions (distribution regions). Distribution regions are created dynamically, based on the stored data. BBoxDB can handle growing and shrinking datasets. The data redistribution is performed in the background and does not affect the availability of the system; read and write access is still possible at any time. Multi-dimensional data can be retrieved using hyperrectangle queries; these queries are efficiently supported by indices. Moreover, BBoxDB introduces distribution groups, the data of all tables of a distribution group are distributed in the same way (co-partitioned). Spatial-joins on co-partitioned tables can be executed efficiently without data shuffling between nodes. BBoxDB supports spatial-joins out-of-the-box using the bounding boxes of the stored data. Spatial-joins are supported by a spatial index and executed in a distributed and parallel manner on the nodes of the cluster.

1 Introduction

Today, location-enabled devices are quite common. Mobile phones or cars equipped with small computers and GPS receivers are ubiquitous. These devices often run applications that use *location-based services* (LBS). For example, the navigation system of a car shows the direction to the next gas station or a restaurant shows an advertisement on all mobile phones within a 10-mile radius around itself. The data in this example is a position in the two-dimensional space. Data with other dimensions are also quite common. For example, the position of a car at a given time results in a point in three-dimensional space.

The challenging task for today's systems is to store multi-dimensional big data efficiently. Four major problems exist: (1) The data can change frequently (e.g., updating the motion vector of a moving car in a moving objects database [23]) and the data store has to deal with high update rates. (2) The datasets can become extremely large, so much so that a single node cannot handle them. (3) Multi-dimensional data should be efficiently retrieved (i.e., hyperrectangle queries should be efficiency supported). (4) For efficient query processing, related data of multiple tables should be stored on the same hardware node (co-partitioning). Performing equi-joins or spatial-joins are common tasks in applications and these type of queries have to be efficiently supported.

1.1 Key-Value Stores

In the last decade, *NoSQL-Databases* have become popular. To handle large datasets, many of them are designed as a distributed system. *Key-value stores* (KVS) are a common type of NoSQL-Databases, which focus on storing and retrieving data. Features such as complex query processing are not implemented. A KVS supports at least two operations: `put(key, value)` and `get(key)`. The first operation stores a new value identified by a unique key (a *tuple*). The key can be later used to retrieve the stored value with the get operation. Usually, the key is a string value and the value is an array of bytes.

In a distributed KVS, the whole dataset is split into pieces (*shards*) and stored on a cluster of nodes. Each node of the cluster stores only a small part of the data; this technique is known as *horizontal scaling*. As the dataset grows, additional nodes can be used to store the additional data. To enhance data availability, data are *replicated* and stored multiple times onto different nodes. In the event of a node failure, the data can be retrieved from another node.

When working with big datasets, a significant challenge is to distribute the data equally across all available nodes. Otherwise, some nodes can be over-utilized while other nodes remain under-utilized.

1.2 Multi-Dimensional Data in Key-Value Stores

A KVS stores a certain value under a certain key. The key has two roles in a distributed KVS: (1) it is used to identify a tuple clearly and (2) to assign a tuple to a certain node. *Range-partitioning* and *hash-partitioning* are used today to determine which node is responsible for which key.

(1) *Range-partitioning:* Ranges of keys are assigned to the available nodes. For example, *Node 1* stores all keys starting with a, b, c, or d; *Node 2* stores all keys starting with e, and so on. Range partitioning is simple to implement and ensures that similar keys are stored on the same node. When a node becomes overloaded, the data is *repartitioned*. The data is split into two parts, and another node becomes responsible for one of the parts.

(2) *Hash-partitioning:* A hash function is applied to the tuple keys. The value range of the function is mapped to the available nodes. The mapping determines which node stores which data. In most cases, the hash function scatters the data equally across all nodes; tuples with similar keys are stored on different nodes. *Consistent hashing* [28] makes it possible to add and remove nodes without repartitioning the already stored data.

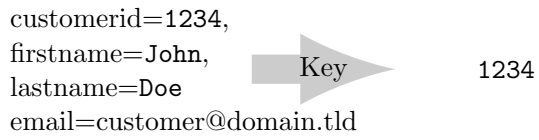
Finding a proper key for multi-dimensional data is hard and often impossible; this is especially true when the data has an extent (non-point data, also called *region* [23]). To retrieve multi-dimensional data from a key-value store, a full data scan is often required. This is a very expensive operation which should be avoided whenever possible.

Figure 1.2 depicts the key determination problem for one- and two-dimensional data. A customer record and the geographical information about a road should be stored in a KVS. For the customer record the attribute *customer id* can be used as the key[1]. Finding the key for a road (two-dimensional non-point data) is much more difficult. In traditional KVS, the data is stored under an unrelated key (e.g., the number of the road, like *road 66*) and a full data scan is required, when the road needs to be retrieved that intersects particular coordinates.

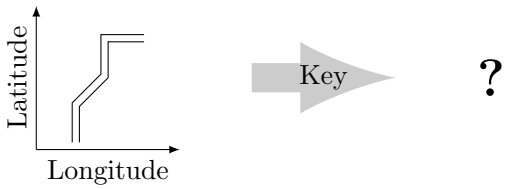
1.3 BBoxDB

BBoxDB uses a different technique to distribute the data. As a *key-bounding-box-value store* (KBVS), BBoxDB stores each value together with an n-dimensional axis parallel bounding box. The bounding box describes the location of the value in an n-dimensional space. For BBoxDB–and generally for KVS–the value simply is an array of bytes. The meaning of the bytes depends on the application that generates the data. Therefore, the value can not be interpreted by the data store. The data store does not know which data belong together and which do not. A tuple in BBoxDB is defined

[1] The customer record has four attributes, but we assume that the customer record is accessed only via the customer id and we can treat the complete record as one-dimensional data.



a) One-dimensional data (e.g., a customer record)



b) Two-dimensional data (e.g., a road)

Figure 1: Determining a key for one- and two-dimensional data.

as $t = (key, bounding\ box, value)$. The key is used to identify a tuple clearly in update or delete operations.

When saving data in BBoxDB, a value and a suitable bounding box must be passed to the `put()` method. This means more work for application developers, who have to build a function that calculates the bounding box. But it solves the problem that only a one-dimensional key and a meaningless array of bytes is passed as a value to the data store. The bounding box is a generic way to provide contextual information independently from the data. It maps the data to an n-dimensional space and geometric data structures can be used to work with the data. Partitioning and organizing elements in an n-dimensional space are a common problem for geometric data structures (see Section 3.2).

A *space partitioner* is responsible for partitioning the space and mapping parts of the space (*distribution regions*) to the available nodes. Overloaded distribution regions are split by the space partitioner and redistributed dynamically (see Section 3.2). Even if distribution regions are split, read and write access to them are still possible at any time (see Section 4.1). At the moment, BBoxDB ships with a K-D Tree space partitioner, a Quad-Tree space partitioner and a Grid space partitioner. Additional space partitioners can be integrated easily by implementing a Java interface.

BBoxDB is written in Java, licensed under the *Apache 2.0 license* [5] and can be downloaded from the website [9] of the project. A *client library* for the communication with BBoxDB is available at *Maven Central* [10]; a popular repository for Java libraries. So, the client library can be easily imported into other Java projects. The client library provides an implementation of the network protocol of BBoxDB and some helper methods that simplify the work with BBoxDB.

1.4 Contributions and Outline

The main contributions of this paper are as follows: (1) We present the novel concept of a key-bounding-box-value store. (2) We employ the geometrical data structures (like the K-D Tree or the Quad-Tree) to solve a new problem: Managing shards in a distributed datastore. (3) We introduce the novel concept of distribution groups. All tables of a distribution group are distributed in the same way (i.e., co-partitioned). (4) We show the handling of objects with an extent. (5) We demonstrate how data with different dimensions can be handled. (6) We provide a practical and freely available implementation.

The rest of the paper is organized as follows: Section 2 describes the basics of the technologies used in BBoxDB. Section 3 introduces the architecture of the system. Section 4 describes some advanced architecture topics. Section 5 gives a evaluation of the system. Section 6 describes the related work. Section 7 concludes the paper.

2 Building Blocks of the System

BBoxDB employs existing technologies to accomplish its work. The basics of these technologies are described in this section in brief.

2.1 Bounding Boxes

A bounding box is a geometric object; it is defined as the smallest rectangle that encloses an object completely. In this paper, we assume that bounding boxes are parallel to the Cartesian coordinate axis (so-called *axis-aligned minimum bounding boxes*). Bounding boxes are a generic data abstraction and they can be constructed for almost all types of data.

Data of an n-dimensional *relational* table can be described as points in an n-dimensional space [41, p. 1]. *Spatial* and *spatio-temporal* data can be described in two or three-dimensional space regarding their location and extent. A *customer record* can be described with a bounding box over some of the attributes (e.g., customer id, first name, last name, email). It is only important that the data has attributes and a function can be found that maps the attributes into $\mathbb{R}$ to construct a bounding box.

2.2 Apache Zookeeper

Building a distributed system is a complex task. Distributed tasks need to be coordinated correctly in a fault-tolerant manner. *Apache Zookeeper* [26] is a software that is designed to handle this work. The developers of Zookeeper describe it as a *distributed metadata filesystem*. Zookeeper was developed after Google published a paper about *Chubby* [13], which is used in systems like *GFS* [21] or *BigTable* [14]. Today, Zookeeper is used in a large number of systems to coordinate tasks.

Zookeeper provides a highly available tree (similar to a file system) that can be accessed and manipulated from client applications. The tree consists of nodes (similar to a directory in a file system) that have a name. A *path* describes the location of a node in the tree and consists of the names of all nodes, beginning from the root, separated by a slash. For example, the path `/node1/childnode2` points to a node called `childnode2`, which is a child of the node `node1`.

The tree consists of two different kinds of nodes: (1) *persistent nodes* and (2) *ephemeral nodes*. Persistent nodes are stored in the tree until they are deleted. Ephemeral nodes are deleted automatically as soon as the creating client disconnects. Zookeeper also supports *watchers*. By creating a watcher, a client gets notified as soon as the watched node changes.

2.3 SSTables

String Sorted Tables (SSTables) [14] are files that contain key-sorted key-value pairs. They are a kind of *Log-Structured Merge-Trees* (LSM-Trees) [37]. Google used SSTables in BigTable, as an alternative to existing data storage engines. The goals of SSTables are to provide a high write throughput and to reduce random disk IO. In contrast to other storage engines like *ISAM* (Index Sequential Access Method) or B-Trees, written data are not modified.

Modifications–such as changing or deleting data–are performed simply by storing a new version of the data. Deletions are simulated by storing a deletion marker for a particular key. A timestamp is used to keep track of the most recent version of the data.

New data are stored in a data structure called *Memtable*. As soon as a Memtable reaches a certain size, the table is sorted by key and written to disk as a SSTable. With time, the number of SSTables grows on disk. In a *major compactification*, all SSTables are merged into a new SSTable. SSTables are sorted by key to enable the compactification process to be carried out efficiently: The SSTables are read in parallel and the most recent version of a tuple is written in the output SSTable. The new SSTable contains only up-to-date tuples. No deletion markers are needed to invalidate older versions of the tuple. Therefore, all deletion markers can be removed. In addition to a major compactification, in which all data is rewritten, smaller *minor compactifications* also exist. In a minor compactification only two or more tables are merged and outdated data are removed.

Deletion markers are written to the new SSTable. Other SSTables may exist which contain older versions of a tuple that needs to be invalidated by a deletion marker.

All SSTables and Memtables are associated with a *Bloom filter* [12]. The filter is used to prevent unnecessary table reads. Bloom filters are a space-efficient probabilistic data structure. The Bloom filter contains information regarding whether or not an SSTable or a Memtable *might* contain a certain key.

3 Architecture of the System

In this section, the architecture of BBoxDB is described. BBoxDB is a distributed system; an installation of BBoxDB usually consists of multiple processes running on multiple nodes. A running BBoxDB process is from now on referred as *BBoxDB instance*. A BBoxDB instance is responsible for the data of one or more *distribution regions* (see Section 3.3).

To blend the distributed BBoxDB instances into a unified system, Zookeeper is used. Zookeeper accomplishes mainly two tasks: (1) It discovers all running BBoxDB nodes, keeps track of the state of the nodes, and provides a list with all contact points (i.e., IP-Addresses and ports). (2) It stores the *distribution directory*. The distribution directory contains a mapping between the distribution regions and the BBoxDB instances that are responsible for a region. Figure 2 depicts an installation with four BBoxDB instances on four hardware nodes (*node1*, *node2*, *node3*, *node4*). Each hardware node runs one BBoxDB instance. Each instance is responsible for multiple distribution regions.

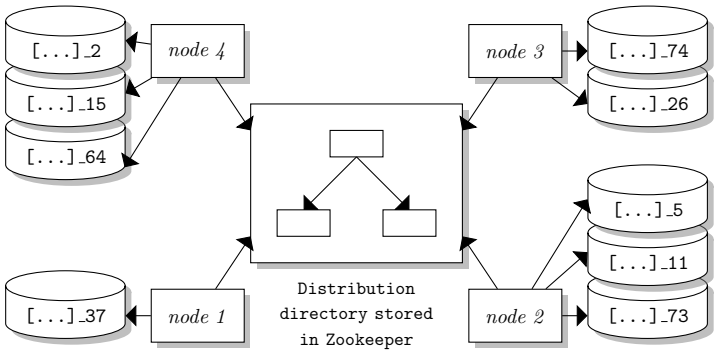


Figure 2: A cluster with four BBoxDB instances. Each instance stores the data of multiple distribution regions.

3.1 Supported Operations

The operations of BBoxDB can be categorized into two groups: (1) operations that define the structure of the data and (2) operations that work with the data. The first category includes operations that define distribution groups and tables while the second category includes operations that read and write tuples. Table 1 presents an overview of the supported operations.

The operations `createDGroup` and `deleteDGroup` are used to create and delete distribution groups. When a distribution group is created, additional parameters such as the dimensionality and the replication factor have to be specified. The operations `createTable` and `deleteTable` are used to create and delete tables. Tables are a collection of tuples. The operation `put` is used to store new data, while the operation `delete` removes data from a table. The operation `getByKey` receives the data for a given key. The operation `getByRect` receives all data, whose bounding box intersects a given hyperrectangle. The operation `getByTime` returns all the data that is newer than the given time. The operation `getByTimeAndRect` combines both of the previous operations–all data that intersect a given hyperrectangle and are newer than a given time stamp are returned. The operation `join` executes a spatial-join based on the bounding boxes of two tables. The tuples of both tables that have bounding boxes that intersect each other inside of a certain hyperrectangle are returned.

Data definition:		
`createDGroup`	string × config	→ boolean
`deleteDGroup`	string	→ boolean
`createTable`	string × config	→ table
`deleteTable`	string	→ boolean
Data manipulation:		
`put`	table × tuple	→ table
`delete`	table × string	→ table
`getByKey`	table × string	→ tuple(...)
`getByRect`	table × hyperrectangle	→ stream(tuple(...))
`getByTime`	table × int	→ stream(tuple(...))
`getByTimeAndRect`	table × hyperrectangle × int	→ stream(tuple(...))
`join`	table × table × hyperrectangle	→ stream(tuple(...))

Table 1: Supported operations by BBoxDB.

3.2 Two-Level Indexing

We have implemented a two-level index structure to efficiently locate tuples. A *global index structure* determines, which node is responsible for which area in space. A *local index structure* on each node is responsible for locating individual tuples. The local index has to be changed as soon as new tuples are written to disk.

The global index structure, called the *distribution directory*, is stored in *Apache Zookeeper*. All clients maintain a local copy of the global index structure in their memory. This enables them to query the data structure quickly. The basic idea of two-level indexing is that the global index structure contains subspaces (*distribution regions*) instead of tuples. Changing the distributed global index structure is an expensive operation. In our implementation, the global index structure has only to be changed when the mapping between the nodes and distribution regions is changed; storing tuples does not affect the global index structure. The local index structure employs an R-Tree [24].

The global index can be created with different data structures. At the moment BBoxDB supports K-D Trees [11], Quad-Trees [18] and a fixed grid to create the global index. Depending on the used data structure, BBoxDB provides different features. Some data structures support dynamic partitioning, others do not. Some support splitting the space at any desired point; others work with fixed split points.

The K-D Tree is the most flexible data structure, it supports the dynamic partitioning of the space and it can split the space at every desired position. The Quad-Tree also supports dynamic partitioning but it can split the space only at fixed positions. The fixed grid is the less flexible data structure. The space is split up into fixed cells and the data can not be partitioned dynamically. The fixed grid is primarily implemented in BBoxDB for comparison experiments with other space partitioners. Because the K-D Tree is the most flexible data structure for the global index, we will discuss only this data structure for the global index in this paper.

3.3 Distribution Groups, Distribution Regions, Table Names, and Region Tables

Distribution groups determine the data distribution of a collection of tables. Each table in BBoxDB belongs to exactly one distribution group. The table name is always prefixed with the name of the distribution group. For example, the table *mytable* in the distribution group *mygroup* is named: `mygroup_mytable`

The data of an n-dimensional space is split into disjoint spaces, called *distribution regions*. A *distribution region* is a subspace of an n-dimensional space. According to the replication factor, one or more BBoxDB instances are responsible for storing the data of a distribution region.

The tables in BBoxDB are split according to the distribution regions. These *region tables* are stored on various nodes. To identify the region table a suffix–called the region id of the table–is

attached to the table name. For example, the fifth region table *mytable* in the distribution group *mygroup* is named: `mygroup_mytable_5`

Figure 3 depicts an example: A two-dimensional distribution group is split up into three distribution regions (shards). The data of the contained table *customer* is spread according to the distribution group's structure. The two attributes *id* and *age* of the table are used to determine the location of the tuples in the two-dimensional space.

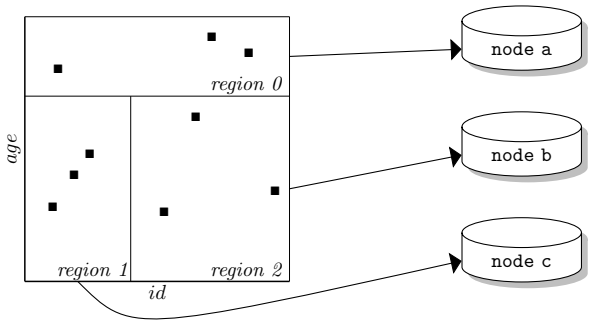


Figure 3: Spreading the table customer onto multiple nodes using two-dimensional shards.

3.4 Redistributing Data

Adding or removing data can cause an unbalanced data distribution. Two values ($t_{overflow}$ and $t_{underflow}$, with $t_{overflow} > t_{underflow}$) control the redistribution of the data. When the size of a distribution region is larger than $t_{overflow}$, the region is split. When the sum of the mergeable distribution regions is smaller than $t_{underflow}$, the region is merged. We call distribution regions *mergeable* when the space partitioner can merge these regions. With the K-D Tree space partitioner, all the leaf nodes of the same parent can be merged. The redistribution of the data is performed in the background. At all times, read and write access to the data is possible. Each distribution region also has a state that determines whether read or write operations are sent to this region or not. Table 2 lists all the possible states of a distribution region.

State	Description	Read	Write
`CREATING`	The distribution region is in creation.	no	no
`ACTIVE`	This is the normal state of the region.	yes	yes
`ACTIVE-FULL`	The region has reached the maximal size and gets split soon.	yes	yes
`SPLITTING`	The data is spread to the child nodes at the moment.	yes	no
`SPLIT`	The region has been split and the data is spread to the child nodes.	no	no

Table 2: The states of a distribution region.

Mainly the split of a distribution region consists of two steps: (1) finding a good split point. This means that after the split the resulting regions should have an almost equal amount of stored tuples and (2) redistributing the already stored data. BBoxDB reads samples from every table in the distribution group. The samples are used to find a split point that creates two distribution regions with equal numbers of stored tuples.

Keeping the distribution region accessible during the split requires some additional steps. The basic idea is, to create the new regions and store all newly written data immediately in the new regions. Queries are executed on the old and the new regions until the data is completely redistributed. On a replicated distribution group, several BBoxDB instances could notice at the same time that the region needs to be split. To prevent this, the state of the distribution region is changed from `ACTIVE` to `ACTIVE-FULL`. Zookeeper is used as a coordinator that allows exactly one state change. Only the instance who performs this transition successfully, executes the split.

When a region is split, the split position is published in Zookeeper and two new child-regions are created. The new regions are created with the state `CREATING` to indicate that the data of the region might be incomplete in Zookeeper. As soon as all data is available in Zookeeper (e.g., all systems are stored in Zookeeper that are responsible for the region), the state of the regions is set to `ACTIVE`. Beginning from this point in time, data is written to the old and to the new regions. Also, queries will be sent to all regions.

Now, the existing data is spread to the new regions and the state of the old region is set to `SPLITTING`. The old region does not accept new data from now on; but read access is still possible. Tuples with a bounding box that intersects both new regions are stored in both new regions; these tuples are duplicated. After all data is redistributed, the state of the old region is set to `SPLIT`. From now on, no queries will be executed on this distribution region and the data of the old region can now safely be deleted.

3.5 The Local Read and Write Path

Writing data is performed as already described in Section 2.3. To access data, BBoxDB supports two different methods: (1) query a tuple by key and (2) query all tuples that intersect a bounding box. Depending which access method is used, BBoxDB uses different strategies to retrieve the data.

Retrieving the tuple for a given key requires searching in all Memtables and SSTables for the requested key. Memtables–which are very small and only exist until they are flushed to disk– are completely scanned for the requested data, while SSTables are scanned using binary search. To decrease the number of scanned Memtables and SSTables, Bloom filters are used. Only the Memtables and SSTables that might contain the given key are read. The SSTables are accessed using *memory mapped files* [27], which speed up data access. Memory mapped files provide faster data access compared to standard file oriented IO, by reducing the number of needed system calls and avoiding unnecessary memory copy operations. After all the Memtables and SSTables are processed, the most recent version of the tuple is returned.

To find efficiently all tuples that intersect a particular bounding box, a spatial index is used. The current version of BBoxDB uses an R-Tree to index the bounding boxes of the stored data. In contrast to other KVS that support multi-dimensional data, the index-layer is directly implemented at file system level and not build as an additional layer over the KVS.

3.6 The Network Read and Write Path

BBoxDB is a client-server application. Clients connect to the BBoxDB instances and request or insert data. Therefore, it must be known which BBoxDB instances are alive, how they can be reached and what data is stored on which instance. As described earlier, this information is stored in Zookeeper. Each client connects to Zookeeper and reads this data. With the list of available BBoxDB instances and the distribution directory, clients know which BBoxDB instances they have to contact in order to perform an operation.

Compared to accessing data from a local hard disk or from memory, transferring data over a network is relatively slow. Handling operations (see Table 1) can also take some processing time. To deal with this, BBoxDB works with a binary, asynchronous, package oriented protocol. This means that every operation is represented by a network package, i.e., a sequence of bytes. Each package contains a *sequence number* to identify the package. BBoxDB handles the packages in an asynchronous way; i.e., a BBoxDB instance accepts new packages continuously, even when previous operations are not complete. A network connection can be used in a multiplexed manner; a client can sent multiple packages in one connection, without waiting for the completion of the previously transmitted packages.

As soon as a BBoxDB instance has read a package from the client, it performs the necessary tasks to complete the operation. After an operation is completed, the server sends a result back to the client. The result contains the original sequence number, in order to enable the client to

identify the original operation. Since the client application does not have to wait for the completion of operations, a single connection can handle many operations in parallel.

For example, when three tuples should be stored on a BBoxDB instance, three *put operation packages* (see Section 3.1) with different sequence numbers are sent to the BBoxDB instance. The BBoxDB instance acknowledges the processing of each package as soon as the data is stored.

3.7 Joins and Co-Partitioned Tables

In comparison to data accessed from a local hard disk, data transfer through a computer network is considerably slow, has high latency, and reduces the speed of data processing immensely. A common technique to reduce network traffic is to run the data storage and the query processing application on the same hardware. The aim is to access all the data from the local hard disk instead of accessing the data from another computer.

BBoxDB and its distribution groups are designed to exploit data locality. All tables in a distribution group are distributed in the same manner (co-partitioned). This means that the same distribution region of all tables is stored on one node. On co-partitioned data, a distributed equi- or spatial-join can be efficiently executed only on locally stored data. For a join $\bowtie_p$, we call two partitioned relations $R = \{R_1, ..., R_n\}$ and $S = \{S_1, ..., S_n\}$ *co-partitioned* iff $R \bowtie_p S = \bigcup_{i=1,...,n} R_i \bowtie_p S_i$. BBoxDB supports spatial-joins out-of-the-box using the bounding boxes of the stored data with the `join()` operation (see Section 3.1).

For example: the two tables *roads* and *forests* store the spatial data of all roads and of all forests in a certain area. Both tables are stored in the same distribution group. The bounding box of the tuples is calculated using the location of the tuples in the two-dimensional space. A spatial-join should determine which roads lead through a forest. Because all tables of a distribution group are stored co-partitioned, all roads and all woods that might share the same position in space are stored on the same BBoxDB instances (see Figure 4).

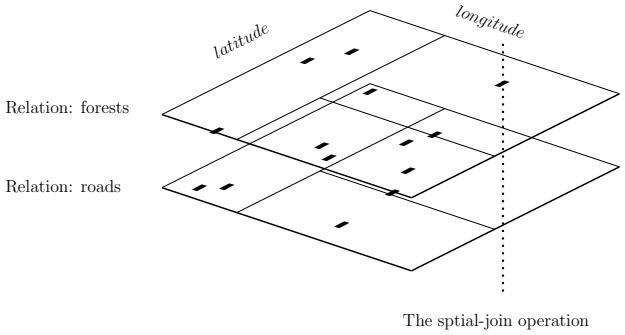


Figure 4: Executing a spatial-join on local stored two-dimensional data. Both relations are stored in the same distribution group. Therefore, the relations are partitioned and distributed in the same manner.

4 Implementation Details

In the previous section, the basic architecture of BBoxDB is discussed. This section describes some of the implemented techniques to improve the functionality.

4.1 Operation Routing

When a distribution region is split, the state of the region is changed multiple times (see Section 3.4). There is a delay between changing the state in Zookeeper and the moment when all Zookeeper clients have received the update. During that time, these nodes have an outdated version of the distribution directory. We have observed that this delay can cause problems.

For example, assume we have two BBoxDB-instances A and B. A is responsible for the distribution regions 1 and 2 and B is responsible for the distribution regions 3 and 4. A client wants to write data into the regions 2, 3 and 4. The client sends a network package with the put operation to instance A which processes the package. Now, the package is sent to instance B. We assume that the distribution region 4 is split before the package has arrived at instance B. The newly created region 5 is located on instance A and region 6 is located on instance B. Now the put operation is processed by the instance B and the data is inserted into the local storage and active instances 3 and 6. The put operation has also be executed on the newly created region 5 on instance A, which was missed. The operation was only executed on the regions 2, 3 and 6.

Our first solution for this problem was to check whether the distribution directory in Zookeeper has changed between the begin and the completion of the write operation. We have observed that Zookeeper updates have a long delay when a client is under load. Therefore, the instances A and B can already have received the changes, and the client is still operating on an outdated version.

Another problem is that data often has to be stored on multiple nodes. This occurs (1) when a distribution group is replicated on multiple nodes or (2) the bounding box of the data intersects multiple distribution regions. In both cases, the client has to send the data to multiple BBoxDB instances. The network bandwidth could become a bottleneck in this situation.

Most modern computer networks support *full-duplex* data transfer; data can be simultaneously sent and received with the full speed of the network link. To exploit the full-duplex network links BBoxDB implements a routing mechanism for put operations. The put network package contains a routing list, which determines which BBoxDB instances should receive the package. The package is forwarded between the BBoxDB instances regarding the routing list. The last node of the routing list sends a *success package* back to the instance from which it received the package. The instance in turn sends a success package to the node from which it received the package. That creates a chain of success packages which finally arrives at the client (see Figure 5). This technique is known as *pipelining* and is also used in GFS [21].

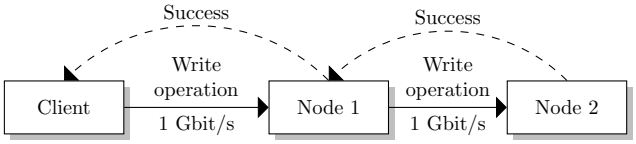


Figure 5: Routing of a put operation between two nodes. The client sends the operation to node 1. This node receives and sends data at the same time with 1 Gbit/s.

The package routing is also used to ensure that operations are sent to all desired distribution regions, which solves the problem described in the example. The client generates the routing list at the moment when the operation is triggered. The routing list contains the connection points (i.e., the IP-address and the port number) of all BBoxDB instances and the number of all distribution regions which the operation should reach. The list with all active connection points is fetched from Zookeeper. Each BBoxDB instance publishes its connection point there. Zookeeper keeps track that unreachable instances are automatically removed. When a BBoxDB-instance receives a package with a write operation, the contained data is inserted into the specified distribution regions, the routing hop is increased by one and the package is send to the next BBoxDB instance in the routing list.

Before a BBoxDB instance inserts data into the local stored distribution regions, it checks whether all local regions specified in the routing list are still writeable. When a distribution region is split, the region becomes read-only (see Table 2). If data should be written to an read-only region, the BBoxDB instance sends an error package back. This error package is forwarded the hole routing list back to the client. In this case, the client gets notified that the data was not written completely. The client waits a short time, computes an up-to-date routing list and sends the insert package again. This is repeated until the insert package has written the data to all desired distribution regions. When the clients received the success package back, the data has reached all desired distribution regions.

Almost the same technique is implemented for read operations (queries). In contrast to the write operations, the read operations are sent only to one node and the result of the operation is a list of tuples. The routing list is used to specify which distribution regions should be queried. If a distribution region cannot be queried (e.g., the split of a region is complete and the data is deleted) an error package is sent back to the client. The client waits a short time and executes the query again based on the newest data stored in Zookeeper.

4.2 The Java-Client

BBoxDB provides an implementation of the network protocol for the programming language Java (the *BBoxDB Java-Client*). The Java-Client is available at *Maven Central* [10]; a popular repository for Java libraries. All the supported operations of BBoxDB are available as a method of a Java class. In addition, the driver reads the state and the addresses of the BBoxDB servers from Zookeeper and creates a TCP-connection to each available instance. Failing and newly started instances are automatically noticed due to the changes in the directory structure of Zookeeper.

The driver also keeps track of the state of the TCP-connections; it reopens a connection that has closed unexpectedly. Also, *keep-alive packages* are sent periodically to keep the connection open even when no data is being transferred. Sending keep-alive packages is a common technique in other protocols also, for letting TCP-connections be established over a longer period. Components like firewalls often track the state of connections and terminate idle connections.

The network protocol works in an asynchronous manner. Therefore, the result of an operation becomes available after some time. We have implemented the *Future pattern* [8] in the BBoxDB-Client. Each operation that involves server communication returns a future, not a concrete value. The calling application code can wait until the concrete result is returned from the server or can process other tasks during that time. This creates a high degree of parallelism in the client code and helps to utilize the resources (e.g., the CPU cores and the network) as much as possible.

4.3 Merging and Compression of Operations

All operations between a BBoxDB instance and a client are encoded as packages. The BBoxDB driver creates a TCP-Socket to each available BBoxDB instance. Each connection is associated with a queue. The packages for each instance are stored in that queue. After 200 ms, the content of each queue is put into a big *envelope package* and compressed. Afterward, the envelope package is transferred over the network (see Figure 6). The receiving BBoxDB instance decompresses the envelope package and processes all contained packages.

Compressing multiple operations into one package improves the compression rate and decreases the network traffic for most kinds of data. Also, writing data to a network socket and flushing the socket requires at least one system call. System calls are time-consuming operations. Therefore, merging multiple operations into one bigger network package reduces the amount of expensive writing and flushing operations. An experimental evaluation of this technique is discussed in Section 5.2.

The queuing of the operations adds some latency to the processing time of the package. As described in Section 4.2, all network operations are executed asynchronously. Normally, the queues are flushed every 200 ms. For inserting new data into BBoxDB, the latency does not matter. In contrast, a query should be executed as fast as possible. The queuing increases the time until the query is executed. Therefore, the queue is immediately flushed as soon as a query package is inserted.

In this section, the technique is described from a client's point of view. The same technique is implemented on the server side as well.

4.4 The GUI

BBoxDB ships with a GUI that visualizes the distribution directory and shows all discovered BBoxDB instances. The installed version of BBoxDB and the state of each instance are also shown. Figure 7(a) shows the distribution directory and the state of the distribution regions. In the left

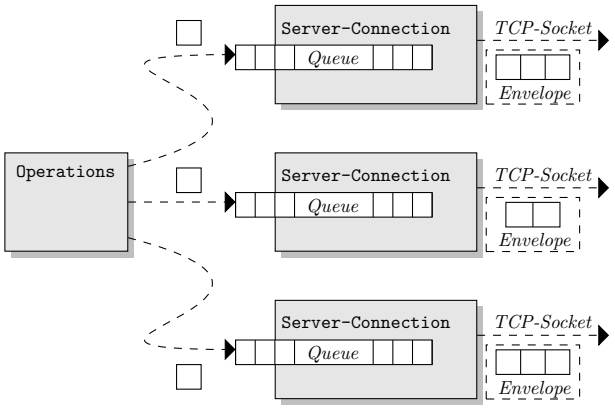
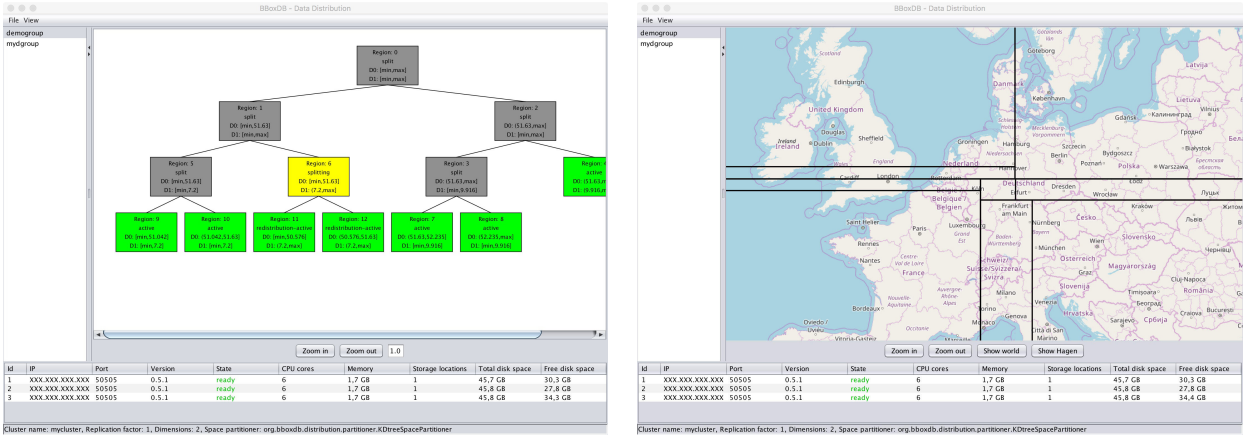


Figure 6: The implemented network package compressing mechanism.

area of the screen, all distribution groups are shown. The global index of the selected distribution group is visualized in the middle of the screen. At the bottom of the screen, all discovered BBoxDB instances are shown. For two-dimensional distribution groups, which use *WGS84 coordinates*, an open street map overlay is also available. The distribution groups are painted on a world map as is shown in Figure 7(b).



(a) The GUI of BBoxDB shows the distribution directory. (b) The distribution tree as an OpenStreetMap overlay.

Figure 7: The GUI of BBoxDB.

5 Evaluation

The evaluation of BBoxDB is performed on a cluster of 10 nodes. Five of these nodes (*node type 1*) contain a Phenom II X6 1055T processor with six cores, eight GB of memory and two 500 GB hard disks. Five of these nodes (*node type 2*) contain an Intel Xeon E5-2630 CPU with eight cores, 32 GB of memory and four 1-TB hard disks. All nodes are connected via a 1 Gbit/s switched Ethernet network and running Oracle Java 8 on a 64 bit Ubuntu Linux.

For the evaluation, two synthetic and two real datasets with one to four dimensions are used. The characteristics of the datasets are shown in Table 3.

TPC-H dataset (one-dimensional): This dataset is generated by the data generator of the TPC-H benchmark [44]. The benchmark data is generated with a scale factor of 20. The table *order* contains 30 000 000 entries and has a size of 3 GB. To create a point bounding box in the one-dimensional space, the order date of the table is used. The table *lineitem* has a size of 15 GB and contains 119 994 608 line items. For the range data, the range of the ship date and the receipt date is used.

Dataset	Data	Type	Elements	Size
TPC-H	Order	Point	30 000 000	3 GB
TPC-H	Line item	Range	119 994 608	15 GB
OSM	Tree	Point	5 317 989	900 MB
OSM	Road	Range	52 628 226	22 GB
NYC Taxi	Trip	Point	69 406 520	11 GB
NYC Taxi	Trip	Range	69 406 520	11 GB
Synthetic	Synthetic	Point	2 000 000	20 GB
Synthetic	Synthetic	Range	2 000 000	20 GB

Table 3: Summary of the used datasets.

OSM dataset (two-dimensional): Data from the *OpenStreetMap Project* [38] are used for this dataset; the spatial data of Europe are used. The dataset has a size of around 20 GB in the compact binary encoded *protocol buffer binary format*. Excerpts of the dataset, such as trees (point data) or all roads (range data), are used in the experiments. The dataset includes 5 317 989 trees (900 MB) and 52 628 226 roads (22 GB).

NYC Taxi dataset (three-dimensional): The *New York City Taxi and Limousine Commission* (NYC TLC) collects data about the beginning and end of the trips of the taxis in New York City [43]. The data of the *yellow taxis* between 01/2016 and 06/2016 is used in this dataset. The dataset has a size of 11 GB and includes 69 406 520 trips. Again, the bounding box for these trips is generated in two different ways: (1) the longitude and latitude of the pick-up point and the time of boarding are used to create a point in a three-dimensional space. (2) The longitude and latitude of the destination and the time of arrival are additionally used to create a rectangle in the three-dimensional space.

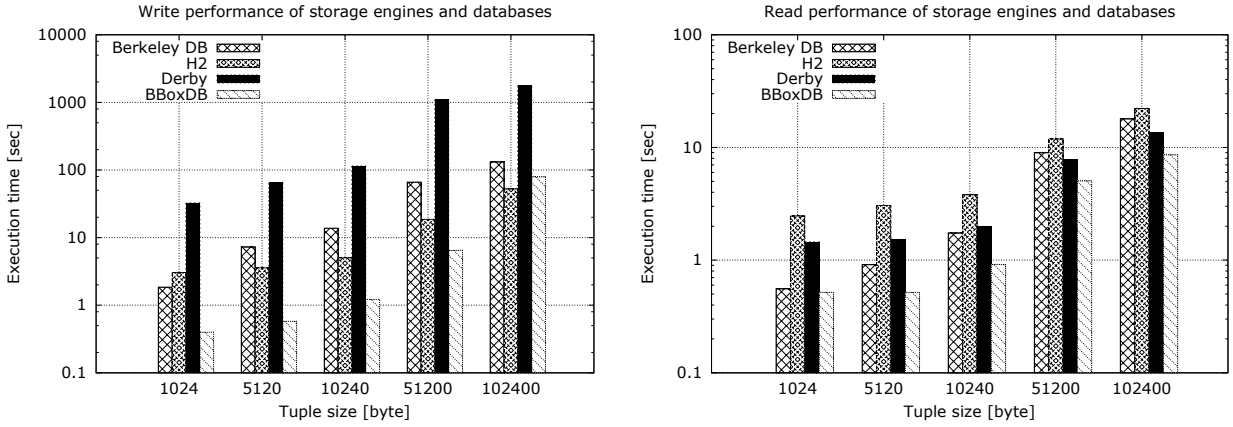
Synthetic dataset (four-dimensional): BBoxDB contains a data generator that generates synthetic random data in any desired dimension and with any length. The data generator generates point or hyper-rectangle bounding boxes in the specified dimension. For this dataset, 2 000 000 elements with a data length of 10 000 bytes are generated; this results in a dataset of 20 GB. The dataset is generated twice: (1) once with four-dimensional point bounding boxes and (2) once with four-dimensional hyperrectangle bounding boxes. Each dimension of the bounding box can cover up to 10% of the space.

5.1 Performance of the Storage Layer

BBoxDB uses SSTables to store data. In this section, the performance of the storage layer is compared with other popular systems, that run inside of the Java virtual machine and don't require network communication. Such as (1) *Oracle Berkeley DB Java edition* [39] (version 7.3.7), the embedded Java SQL-database (2) *H2* (version 1.4.195) [25], and (3) the embedded Java SQL-database *Apache Derby* (version 10.13.1.1) [2].

For this experiment, 100 000 tuples are written and read with sizes of 1-100 KB. If supported, transactions are disabled to speed up the query performance. H2 and Derby are accessed using a JDBC driver through prepared SQL statements. The H2 database is configured using the *nio* file access method. The SSTables are accessed directly without writing the tuples on a network socket or using any logic for data distribution. The experiment is performed on one hardware node of type 2. Figure 8(a) shows a performance comparison for write operations, while Figure 8(b) shows a performance comparison for read operations.

It can be seen that our SSTables implementation provides a very fast storage engine. In most cases, the storage engine is faster than the other evaluated software. But it is important to keep in mind, that our SSTables implementation is a very simple data structure. Features such as transactions or locking, which are supported by all other competitors, are not supported. H2 and Derby have to process SQL queries which generates some overhead compared to direct data access. In addition, these databases support a rich data model and features such as joins or secondary indices. These features are not provided by the SSTables implementation, which makes the application pos-



(a) Comparison of the write performance.

(b) Comparison of the read performance.

Figure 8: Comparison of Java based databases / storage engines and the storage engine of BBoxDB.

sibilities limited. Other databases that employ SSTables as storage engine (like *LevelDB* [30]) show a similar read and write performance compared to traditional SQL based databases [31].

5.2 Efficiency of the Network Package Compression

In this experiment, the efficiency of the compression of network packages is examined. The result of the experiment is given in Figure 9. The compression rates of the point and the range versions of the datasets are almost identical. Therefore, only one version of each dataset is plotted in the figure.

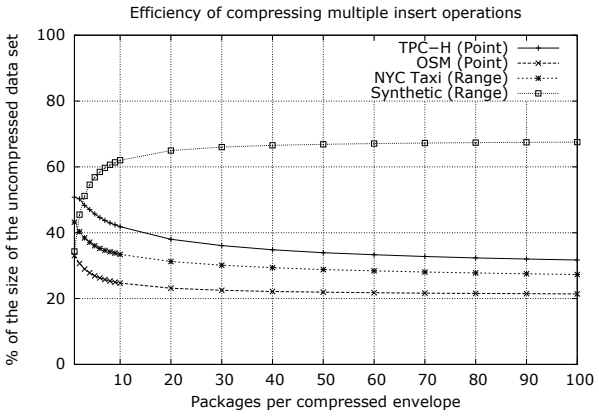


Figure 9: Efficiency of compressing multiple insert operations.

It can be seen that all datasets expect the synthetic dataset can be compressed with a higher ratio when more packages are compressed at once. The datasets consist of text that contains repeated strings and can be very well compressed. The synthetic dataset consists of random data, which contains almost no repeated strings, making it hard to compress. Only the repeated parts can be compressed, like the header of the network package. But even for this type of data the compression is advantageous. By compressing multiple packages, the dataset can be compressed by around 30%.

Compared to access data from memory or a local hard disk, transferring data through a computer network is a slow task. Therefore, it is important for a distributed system to keep the amount of data transfers small. In addition, most public cloud providers such as *Amazon Web Services* (*AWS*) [6] or *Microsoft Azure* [32] only provide network connections with a fixed bandwidth.

Even the number of nodes can be scaled-up to an almost unlimited extent, the network bandwidth of a single node is fixed. Moreover, public cloud providers only specify the bandwidth for a single

region[2], the available bandwidth between multiple regions is not guaranteed and the traffic between regions usually has to be paid [7].

5.3 The Sampling Size

The sample-based splitting algorithm from Section 3.4 reads some tuple samples to calculate a good split point. The experiment of this section examines the amount of data that has to be read to find a good split point.

First, a certain number of random samples (between 0.01% and 10%) are read from the datasets and the split point is calculated. Then, the whole dataset is split at this point. Afterward, the size of both distribution regions is compared. The experiment is repeated 50 times for every sample size, Figure 10 depicts the average results of this experiment.

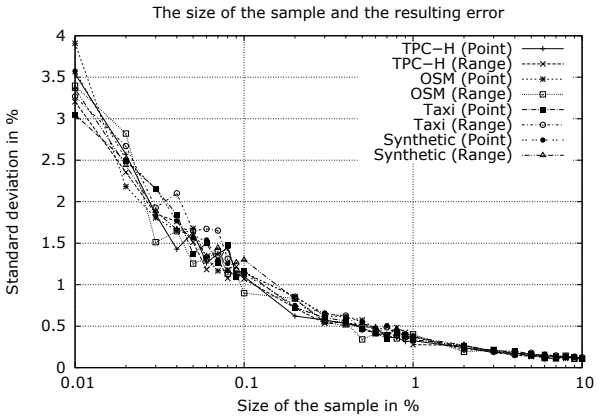


Figure 10: The sampling size and the resulting difference on a region split.

It can be seen, that a small sampling size is sufficient to find a good split point. Reading 0.1% finds a split point with a standard deviation of 1%. Reading 1% of the dataset lowers the standard deviation to 0.5%. As also can be seen in the figure, the standard deviation is almost independent of the used dataset. In BBoxDB, 0.1% of the data of a distribution region is used as a sample, for finding a good split point.

5.4 Distribution Region Size

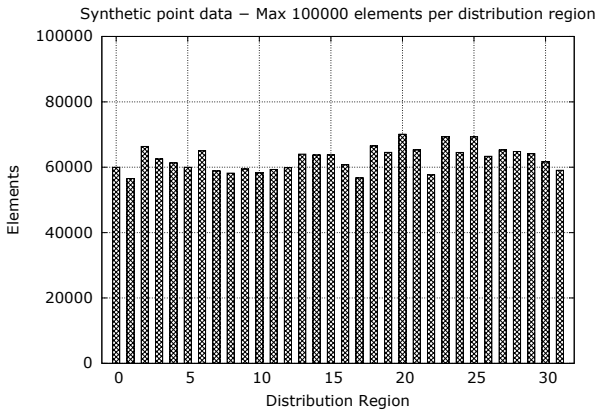
This experiment examines how the K-D Tree partitions the data of the datasets. The datasets are read and the data are inserted into the K-D Tree. The tree is split as soon as a region contains more than a fixed size of elements. The experiment is executed in two different variations: (1) all elements are of the same object class (e.g., the trips of a taxi) and (2) the elements consist of different object classes (e.g., roads and trees).

5.4.1 Similar Object Classes

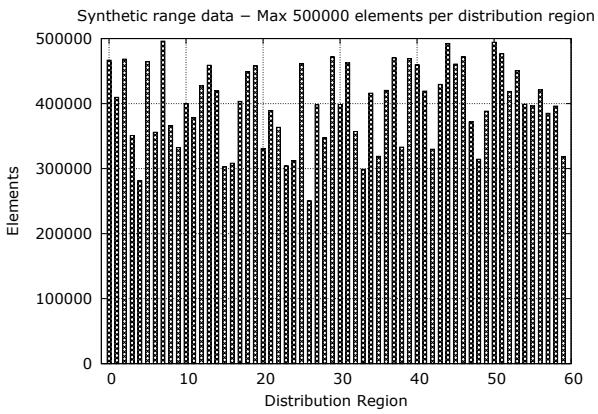
The results of this experiment are shown in Table 4, the elements of the datasets can be found in Table 3. The Figures 11 and 12 show the size of the distribution regions for some selected datasets and distribution region sizes. It can be seen that BBoxDB can split all datasets into distribution regions with a desired maximum size. The number of point elements does not change during the split task. Range data has to be duplicated, if an object belongs to more than one region. This leads to more elements after the split.

We compare BBoxDB with other approaches that use a static grid to handle spatial data, like *SJMR* [45] or Distributed Secondo [34]. To perform a spatial-join, they use a modified version of

[2]A region or an availability zone is a part of the cloud that is isolated from other parts against failures such as power or network outages.

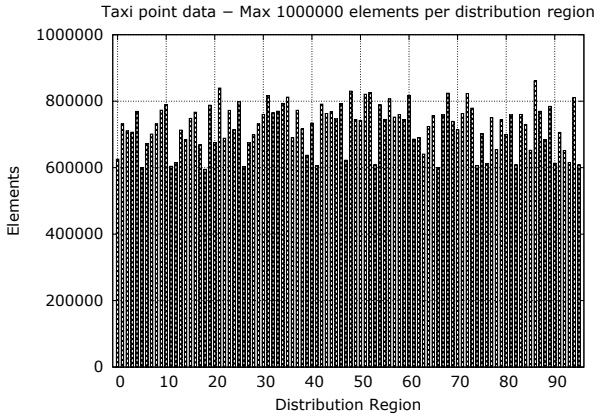


(a) Four-dimensional point data.

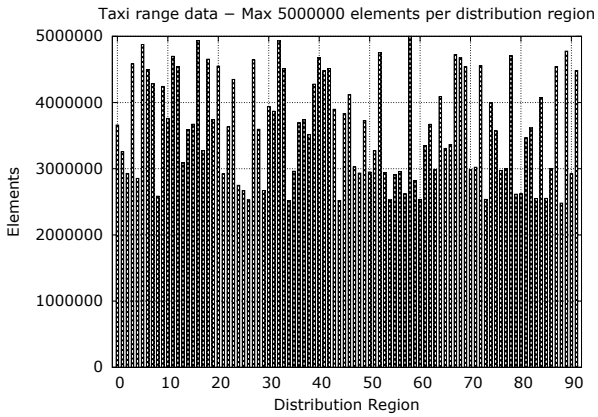


(b) Four-dimensional range data.

Figure 11: The size of the four-dimensional distribution regions in BBoxDB.



(a) Three-dimensional point data.



(b) Three-dimensional range data.

Figure 12: The size of the three-dimensional distribution regions in BBoxDB.

the *partition based spatial-merge join* [40] algorithm. An n-dimensional grid is created to partition the space into tiles. The input data is decomposed on the basis of the tiles. To handle the partition skew, the number of tiles is much higher than the available number of query processing nodes. To achieve this, in Distributed Secondo a hash function is applied to the tile numbers and the content of multiple tiles is merged into fewer *units of work* (UOWs). These UOWs are processed in parallel, independently by multiple nodes. Distributed Secondo uses 64 units of work per available query processing node. Table 5 show the size of the UOWs for the datasets. Two different grid sizes are used for every dataset: (1) *big tiles* to minimize the needed duplicates for range data and (2) *small tiles* to handle dense areas. To calculate the standard deviation (σ), the tiles mapped are mapped to 384 UOWs, which is used by a Distributed Secondo installation with 6 query processing nodes and 64 OUWs per query processing node.

It can be seen in the Figures 13 and 14 that very populated and almost empty UOWs exist. Since the grid is a static structure, more dense regions (e.g., a popular start place like an airport in the taxi dataset) cannot be split dynamically. It also can be seen, that objects that are part of multiple tiles needs to be duplicated. When the cells become smaller, one object belongs to more tiles and the number of stored elements increases. For example, the number of stored elements for the tpc-h (range) and synthetic (range) dataset increases sharply, when more tiles are used.

In summary, the dynamic approach, used by BBoxDB, generates more balanced partitions than the static cell-based approach used by systems like Distributed Secondo. In addition, less data duplication is needed.

Dataset	Max size	Needed regions	Total elements	σ
TPC-H (Point)	5 000 000	32	119 994 608	52 829
TPC-H (Point)	1 000 000	134	119 994 608	124 927
TPC-H (Range)	20 000 000	11	174 040 906	2 346 217
TPC-H (Range)	10 000 000	69	622 928 594	1 015 005
OSM (Point)	100 000	77	5 317 989	14 448
OSM (Point)	50 000	159	5 317 989	7 314
OSM (Range)	5 000 000	15	52 656 507	589 472
OSM (Range)	1 000 000	74	52 708 915	162 186
Taxi (Point)	5 000 000	17	69 406 520	711 407
Taxi (Point)	1 000 000	96	69 406 520	69 669
Taxi (Range)	10 000 000	31	211 963 948	1 221 402
Taxi (Range)	5 000 000	92	332 674 737	783 975
Synthetic (Point)	200 000	16	2 000 000	6 551
Synthetic (Point)	100 000	32	2 000 000	3 694
Synthetic (Range)	500 000	60	23 862 525	62 477
Synthetic (Range)	200 000	1 739	315 526 514	17 385

Table 4: The size of the distribution regions in BBoxDB.

5.4.2 Different Object Classes

The TPC-H and the OSM dataset consist of different object classes. In the TPC-H dataset we have the orders and the line items; in the OSM dataset we have roads and trees. Both object classes are also loaded into BBoxDB in this experiment and the distribution of the objects are examined. The result can be seen in Table 6. The experiment confirms that BBoxDB can also create multi-dimensional shards for different object classes. The data that are partitioned in that way can be used to execute joins efficiently only on locally stored data. For example, the table lineitem from the TPC-H dataset is partitioned using the *receipt date* and *ship date*; the data from the table order is partitioned using the *order date*. An equi-join can be used to find all orders that are shipped on the order date. The data from the OSM dataset are partitioned using the longitude and latitude. A spatial-join on this data can be used to find all trees that belong to the roads in the dataset[3].

5.5 Insert Performance

This experiment examines, how an increasing amount of BBoxDB instances can be used to improve the insert performance. The datasets are written to a BBoxDB cluster tuple per tuple. Between the experiments, the amount of available BBoxDB instances is increased from one to ten instances. The first five BBoxDB instances are executed on the hardware nodes of type 2; the last five instances are executed on hardware nodes of type 1. During the insert, up to 1000 futures can be unfinished while inserting multiple tuples in parallel (see Section 4.2).

The result of the experiment is depicted in Figure 15. It can be seen, that all datasets can be imported faster with an increasing amount of BBoxDB instances. The range datasets need more time to get imported than the point datasets. In the range datasets, several tuples need to be stored on multiple nodes; this increases the number of needed insert operations and increases the required time to import the whole dataset. It can also be seen, that the speed-up factor is not linear in this experiment. This is one of the topics we would like to improve in the next BBoxDB versions (see Section 7).

[3]Trees usually do not grow on roads; the tree and the road might not cover the same position in space. Therefore, the bounding box of one of the objects needs to be extended, so that the bounding box of the road and the tree do overlap.

Dataset	Grid size	Total elements	σ
TPC-H (Point)	1 000	30 000 000	100 384
TPC-H (Point)	10 000	30 000 000	103 562
TPC-H (Range)	1 000	1 395 344 574	4 313 743
TPC-H (Range)	10 000	12 864 972 636	11 173 030
OSM (Point)	500x500	5 317 989	15 400
OSM (Point)	1000x1000	5 317 989	10 158
OSM (Range)	500x500	54 393 009	30 138
OSM (Range)	1000x1000	56 315 204	17 311
Taxi (Point)	100x100x100	69 406 520	366 311
Taxi (Point)	250x250x250	69 406 520	231 624
Taxi (Range)	100x100x100	140 181 943	3 519 900
Taxi (Range)	250x250x250	426 437 491	751 350
Synthetic (Point)	10x10x10x10	2 000 000	997
Synthetic (Point)	15x15x15x15	2 000 000	399
Synthetic (Range)	10x10x10x10	683 750 980	422 485
Synthetic (Range)	15x15x15x15	2 553 677 770	858 766

Table 5: The size of the cells in a fixed grid.

Dataset	Max size	Needed regions	Total elements	σ
TPC-H	20 000 000	15	235 654 217	1 317 229
TPC-H	10 000 000	84	776 202 632	672 428
OSM	5 000 000	15	57 973 871	672 997
OSM	1 000 000	88	58 035 964	140 051

Table 6: The size of the distribution regions with different object classes.

6 Related Work

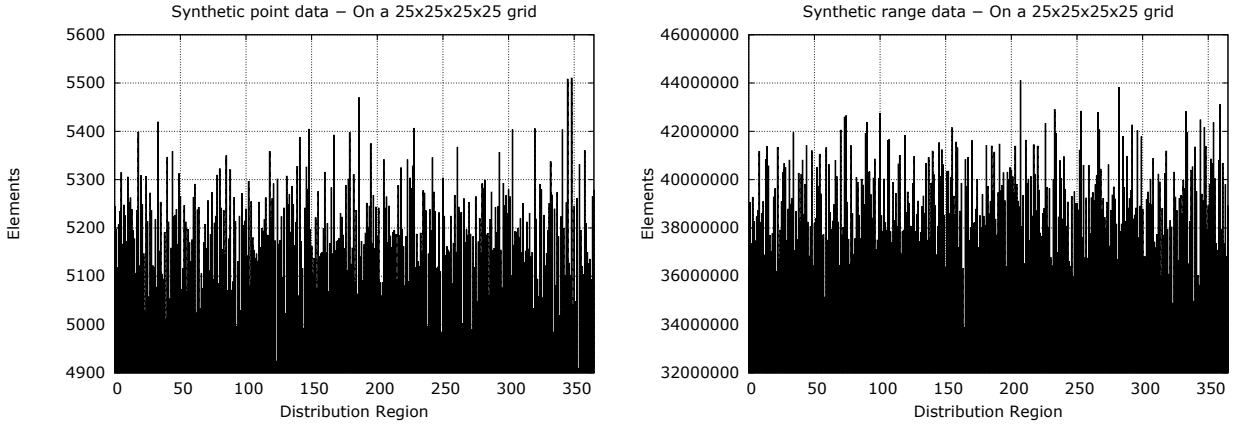
This section presents similar works; these works are explained below in greater detail. Table 7 compares the features of the related work with BBoxDB.

System	Supports multi-dimensional data	Supports region data	Supports updates	Supports dynamic redistribution	Supports the same distribution for multiple tables
GFS/HDFS	no	no	append only	yes	no
HBASE	no	no	yes	yes	no
Cassandra	no	no	yes	no	yes
ElasticSearch	2d	yes	yes	limited	no
GeoCouch	2d	no	yes	limited	no
MD-HBase	any	no	yes	yes	no
Distributed SECONDO	any	yes	yes	no	yes
Spatial Hadoop	2d	yes	no	no	yes
BBoxDB	any	yes	yes	yes	yes

Table 7: Features of the systems described in the related work section.

Distributed Systems for BigData. The *Google File System* (GFS) [21] and its open source implementation *Hadoop File System* (HDFS) are distributed file systems. HDFS is used by a wide range of applications like *Hadoop* [3] or HBase. Data is split up into *chunks* and stored on *chunk servers*. Chunks can be replicated onto multiple chunk servers. HFS/HDFS are a append only filesystem, written data can not be changed without rewriting the whole file. Both systems distribute data based on the generated chunks. BBoxDB instead uses the location of the data in a n-dimensional space for the placement. This allows one to create locality for adjacent data.

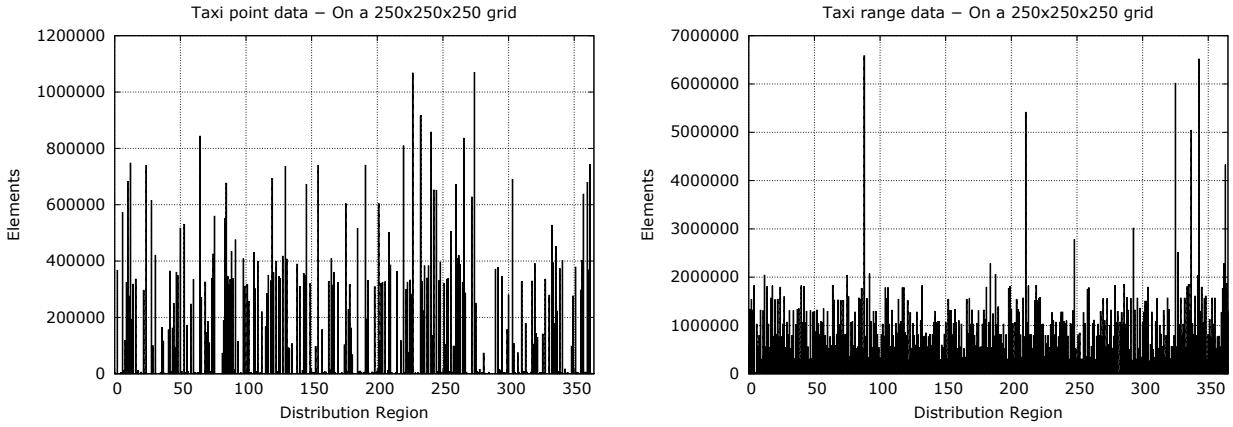
Apache Cassandra [29] is a scalable and fault tolerant NoSQL-Database. A logical ring represents the value range of a hash function. Ranges of the ring are assigned to different nodes. It is assumed



(a) Four-dimensional point data.

(b) Four-dimensional range data.

Figure 13: Using a static grid to partition the four-dimensional datasets.



(a) Three-dimensional point data.

(b) Three-dimensional range data.

Figure 14: Using a static grid to partition the three-dimensional datasets.

that this hash function can distribute the data uniformly across the logical ring. If that is not possible, the logical ring becomes unbalanced. Cassandra does not rebalance data automatically; unbalanced data has to be rebalanced manually. BBoxDB redistributes the data dynamically.

Apache HBase [4] is a KVS built on top of HDFS; it is the open source counterpart of BigTable [14]. HBase uses range partitioning to assign key ranges (regions) to nodes. The information regarding which server is alive and which server is responsible for which region is stored in Zookeeper. When a region becomes too large, it is split into two parts. The split of the region is created based on the key of the stored data. BBoxDB instead uses the location of the data for the distribution which results in inherently multi-dimensional join-partitioned regions. Additionally, BBoxDB supports distribution groups which ensures, that a set of tables is distributed in the same manner. In HBase the tables are distributed individually.

NoSQL Databases with Support for Spatial Data. *MongoDB* [33] is a document-oriented NoSQL database. Documents are represented in JSON. The database also supports Geodata, which is encoded as GeoJSON. Queries can be used to locate documents, that are covered or intersected by a given spatial object. In contrast to BBoxDB, the geodata cannot be used as a sharding key. Also, MongoDB supports only two-dimensional geodata.

ElasticSearch [15] is a distributed search index for structured data. The software supports the handling of geometrical data. Types such as points, lines or polygons can be stored. The data distribution is based on Quad-Trees or *GeoHashing* [20]. Querying data that contains or intersects a given region is supported. In contrast to BBoxDB, only two-dimensional data is supported.

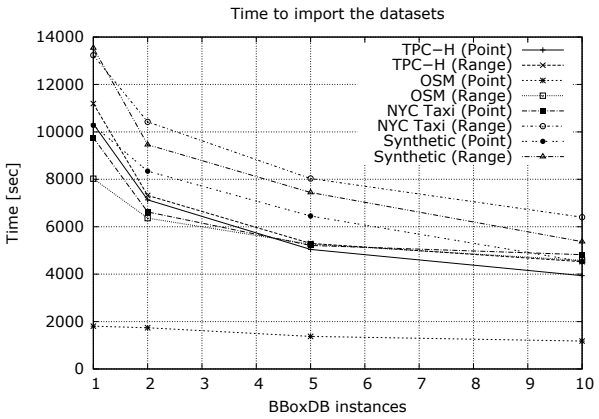


Figure 15: Importing the datasets on a varying amount of BBoxDB instances.

GeoCouch [19] is a spatial extension for the NoSQL database *Apache CouchDB* [1]. It allows executing bounding box queries on spatial data. CouchDB does not support the re-sharding of existing data. When a database is created, the user has to specify how many shards should be created. When the database grows, these shards can be stored on different servers. The database can not be scaled out to a higher number of nodes than the number of shards, specified at the creation of the database. BBoxDB, in contrast, allows to re-shard the database on any number of nodes.

MD-HBase [35] is an index layer for n-dimensional data for HBase. The software adds an index layer over the KVS HBase. MD-HBase employs *Quad-Trees* and K-D Trees together with a Z-Curve to build the index. BBoxDB, in contrast, directly implements the support for n-dimensional data. Also, BBoxDB can handle data with an extent and introduces the concept of distribution regions, which is essential for efficient spatial-join processing. MD-HBase is the most similar related work, but the source code is not publicly available. The authors have published *Tiny MD-HBase* [42]–a basic version that illustrates the key concepts of the MD-HBase paper. But this version is very simple and can only handle small datasets. Therefore, we could not present a performance evaluation of this software in this paper.

Distributed SECONDO [34] employs the single computer DBMS SECONDO [22] as a query processing engine and Apache Cassandra as a distributed and highly available data storage. Distributed SECONDO uses a static grid-based approach to partition spatial data. The read repair approach of Cassandra reads all replicates of a tuple on read operations, which causes network traffic even when only locally stored data is read. BBoxDB, in contrast, supports dynamical data partitioning, which leads to an equal data distribution across the nodes. Additionally, data locality can be exploited; reading local data does not create network traffic.

Spatial Hadoop [17] enhances Hadoop with support for spatial data. Operations like spatial indexing or spatial joins are supported. The software focuses on the processing of spatial data. *Pigeon* [16] is a set of user-defined functions, which allows to use the features of Spatial Hadoop in *Pig Latin* [36] scripts. As in most Hadoop/HDFS based solutions, modifying stored data is not supported. BBoxDB allows the modification of stored data. In addition, BBoxDB supports any dimensional data.

7 Conclusions and Future Work

In this paper, we presented the version 0.5.0 of BBoxDB, a key-bounding-box-value store. The system is capable of handling multi-dimensional big data; distribution regions are created dynamically based on the stored data, and are spread on different servers. BBoxDB supports data replication and does not provide a single point of failure. The software is licensed under the Apache 2.0 license and available for download from the website of the project [9].

In the upcoming versions, we plan to enhance the compactification tasks and store a certain number of old versions of a tuple. This data could be used to make the history of a tuple available. For example, if the position of a car is stored, it could be useful to get all versions (the positions) of this tuple of the last hour. The experiments have shown that BBoxDB is capable of splitting various datasets into almost equal-sized distribution regions and spread them across various nodes. In addition, the experiments have shown that the performance of the BBoxDB storage layer is fast. We plan to implement performance counters (e.g., amount of unflushed Memtables or received network operations per second) to get more insights about the behavior of the system. BBoxDB was examined on a cluster with 10 nodes; we plan to execute more experiments in a cluster with more nodes.

References

[1] Website of the Apache CouchDB project, 2017. `http://couchdb.apache.org/` - [Online; accessed 03-Jun-2017].

[2] Website of the derby database, 2017. `https://db.apache.org/derby/` - [Online; accessed 03-Jun-2017].

[3] Website of Apache Hadoop project. `http://hadoop.apache.org/`, 2017. [Online; accessed 15-Oct-2017].

[4] Website of Apache HBase. `https://hbase.apache.org/`, 2017. [Online; accessed 03-Jul-2017].

[5] Apache software license, version 2.0, 2004. `http://www.apache.org/licenses/` - [Online; accessed 15-May-2017].

[6] Amazon Web Services. `https://aws.amazon.com`, 2017. [Online; accessed 15-Jun-2017].

[7] Amazon Web Services - Regions and Availability Zones, 2017. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html` - [Online; accessed 15-May-2017].

[8] H.C. Baker and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[9] Website of the BBoxDB project. `http://bboxdb.org`, 2017. [Online; accessed 03-Jul-2017].

[10] BBoxDB at the maven repository, 2018. `https://maven-repository.com/artifact/org.bboxdb` - [Online; accessed 03-Jan-2018].

[11] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[12] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[13] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[14] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[15] Website of Elasticsearch. `https://www.elastic.co/products/elasticsearch/`, 2017. [Online; accessed 03-Jul-2017].

[16] A. Eldawy and M.F. Mokbel. Pigeon: A spatial mapreduce language. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 1242–1245, 2014.

[17] A. Eldawy and M.F. Mokbel. SpatialHadoop: A MapReduce Framework for Spatial Data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1352–1363, 2015.

[18] R.A. Finkel and J.L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Inf.*, 4(1):1–9, March 1974.

[19] Website of GeoCouch. `https://github.com/couchbase/geocouch`, 2017. [Online; accessed 03-Jul-2017].

[20] The Wikipedia article about Geohashing. `https://en.wikipedia.org/wiki/Geohash`, 2017. [Online; accessed 03-Jul-2017].

[21] S. Ghemawat, H. Gobioff, and S.T. Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[22] R.H. Güting, T. Behr, and C. Düntgen. Secondo: A platform for moving objects database research and for publishing and integrating research implementations. *IEEE Data Eng. Bull.*, 33(2):56–63, 2010.

[23] R.H. Güting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann, 2005.

[24] A. Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[25] Website of the H2 database, 2017. `http://www.h2database.com/html/main.html` - [Online; accessed 03-Jun-2017].

[26] P. Hunt, M. Konar, F.P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–25, Berkeley, CA, USA, 2010. USENIX Association.

[27] The class MappedByteBuffer of the Java API, 2017. `https://docs.oracle.com/javase/7/docs/api/java/nio/MappedByteBuffer.html` - [Online; accessed 03-Jun-2017].

[28] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY. USA, 1997. ACM.

[29] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[30] Website of the LevelDB project, 2017. `http://leveldb.org/` - [Online; accessed 03-Jun-2017].

[31] Benchmarks of the LevelDB project, 2017. `http://www.lmdb.tech/bench/microbench/benchmark.html` - [Online; accessed 03-Jun-2017].

[32] Microsoft Azure. `https://azure.microsoft.com/`, 2017. [Online; accessed 15-Jun-2017].

[33] Website of MongoDB project. `https://www.mongodb.com/`, 2017. [Online; accessed 03-Jul-2017].

[34] J.K. Nidzwetzki and R.H. Güting. Distributed SECONDO: A highly available and scalable system for spatial data processing. In *Advances in Spatial and Temporal Databases - 14th International Symposium, SSTD 2015, Hong Kong, China, August 26-28, 2015. Proceedings*, pages 491–496, 2015.

[35] S. Nishimura, S. Das, D. Agrawal, and A.E. Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *Proceedings of the 2011 IEEE 12th International Conference on Mobile Data Management - Volume 01*, MDM '11, pages 7–16, Washington, DC, USA, 2011. IEEE Computer Society.

[36] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[37] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[38] Website of the Open Street Map Project, 2017. `http://www.openstreetmap.org` - [Online; accessed 15-May-2017].

[39] Oracle Berkeley DB Java Edition, 2017. `http://www.oracle.com/technetwork/database/berkeleydb/overview/index-093405.html` - [Online; accessed 03-Jun-2017].

[40] J.M. Patel and D.J. DeWitt. Partition based spatial-merge join. *SIGMOD Rec.*, 25(2):259–270, June 1996.

[41] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[42] The Tiny MD-HBase project on Github, 2017. `https://github.com/shojinishimura/Tiny-MD-HBase` - [Online; accessed 15-May-2017].

[43] New York City - Taxi and Limousine Commission - Trip Record Data, 2017. `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml` - [Online; accessed 15-May-2017].

[44] TPC BENCHMARK H (Decision Support) Standard Specification, 2017. `http://www.tpc.org/tpch/` - [Online; accessed 15-May-2017].

[45] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. SJMR: Parallelizing spatial join with MapReduce on clusters. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA*, pages 1–8, 2009.

Verzeichnis der zuletzt erschienenen Informatik-Berichte

[366] Lu, J., Güting, R.H.:
Simple and Efficient Coupling of a Hadoop With a Database Engine, 10/2012

[367] Hoyrup, M., Ko, K., Rettinger, R., Zhong, N.:
CCA 2013 Tenth International Conference on Computability and Complexity in Analysis (extended abstracts), 7/2013

[368] Beierle, C., Kern-Isberner, G.:
4th Workshop on Dynamics of Knowledge and Belief (DKB-2013), 9/2013

[369] Güting, R.H., Valdés, F., Damiani, M.L.:
Symbolic Trajectories, 12/2013

[370] Bortfeldt, A., Hahn, T., Männel, D., Mönch, L.:
Metaheuristics for the Vehicle Routing Problem with Clustered Backhauls and 3D Loading Constraints, 8/2014

[371] Güting, R. H., Nidzwetzki, J. K.:
DISTRIBUTED SECONDO: An extensible highly available and scalable database management system, 5/2016

[372] M. Kulaš
A practical view on substitutions, 7/2016

[373] Valdés, F., Güting, R.H.:
Index-supported Pattern Matching on Tuples of Time-dependent Values, 7/2016

[374] Sebastian Reil, Andreas Bortfeldt, Lars Mönch:
Heuristics for vehicle routing problems with backhauls, time windows, and 3D loading constraints, 10/2016

[375] Ralf Hartmut Güting and Thomas Behr:
Distributed Query Processing in Secondo, *12/2016*

[376] Marija Kulaš:
A term matching algorithm and substitution generality, *11/2017*