Distributed and Parallel Databases (2020) 38:439–493
https://doi.org/10.1007/s10619-019-07275-w

# BBoxDB: a distributed and highly available key-bounding-box-value store

**Jan Kristof Nidzwetzki**[1] · **Ralf Hartmut Güting**[1]

Published online: 15 November 2019


## Abstract

BBoxDB is a distributed and highly available key-bounding-box-value store, which is designed to handle multi-dimensional big data. To handle large amounts of data, the software splits the stored data into multi-dimensional shards and spreads them across a cluster of nodes. Unlike 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. Multi-dimensional data can be retrieved by using range queries, which are efficiently supported by indices. A space partitioner (e.g., a K-D Tree, a Quad-Tree or a Grid) is used to split 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. BBoxDB works with distribution groups, the data of all tables in 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. The joins are supported by a spatial index and executed in a distributed and parallel manner on the nodes of the cluster.

**Keywords** Distributed data store · Storage engine · Key-bounding-box-value store · Multi-dimensional data store

✉ Jan Kristof Nidzwetzki
jan.nidzwetzki@studium.fernuni-hagen.de

Ralf Hartmut Güting
rhg@fernuni-hagen.de

[1] Faculty of Mathematics and Computer Science, FernUniversität Hagen, 58084 Hagen, Germany

## 1 Introduction

Today, location-enabled devices are quite common. Mobile phones and cars equipped with small computers and GPS receivers are ubiquitous. These devices often run applications that use *location-based services*. For example, the navigation system of a car shows the direction to the next gas station or a restaurant shows its advertisement on all mobile phones within a 10-mile radius around itself. In this case, the data is a position in the two-dimensional space. However, 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.

Working with today's data stores and multi-dimensional data is very laborious. *Key-value stores* (KVS) are often used these days as a scalable data store for large amounts of data. The key in a KVS is the access path to the stored data. By knowing the key, the value can be directly retrieved. When the key is unknown a full data scan has to be performed to search for a certain value. Finding keys for data that is accessed by using only one dimension is easy. For data that needs to be accessed via multiple attributes, a key is hard to find. Figure 1 illustrates the problem by showing an example record from two different datasets.

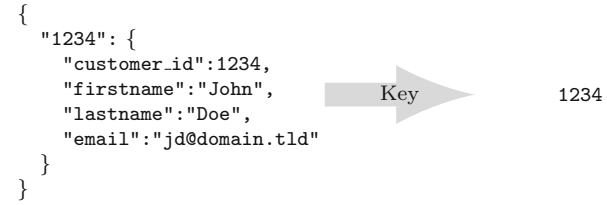
In Fig. 1a, a JSON encoded customer record is shown. The record needs to be retrieved only via the *customer_id* attribute (one-dimensional point data). Therefore, this attribute can be used as the key in the KVS[1]. In Fig. 1b, a record is shown that contains the geographical data of a road in the two-dimensional space. The roads should be stored in such a way that for a given query point all roads can be retrieved that intersect[2] this point.

Finding a proper key for the roads is difficult. In current applications, multi-dimensional data is often stored under a generic key (e.g., *road_66*) and a full data scan is performed, when a stored tuple should be retrieved. A full data scan is an expensive operation which makes the retrieval of multi-dimensional data in key-value stores very time-consuming.
We have identified seven major challenges when working with multi-dimensional data:

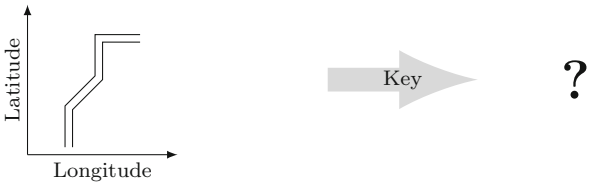
1. Multi-dimensional data have to be efficiently stored and retrieved (i.e., range queries should be efficiently supported) and full data scans should be prevented whenever possible.
2. Data of an $n$-dimensional relational table can be described as points in an $n$-dimensional space [54, p. 1]; spatial data in contrast has often an extent in space (e.g., the geographical data of a road). To support all common use-cases, point and non-point data (often called *regions*) have to be supported.
3. Data can change frequently (e.g., updating the motion vector of a moving car in a moving objects database [32]) and the data store has to deal with high update rates.
4. Datasets can become so large that they cannot be stored on a single node anymore; they need to be split up into parts and spread across a cluster of nodes.

[1] In Sect. 2.2 we discuss how $n$-dimensional point data could also be stored efficiently in a KVS.

[2] *"If a geometry or geography shares any portion of space then they intersect."* [39]




**(a)** A JSON encoded customer record which is accessed only via one attribute (via the customer_id attribute).




**(b)** A record that contains the geographical information of a road. The record is accessed via its coordinates in space (e.g., find the street that is located at certain coordinates).

**Fig. 1** Values in a KVS are stored under a unique key. The key is used later to retrieve the data. Only for data that is addressed via only one attribute, the key is easy to find

5. Datasets can grow and shrink and need to be redistributed dynamically to prevent over- or under-utilized nodes.
6. The availability of the stored data at all times is essential for most applications. Data redistribution should be performed in the background and should not affect the availability of the data.
7. Spatial joins are common queries on multi-dimensional data. Related data of multiple tables should be stored on the same hardware node (*co-partitioned*) to execute these joins efficiently and to prevent expensive data shuffling operations.

Our solution solves these challenges. Furthermore, this paper contains the following novel contributions:

1. The idea of a data store that is able to handle multi-dimensional point and non-point data is discussed.
2. A practical implementation of the idea is presented, which is available as open source software.
3. The novel concept of key-bounding-box-value tuples as a generic data model for multi-dimensional data is presented.
4. The idea of distribution groups for co-partitioned data is described and it is discussed how spatial joins can be executed efficiently.
5. Existing papers describe how data inserts can be handed, whereas updates are not discussed in these papers. Updates are more complicated because they have to invalidate old data. We discuss this problem and present some algorithms to handle updates efficiently.

The rest of this paper is structured as follows: Sect. 2 discusses the problem of storing multi-dimensional data in key-value stores in more detail. Section 3 describes

our solution: a key-bounding-box-value store called BBoxDB [47]. Section 4 contains an experimental evaluation of our development. Section 5 discusses the related work and Sect. 6 concludes the paper.

## 2 Background and motivation

In this section, a brief overview of the basic functions of key-value stores is given, and it is discussed why they have difficulties with the handling of multi-dimensional data.

### 2.1 Key-value stores

Nowadays, KVS are a popular type of data store. A KVS focuses 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)`. Both operations need a key as a parameter, which is usually a string; the value is usually an array of bytes. The `put` operation stores a value, which is identified by a unique key. The `get` operation uses a key to retrieve a previously stored value.

In many KVS no update or delete operations are implemented; this functionality is provided by the `put` operation. By calling the `put` operation with an already stored key, the stored value is overwritten; by calling the operation with a key and a `null` value, the stored data for a key is then deleted.

**Definitions:** In this paper, we call key-value pairs (and later key-bounding-box-value pairs) *tuples*. A set of tuples is called a relation and denoted as $R$. $|R|$ denotes the cardinality of the relation $R$.

If the KVS stores the tuples sorted by key, the `get` operation can be performed in $\mathcal{O}(\log |R|)$ time using *binary search*[3]. If a tuple needs to be retrieved without knowing the key (e.g., when retrieving all tuples that have a value containing the characters 'abc'), an expensive full data scan has to be performed. This operation has a time complexity of $\mathcal{O}(|R|)$.

To handle large datasets, many KVS are implemented as a *distributed key-value store* (DKVS). In a DKVS, the whole dataset is split into pieces (called *partitions* or *shards*) and stored on a cluster of nodes; each node contains only a small part of the data. Data partitioning is usually done by applying a *range-partitioning*[4] or a *hash-partitioning function*[5] to the key of the tuples. As the dataset grows, additional nodes can be used to store the data (*horizontal scaling*). To enhance data availability, the

[3] Most KVS use binary search on an key sorted storage to retrieve the tuples. For smaller datasets with a predictable size, hashing can be used and tuple retrieval can be done with a time complexity of $\mathcal{O}(1)$.

[4] Ranges of keys are assigned to the nodes. For example node $n_1$ stores the tuples whose keys begin with $[a - d]$ and node $n_2$ stores the data for the keys $[e - g]$ and so on. When a node becomes overloaded, the data is *repartitioned*. The old range is split into two parts and another node becomes responsible for one of the parts.

[5] A hash function is applied on the tuple's key. The value range of the function is mapped to the available nodes. The mapping determines which node stores which data. *Consistent hashing* [40] makes it possible to add and remove nodes without repartitioning the already stored data.

**Table 1** The used symbols in this paper

| Symbol | Description |
|---|---|
| $R$ | A relation. |
| $\|R\|$ | Denotes the cardinality of the relation $R$. |
| $C$ | A cluster of hardware nodes. |
| $n$ | A hardware node within a cluster $C$. |
| $s$ | The number of hardware nodes $n$ within a cluster $C$. |
| $P_R$ | A partitioned relation. |
| $\lambda$ | The replication factor of a partitioned relation. |
| $\overline{R}$ | The average number of tuples of the relation $R$ stored per node. |

data can be *replicated* and stored multiple times onto different nodes. In the event of a node failure, the data can be retrieved from another node.

**Definitions:** Let $C = \{n_1, n_2, ..., n_s\}$ be a cluster of $s$ hardware nodes. A relation $R$ is decomposed by a *partitioner* function into $p$ disjoint partitions $P_R = \{R_1, R_2, ..., R_p\}$. An *allocator* function $alloc()$ is used to assign these partitions to the nodes of the cluster ($alloc : P_R \rightarrow \mathcal{P}(C)$). We call $\lambda$ the *replication factor* such that $\exists\lambda \in \mathbb{N} \setminus \{0\} \forall R_i \in P_R : |alloc(R_i)| = \lambda$. A partition can be allocated to more than one node to increase data availability ($\lambda > 1$); the tuples are replicated and stored multiple times. $\overline{R}$ denotes the average number of stored tuples of the relation $R$ per node: $\overline{R} = \frac{\lambda \cdot |R|}{|C|}$. Table 1 summarizes the used symbols in this paper.

In a DKVS, the `get` operation needs $\mathcal{O}(\log \overline{R})$ time in the average case. By knowing the partitioner and the allocator function, it can be determined which node might store the tuple for the given key. This node is contacted and–like in a single computer KVS– the tuple can be retrieved using binary search. When a tuple needs to be retrieved and the key is unknown, a full data scan has to be executed to find the tuple. This can be done in $\mathcal{O}(s \cdot \overline{R})$ time. All nodes of the cluster are contacted and they have to scan the stored data for the desired tuple.

To summarize, the key of a tuple plays two roles in a DKVS: (1) it identifies a tuple clearly (e.g., in update and delete operations) and (2) it determines which node stores the tuple.

A KVS stores tuples; these tuples describe entities with one or more attributes[6]. The number of attributes determines the dimensionality of the entity. Each attribute of the entity can have an extent in space or not. If at least one attribute of the entity has an extent, we call the entity *non-point data* and *point data* otherwise.

**Definition:** Let $e$ be an entity with $n$ dimensions. $min_d(e)$ is the function that determines the lowest coordinate for the entity $e$ in the dimension $d$ with ($1 \leq d \leq n$). $max_d(e)$ is the function that determines the highest coordinate for the entity $e$ in the

[6] An entity describes an object with one or more attributes. A record consists of a collection of fields and represents the technical view of a stored entity. In this paper, we use both terms synonymously.

dimension $d$ with $(1 \leq d \leq n)$.[7] We call *e point data*, iff $\sum_{i=1}^{n}(max_i(e) - min_i(e)) = 0$ and *non-point data* iff $\sum_{i=1}^{n}(max_i(e) - min_i(e)) > 0$.

### 2.2 Multi-dimensional point data

KVS are optimized for point-data that is accessed using a single key. The key is the only access path to the data; alternative access paths are not directly supported by the KVS (like a secondary index in an RDBMS [52][p. 262]). However, entities often consists of multiple dimensions. For example, a customer entity consists, among other things, of the following attributes: a *customer_id*, a *firstname*, a *lastname*, and an *email_address*. In this example, the customer record has four dimensions.

Efficient handling of multi-dimensional data is not addressed in the DKVS used today. Projects like *GeoMESA*, *MD-HBase* or *Distributed SECONDO* (see Sect. 5 for a discussion of these systems) have implemented their own solutions for handling multi-dimensional data. In the academic world these days, MD-HBase [48] is one of the most popular papers in this area. MD-HBase is a multi-dimensional index layer built on top of the popular DKVS HBase. MD-HBase uses a K-D Tree or a Quad-Tree to partition an $n$-dimensional space ($\mathbb{R}^n$) into regions and then assigns these regions to HBase nodes. The paper of MD-HBase uses *buckets* to store the data. Several storage models for these buckets are discussed. A bucket could be a single small table that is located on one HBase node or a key-range of a larger table which is located on multiple HBase nodes. For better comparability we describe the *one bucket is one table* model here. Linearization [45] is used to encode the location of a tuple in the $n$-dimensional space into a one-dimensional key. MD-HBase keeps track of which part of the space is stored at which node in which bucket. MD-HBase does not index the content of the buckets, therefore a full data scan of the relevant bucket is required.

To store multi-dimensional data, all attributes can be encoded (nested) into a single tuple or the attributes are stored in individual tuples. The details about these two solutions will be discussed in the following sections.

#### 2.2.1 Nesting attributes

One way to store multi-dimensional data in a KVS is to store the complete record in one tuple. The record with all attributes is encoded into a single value and stored under a single key. To encode all attributes into a single value, formats like *comma separated values* (CSV) or *JavaScript Object Notation* (JSON) can be used.

Sometimes the value is additionally compressed or converted to a format that is easy to handle (like *Base64*). When a tuple is retrieved, the value is decoded and the individual attributes of the record can be accessed. Only the application that knows the encoding of the value can decode the data. For the KVS the value is a meaningless array of bytes. In case of plain JSON, the record is stored as a string and a text match

[7] When the data type of the attribute is not numeric (e.g., a string, a JPEG encoded image), the values need to be mapped to a numeric data type before the $min$ and $max$ function can be applied. This can be done in several ways, like with a *perfect hash function* [55], treating the bytes of the datatype as numbers or with a custom mapping function.

| Key | Value |
|---|---|
| 1234 | { "customer_id":1234, "firstname":"John", "lastname":"Doe", "email_address":"jd@domain.tld" } |

**Fig. 2** All attributes of the customer record are encoded into a single JSON encoded string. The complete JSON string is stored as a value in a KVS

against it is possible (e.g., to find the customer record which contains *john*), identifying individual attributes however is not possible for the KVS (e.g., to find the customer record with the first name attribute is equal to *john*).

When the record is stored in a KVS, a proper key needs to be determined. This key is the only access path to the record. In our example with the customer record, the customer id attribute might be the right choice for the key. However, this attribute has to be known when the records need to be retrieved; otherwise, a full data scan needs to be performed. Figure 2 shows an in JSON encoded customer record that is stored in a KVS.

### 2.2.2 Storing attributes individually

Applications often need to access tuples by several different attributes. Some KVS, like Cassandra or HBase, support multi-dimensional point data. They decompose the multi-dimensional data into several one-dimensional key-value pairs (like in a *column-oriented database*). The values are restricted to a particular data type (e.g., *string* or *integer*) so that the KVS can recognize these values and execute operations on them (e.g., string equality tests or numerical comparisons such as $<$, $>$ or $=$).

So far, the customer record from the example can be only retrieved by its customer id. If the record needs to be retrieved via additional attributes like *email address* or *last name*, the record can be decomposed into the three key-value pairs with the keys *customer_id:firstname*, *customer_id:lastname* and *customer_id:email*. The keys are now composed of two values: they consist of (1) a *partitioning key* (the customer id in this example) and (2) the name of the attribute of the record. The partitioning key is identical to the regular key in a KVS; it is used to identify the record and has to be unique per record. The partitioning key is the only part of the composed key which is used as input for the partitioning function. Therefore, it determines on which node all tuples of a record are stored (see Fig. 3).

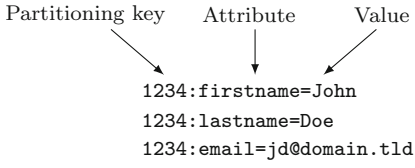



**Fig. 3** A customer record, decomposed into three key-value pairs. The key consists of two parts: the partitioning key and the name of the attribute

When a customer record is retrieved, all record related key-value pairs are read and the complete customer record is reconstructed. By splitting up a multi-dimensional tuple into several one-dimensional tuples and by restricting the type of the values to a particular data type, the KVS can access the individual attributes. When a customer needs to be retrieved by its email address, the KVS loads the values of the keys ending with *:email*. Then it is tested whether or not the value is equal to the searched value. If so, the *customerid* from the key is used to load and reconstruct the complete customer record. Additional indices[8] can be created on the values to execute the search in $\mathcal{O}(\log \overline{R})$ time on one node and in $\mathcal{O}(s \cdot \log \overline{R})$ on all nodes of the cluster.

By decomposing multi-dimensional point data into multiple-key value pairs and by building an index over additional attributes, tuple retrieval based on the partitioning key and these attributes can be efficiently executed. However, an additional index on further attributes does not solve the problem that the primary key is the only access path to the data. If the primary key is not part of the query, it is not possible to determine which nodes contain the data. The query is sent to all nodes which makes the query expensive and prevents a proper scale-up of a DKVS (see [18] for a more detailed discussion of this problem).

By knowing only the email address of the customer record, it is impossible to determine which node stores the record. All $s$ nodes of the cluster need to be contacted and all nodes need to check the locally stored data. This can be done in $\mathcal{O}(s \cdot \log \overline{R})$ time. Contacting all nodes might become a problem when the number of nodes is large. Moreover, running the retrieval on all nodes of the cluster consumes resources on all nodes which reduces the performance of the whole cluster. It would be better if only a few nodes (or ideally precisely one node) need to be contacted.

Current DKVS implementations like HBase or Cassandra do not provide support for multi-column secondary indices. HBase provides no direct support for such indices [36] and Cassandra provides only support for one attribute per index [19].

For example, this can become problematic in a *geographic information system* which stores the positions of traffic lights in a KVS. The traffic lights are given by the attributes `id`, `longitude` and `latitude`; the records of the traffic lights are decomposed by the attributes. The id of the traffic light is used as the partitioning key in the DKVS. Because the partitioning key has to be unique, the other attributes can not be used. When the traffic lights of a given area should be retrieved, all nodes of the cluster need to be contacted. Then all tuples with a matching `longitude` and `latitude` have to be reconstructed. It doesn't matter which attribute is considered first. For this attribute, an index can be created to find the tuples with a matching attribute efficiently. However, since the local secondary indices can only be defined on one column, the complete retrieval operation is expensive. For all tuples that are contained in the index, the remaining attribute (`longitude` or `latitude`) is loaded from disk and a filter operation is applied. The filter operation removes the tuples with an incorrect value in the remaining attribute. In this step, a lot of unneeded tuples are reconstructed which are removed by the filter operation from the result set.

[8] These indices are typically built independently on each node. Building a global index over the complete data requires a lot of coordination between the nodes; most DKVS are working with *local secondary indices*.

By using systems that are capable of handling multi-dimensional point data, the search for all traffic lights inside a given area can be efficiently executed. For example, the multi-dimensional index of MD-HBase improves the query time. A range query on the multi-dimensional index and the scan of one or more buckets can be performed in $\mathcal{O}(\overline{R} + t)$ time; where $t$ denotes the number of returned tuples of the operation.

### 2.3 Multi-dimensional non-point data

So far, it is discussed how multi-dimensional point data can be stored in a key-value store. Non-point data cannot be handled with the already presented techniques. For example, non-point data can describe the geographical information about a road or a range-value like the time period of a meeting in an RDBMS. Working with non-point data is harder than working with point data, because the extent of the data in space makes it hard to encode the location of the data into one value using linearization.

Systems like *Distributed SECONDO*, *Elastic Search* or *Spatial Hadoop* are capable of handling non-point data. But all of these systems have some drawbacks like supporting only a static dataset (read only and non-changeable data), static partitioning or the ability to handle only data of a certain dimensionality. Section 5 contains a detailed discussion of these systems.

Most existing approaches build an additional index layer on top of an existing storage engine. This causes some inefficiencies, for example, MD-HBase builds an index structure and its own data distribution technique on top of HBase tables. DISTRIBUTED SECONDO uses a static grid on top of Cassandra to handle multi-dimensional data. These solutions increase the complexity of the whole system and make the distribution techniques of the underlaying storage manager superfluous.

The lack of support for non-point data in most systems, the inefficiency that an additional layer introduces, the missing support for co-partitioned data (see Sect. 3.8) and the fact, that the source code of some systems is not publicly available, have motivated us to develop our own storage manager. Our storage manager is called BBoxDB, it is a complete new development from scratch and offers direct support for multi-dimensional data.

## 3 BBoxDB

In this section we present BBoxDB, a data store that is capable of handling multi-dimensional data. BBoxDB is a distributed and highly available key-bounding-box-value store that enhances the classical key-value data model with an axis-parallel bounding box. In BBoxDB, all bounding boxes are parallel to the Cartesian coordinate axis (so-called *axis-aligned minimum bounding boxes*). Bounding boxes are used as a generic data abstraction and they can be constructed for almost all types of data.

Data of an $n$-dimensional table in a *relational database management system* can be described as points in an $n$-dimensional space. A table with only one attribute can be described with a point in the one-dimensional space. *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.

A *space partitioner* is used to partition the $n$-dimensional space into smaller *distribution regions*. These regions are assigned to one or more nodes of the cluster, depending on the replication factor of the data. The distribution regions are created dynamically, based on the distribution of the stored data. Uneven data distributions are redistributed dynamically in the background without interrupting the data access.

The architecture of the whole system is highly available, no single point of failure does exist. BBoxDB is written in Java, licensed under the *Apache 2.0 license* [4] and can be downloaded from the website [10] of the project. In addition, BBoxDB images for *Docker* [11] are available on *Docker Hub* [8]. We also provide a configuration for *Docker Compose* [9] that allows the user to setup a virtual cluster of five BBoxDB nodes and three ZooKeeper nodes within two minutes.

### 3.1 A key-bounding-box-value store

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 location of the value in an $n$-dimensional space.

#### 3.1.1 Bounding boxes

**Definition:** The *axis-aligned minimum bounding box*, which is simply called bounding box in this paper, for an $n$-dimensional object is the smallest $n$-dimensional hyperrectangle within which the complete object lies.

An $n$-dimensional bounding-box is represented by an $n$-dimensional hyperrectangle in BBoxDB. The hyperrectangle is described as a tuple consisting of $2 * n$ values of the datatype `double` (see Listing 1 on Page 26). For each dimension of the hyperrectangle, an interval with a start and end point is created. The element $2 * (n - 1)$ of the tuple describes the lowest included coordinate in the dimension $n$, the element $2 * (n - 1) + 1$ describes the highest included coordinate in the dimension $n$. For example, the tuple $(1.3, 5.2, -5.1, 10.2)$ describes a two dimensional hyperrectangle. In the first dimension the range $[1.3, 5.2]$ and in the second dimension the range $[-5.1, 10.2]$ are covered.

In Sect. 2.1, the $min$ and $max$ functions on entities are defined. These functions can be used to calculate the bounding box for an entity. For each dimension of the entity, the minimum and maximum coordinate are calculated to determine the bounding box. For a one-dimensional entity, the bounding box can be calculated as follows: $(min_1(e), max_1(e))$. For a 2-dimensional entity $e$ (like the *road* in Fig. 1(b)), the bounding box can be calculated as follows: $(min_1(e), max_1(e), min_2(e), max_2(e))$. Figure 4 illustrates the calculation of the bounding box for a *road*.

For $n$-dimensional entities which contain only point attributes (i.e., a string encoded *surname*, and an integer encoded *age*), the bounding box can be calculated in the same

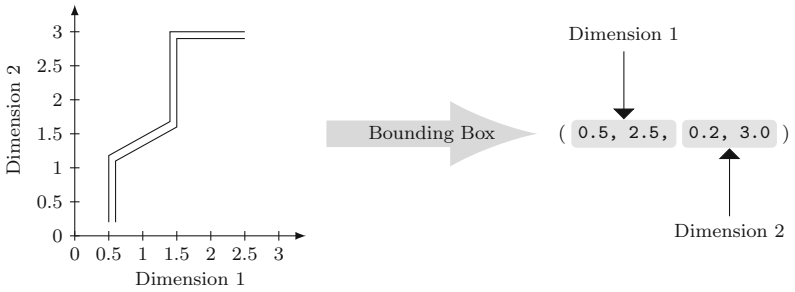



**Fig. 4** The calculation of the bounding box for a two-dimensional non-point entity (e.g., a road). In each dimension, the minimum and maximum coordinate are calculated and used to determine the bounding box

way. An entity consisting only of point attributes has no extent. The result of the $min$ and $max$ function on point data is equal for this entity. Therefore, the bounding box degenerates to a point in space. The point is located at the same coordinates as the $n$-dimensional data point. Listing 1 on Page 25 contains some examples, how bounding boxes for point and non-point data can be specified in the software.

To summarize, the $min$ and $max$ function can be used to calculate bounding boxes for entities and the bounding box of an $n$-dimensional point entity is an $n$-dimensional hyperrectangle, which degenerates to an $n$-dimensional point.

### 3.1.2 The data model of BBoxDB

When storing data in BBoxDB, a value and a suitable bounding box must be passed to the `put()` method (see Sect. 3.6 for an example). In Line 6 in Listing 2 (page 26) is shown, how a bounding box can be created and passed to the put function of BBoxDB. The specification of the bounding box causes a little more work for application developers; they have to build a function that calculates the bounding box. However, the creation of the bounding box solves the problem that only a one-dimensional key and an array of bytes are passed as a value to the data store. A generic data store like BBoxDB is unable to understand the semantics of the value; all possible data encodings and data formats can be used by a user as value (see Sect. 2.2.1). Therefore, the data store is unable to calculate the bounding box on is own for any data type. Only the user who generates the value knows how to interpret the bytes of the value and how to determine the bounding box. Calculating the bounding box is a simple task for most data types. For example, the bounding box for $n$-dimensional spatial data can be calculated by calling the `min()` and `max()` function on the data object for every dimension.

The bounding box is a generic way to specify the location of the value in space. By knowing the location in space, 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 Sect. 3.4). A major difference between a KVS and BBoxDB is that the key of a tuple does not determine the node that is responsible for the tuple; this is determined by the bounding box of the value and the partitioning of the space.

A tuple in BBoxDB is defined as $t$ =(*key*, *bounding box*, *value*, *timestamp*). The key is used to clearly identify a tuple in update or delete operations. The bounding box describes the location of the tuple in the $n$-dimensional space, the value contains the data to store. The timestamp is used to keep track of the most recent version of the tuple. Based on the timestamp of the tuple, BBoxDB provides *eventual consistency* [58]. When no updates are made to the stored data, all replicates become eventually synchronized with the last version of the data. This consistency model allows it to deal with network partitions or unavailable replicates. Internally, BBoxDB provides some methods to ensure that replicates become synchronized when they have lost the connection to the cluster[9].

### 3.1.3 Organization of tuples

A *distribution group* is a collection of *tables*. All these tables are using the same global index; therefore they are partitioned in the same way and spread in the same manner across a cluster of systems (see Sect. 3.4). This means that all the tables of one distribution group are stored co-partitioned. Co-partitioning is the foundation to execute joins in BBoxDB in an efficient manner (see Sect. 3.8). A *table* is a collection of tuples and each table belongs to one distribution group. All tables of one distribution group are distributed in the same manner across the cluster. In BBoxDB, the $n$-dimensional space is split into disjoint spaces, called *distribution regions*. A *distribution region* is a subspace of an $n$-dimensional space. The tables in BBoxDB are split into *region tables* according to the partitioning of the distribution group (see Fig. 5). Depending on the replication factor of the distribution group, one or more BBoxDB instances are responsible for storing the region tables of a distribution region.

Tuples whose bounding box belongs to multiple distribution regions (e.g., *Tuple G* in Fig. 5) are duplicated and stored multiple times. In this case the tuple is automatically send to multiple nodes by BBoxDB. *Tuple G* is stored in the nodes a and c. When a read operation is performed, duplicates are automatically removed from the result set by a filter in the client code of BBoxDB.

BBoxDB can be used to store data of any dimension. Even one-dimensional data (i.e., data that are stored in a regular key-value store) can be handled by this system. The bounding box of one-dimensional data is simply a point in a one-dimensional space. Therefore, BBoxDB can be used as a replacement for a regular KVS and work with such data.

[9] DKVS like Cassandra are also providing eventual consistency to ensure that the system can deal with network or node outages. Eventual consistency does not mean that replicates become outdated for a long time. During the regular operation of the cluster without outages, the replicates are updated with every write operation [41, p. 162ff]. However, when outages occur, replicates can contain outdated data for a longer period before they eventually become synchronized with the last version of the data. Cassandra uses techniques like timestamps, gossip or read repair to ensure the replicates become synchronized. BBoxDB implements similar techniques. However, BBoxDB does not make improvements to these techniques. Therefore the implementation is not covered in this paper. Some implementation details can be found in the documentation of BBoxDB. [12].

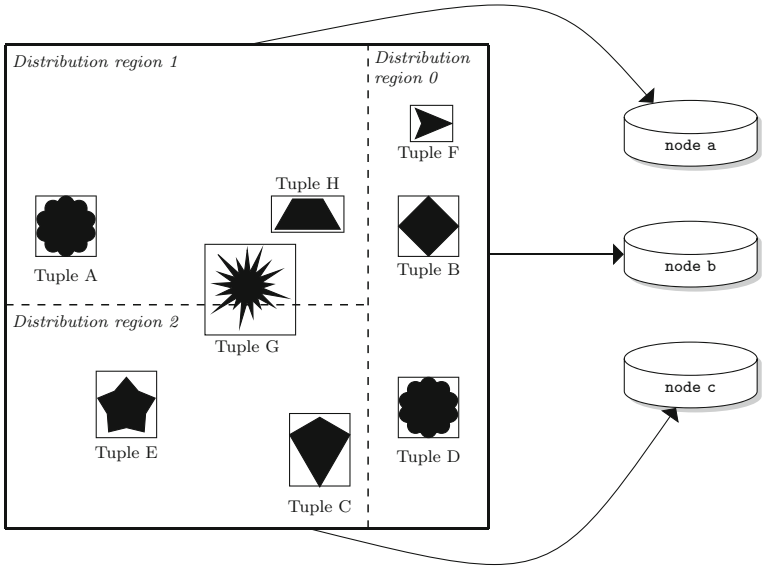



**Fig. 5** BBoxDB partitions the space into distribution regions and assigns these regions to several nodes. The different symbols of the tuples represent the different values. The box around the symbol represents the bounding box of the value

## 3.2 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 manipulate 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 2 presents an overview of the supported operations.

**Table 2** The operations that are supported by BBoxDB

| | | |
|---|---|---|
| **Data definition:** | | |
| `createDGroup` | string × dgroup_config | → dgroup |
| `deleteDGroup` | string | → boolean |
| `createTable` | string × table_config | → table |
| `deleteTable` | string | → boolean |
| **Data manipulation:** | | |
| `put` | table × string × hrect × bytes | → table |
| `delete` | table × string | → table |
| `getByKey` | table × string | → tuple(...) |
| `getByRect` | table × hrect | → stream(tuple(...)) |
| `join` | table × table × hrect | → stream(tuple(...)) |
| `lock` | table × string | → boolean |

The operations `createDGroup(groupname, group_configuration)` and `deleteDGroup(groupname)` are used to create and delete distribution groups. When a distribution group is created, additional parameters have to be specified, such as the dimensionality and the replication factor. The two operations `createTable(tablename, table_configuration)` and `deleteTable(tablename)` are used to create and delete tables. When a table is created, some configuration parameters can be specified.

New data are stored by the operation `put(table, key, hyperrectangle, value)`. In contrast to a put operation in a KVS (which expects the parameters `table`, `key` and `value`), a bounding box (the hyperrectangle) has also to be specified. How this can be done is shown in the Java code example in Sect. 3.6. The operation `delete(table, key)` removes data from a table. The operation `getByKey(table, key)` receives the data for a given key. The operation `getByRect(table, hyperrectangle)` is a range query and receives all data whose bounding box intersects with a given query hyperrectangle. The operation `join(table1, table2, hyperrectangle)` executes a spatial join based on the bounding boxes of two tables. The tuples of both tables that have bounding boxes which intersect within a particular hyperrectangle are returned. The `lock(table, key)` operation locks a tuple against updates from other clients. This method is intended to coordinate updates. Section 3.9 contains a detailed discussion of this operation.

### 3.3 Used technologies

BBoxDB employs existing technologies to accomplish its work. The basics of these technologies (i.e., Apache ZooKeeper and SSTables) are described in this section. A basic knowledge of these technologies is necessary in order to understand how BBoxDB works.

#### 3.3.1 Apache ZooKeeper

Building a distributed system is a complex job. Distributed tasks need to be coordinated correctly in a fault-tolerant manner. *Apache ZooKeeper* [38] is a software which was developed to ease the construction of distributed systems. A large number of popular systems (e.g., Apache Hadooop or Apache HBase) are using ZooKeeper to coordinate distributed tasks. In [38, p. 3] the developers of ZooKeeper compare the data model of ZooKeeper with a filesystem where distributed applications can store their metadata or configuration.

ZooKeeper provides a highly available tree (similar to a hierarchical directory structure in a file system) that can be accessed and manipulated from client applications. The tree consists of nodes that have a name (similar to a directory in a file system). 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`.

Different types of nodes are supported by ZooKeeper. The two important node types are: (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. ZooKeeper is used by BBoxDB for two purposes: (1) *storing the global index* (see Sect. 3.4) and (2) *service discovery*.

BBoxDB is a distributed system, service discovery makes it possible to determine which BBoxDB instances are available and on which IP and on which port they can be reached. Each BBoxDB instance creates an ephemeral node at initialization; for example, a child of the node `/nodes`. The name of the node contains the local IP address and the port number that this BBoxDB instance is listening to. By reading the children of the node `/nodes`, the connection points of all BBoxDB instances are known. When a BBoxDB instance crashes or terminates, the ephemeral node gets automatically deleted by ZooKeeper. Also, each BBoxDB instance establishes a watcher on the children of the path `/nodes`. Therefore, each BBoxDB instance gets notified when the group of running BBoxDB instances changes.

### 3.3.2 SSTables

In the Introduction (Sect. 1) we have presented seven challenges when working with large amounts of multi-dimensional data. The third challenge is the support for high update rates. *String Sorted Tables* (SSTables) [49] (also called *Log-Structured Merge-Trees* [20]) are a write-optimized data structure [5]. SSTables have shown a better write performance than commonly used data structures such as the B-Tree [41, p. 83]. They are used by many modern NoSQL databases such as Cassandra or HBase. To achieve a good performance for write operations, we have also chosen SSTables as basic data structure for BBoxDB. To support the spatial operations (range queries and joins) that are provided by BBoxDB in the best possible way, we combined SSTables with a multi-dimensional index.

In SSTables, data are stored as key-value pairs in key-sorted files. The two main goals of this data structure are (1) to provide a high write throughput and (2) a reduced random disk IO. Modifications–such as changing or deleting data–are performed simply by storing a new version of the data. Deletions are performed 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* (see Fig. 6). 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 to 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

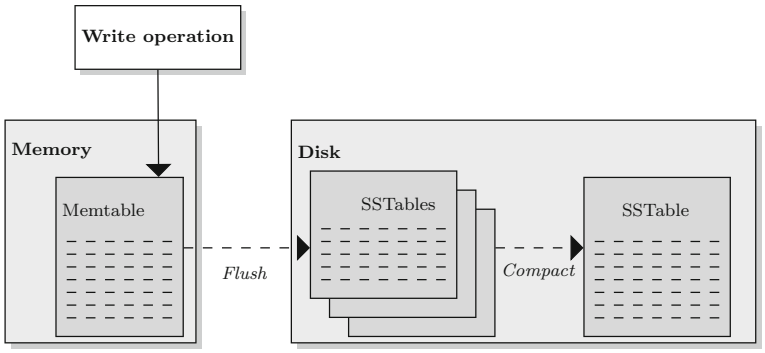



**Fig. 6** The relationship between Memtables, SSTables and compactifications

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.

SSTables are optimized to retrieve a tuple via its key. This is implemented by an index that stores the byte positions of the tuples. The tuples have a variable length, therefore an index is required to access a certain record directly. To find a tuple for a given key, a binary search is performed on this index. In addition, all SSTables and Memtables are associated with a *Bloom filter* [15]. 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. As a probabilistic data structure the bloom filter can contain false positives, but not false negatives. This means that the table does not need to be read, when the bloom filter does not contain the key. In the other case, the above mentioned binary search is performed and it is checked if the key is contained. In this case the complete tuple is read from the table.

BBoxDB stores key-bounding-box-value pairs in SSTables. Retrieving all tuples for a certain region in space is not efficiently supported by regular SSTables. However, this is a common use case in BBoxDB. Therefore, we wrote a new SSTable implementation that is capable of handling range queries based on the bounding boxes of the data. To handle tuple retrieval via a range query, we added an R-Tree to the SSTables data structure. The R-Tree is created, when the Memtables are materialized on disk as SSTable or when a compactification is executed. The R-Tree contains the bounding boxes of the tuples and references to the byte positions of the tuples in the SSTable file (see Fig. 7). In addition, we wrote a complete new R-Tree implementation because the existing libraries for Java are not capable of handling $n$-dimensional point and non-point data (the second challenge from the Introduction). In general, every multi-dimensional index structure can be used at this point. We have chosen the R-Tree in our implementation, because it is a very well researched index structure. The coupling between the SSTables and the index structure is made via a generic interface. Therefore, the R-Tree index can be easily replaced by other index structures if needed. A detailed evaluation of alternatives to the R-Tree for the local index, is a topic for further research.

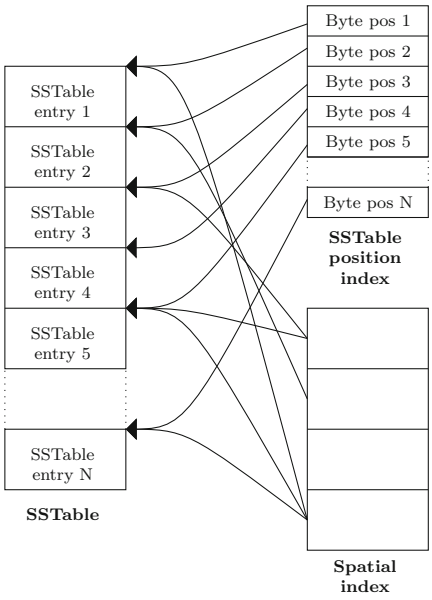



**Fig. 7** One SSTable, the position index and the spatial index. The spatial index references the start positions of the tuples withing the SSTable. The spatial index allows range queries over the bounding boxes of the stored data

The spatial index is stored in a separate file on disk, which increases the needed disk space for the stored data. Each index entry consists of a bounding box and a pointer to the byte position for the value in the SSTable. The size of the bounding box depends on the dimensionality, for each dimension two 64-bit numbers are stored together with a 64-bit pointer. In addition, the data structure of the R-Tree is stored in the index. In general, the index does not need much storage. For an SSTable with 100 000 entries and two-dimensional bounding boxes, a typical size for the index is around 6.5 MB. The actual SSTable is much larger and the size is primarily determined by the stored data.

### 3.4 Accessing tuples efficiently

The first of the seven challenges from the Introduction is the efficient access to multi-dimensional data. The fourth challenge is to spread large datasets efficiently across a cluster of systems. In this section, the two-level index structure and the space partitioner are described. The two-level index structure (global index and local index) is used to access tuples efficiently. The space partitioner is an algorithm that builds up the global index.

Two-level index structure: BBoxDB works with a two-level index structure. The *global index* maps from distribution regions to nodes; this index determines which node is responsible for which part of the space. A *local index* is maintained locally on each node. The local index maps from the space of a distribution region to the tuples stored on the node.
The main purpose of splitting up the index into two pieces

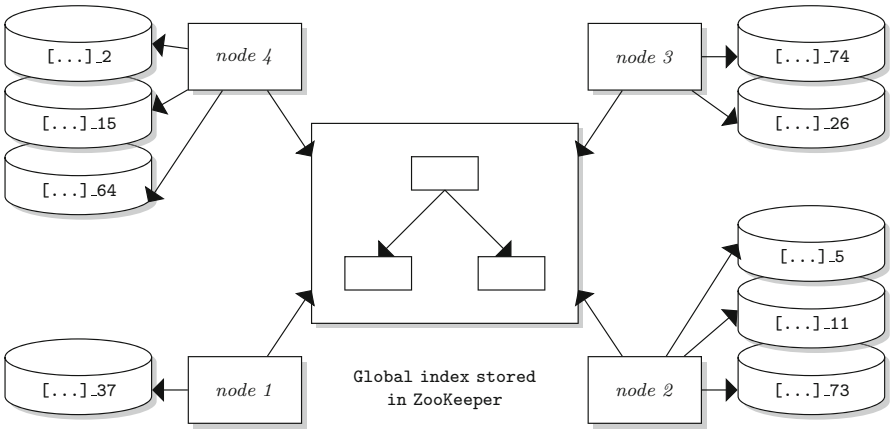



**Fig. 8** A BBoxDB cluster with four nodes. Each node stores the data of multiple distribution regions

is to reduce the costs of maintaining the index. The local index contains references to tuples and has to be changed every time new data is stored. The local index can be altered without coordination between the nodes, which makes changes cheap.

The global index determines the partitioning of the space and is stored in *Apache ZooKeeper* (see Fig. 8). The global index only needs to be modified when the partitioning of the space is changed. This is an expensive operation; the already stored data has to be redistributed (see Sect. 3.5) and operations (e.g., read and write requests) need to be redirected to other nodes (see Sect. 3.4.2). However, the partitioning of the space is relatively stable and needs to be changed only after huge changes in the dataset. The global index is created by the space partitioner and the local index structure is provided in our implementation by an R-Tree [33].

Space Partitioner:

BBoxDB allows the use of different data structures as space partitioner. The space partitioner is also responsible for building and maintaining the global index. At the moment *K-D Trees* [14], *Quad-Trees* [25] and a *fixed grid* are supported. Depending on the used data structure, BBoxDB provides different features. Some data structures support dynamic partitioning; others do not. Some space partitioners support splitting the space at any desired point; while others work with fixed split points.

When the space is partitioned and a tuple belongs to more than one distribution region, the tuple is duplicated and stored in all of these regions. To prevent duplicates in query results, these duplicates are automatically removed

from the result set when a query is executed. This is done by a hash table in the client implementation (see Sect. 3.6), which eliminates duplicates based on the tuple key.

The K-D Tree is a very flexible data structure—it supports the dynamic partitioning of the space and can split the space at every desired position. The Quad-Tree also supports dynamic partitioning but can only split the space at fixed positions. The fixed grid is the less flexible data structure. The space is split up into fixed cells and the data is not partitioned dynamically. The fixed grid is primarily implemented in BBoxDB for comparison experiments with other space partitioners. A detailed analysis of the different space partitioners is omitted, analyzing the suitability of space partitioners is part of our current research and will be published in a further paper. However, the K-D Tree is used in the following sections as the space partitioner.

### 3.4.1 Implementation details of the global index

The global index consists of entries stored in ZooKeeper. The K-D Tree from Fig. 9 is used as an example in this section; it is shown how this K-D Tree is stored in ZooKeeper.

ZooKeeper provides a highly available tree which can be accessed from several clients (see Sect. 3.3.1). The K-D Tree of the global index (and in general, the data structure of all space partitioners) needs to be converted into a structure that is compatible with ZooKeeper.

Each node of the K-D Tree is represented as at least one entry in ZooKeeper. Additional information, such as the region state or the nodes that are responsible for the region, are stored as additional entries. Table 3 shows all the entries stored in ZooKeeper to describe the K-D Tree from Fig. 9.

The region with the id 0 (the upper part of the table) is the root region that covers the whole space (see the value of node `[...]/space`[10]). The region is split at point 5.0 in the first dimension (see the values of the child nodes `[...]/child-1/space` and `[...]/child-2/space`). The region with the id 0 is split and all data is redistributed to the child regions. This is indicated by the `[...]/state` node. In Table 4 the possible states and the meaning of the states are described. Section 3.5 contains a description how the global index can be changed and how the states are used to ensure the global index is always in a consistent state.

[10] Usually, only the split positions are stored in a K-D Tree. The data structure in ZooKeeper instead contains a description of the space which is covered by a region. This makes it possible to reuse this data structure with other space partitioners (e.g., the Quad-Tree or the grid).

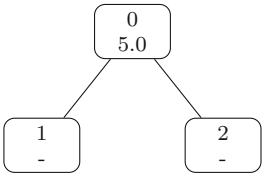



**Fig. 9** A small K-D Tree with three nodes. Each node is equal to a distribution region. The parent region with the id 0 is split at position 5.0, the child regions 1 and 2 are not split

**Table 3** The nodes of a 2-dimensional distribution group with three regions. The region 0 is split and inactive, the regions 1 and 2 are active. The K-D Tree is split at point 5.0 in dimension 0. To improve readability, a part of the path has been omitted. This is indicated by the symbol `[..]`. In the concrete implementation, the path is enhanced by a unique name of the installation and the name of the distribution group

| Node | Value | Description |
|---|---|---|
| `[..]/nameprefix` | 0 | A sequential number, used to name distribution regions. |
| `[..]/state` | `SPLIT` | The state of the distribution group determines whether or not requests are sent to this region. |
| `[..]/nodes` | `[...]` | A list with BBoxDB instances which are responsible of storing the distribution group. |
| `[..]/space` | `[[min,max]-[min,max]]` | The space that is covered by this region. |
| `[..]/statistics` | `[...]` | Statistics (e.g., amount and size) of the stored tuples in this region. |
| `[..]/node-version` | `[...]` | The timestamp when the data was changed the last time. |
| `[..]/child-1/nameprefix` | 1 | ” |
| `[..]/child-1/state` | `ACTIVE` | ” |
| `[..]/child-1/nodes` | `[...]` | ” |
| `[..]/child-1/space` | `[[min,5.0)-[min,max]]` | ” |
| `[..]/child-1/statistics` | `[...]` | ” |
| `[..]/child-1/node-version` | `[...]` | ” |
| `[..]/child-2/nameprefix` | 2 | ” |
| `[..]/child-2/state` | `ACTIVE` | ” |
| `[..]/child-2/nodes` | `[...]` | ” |
| `[..]/child-2/space` | `[[5.0,max]-[min,max]]` | ” |
| `[..]/child-2/statistics` | `[...]` | ” |
| `[..]/child-2/node-version` | `[...]` | ” |

### 3.4.2 Changing the global index

The global index needs to be altered when the partitioning of the space is changed. The global index is stored in ZooKeeper and all BBoxDB clients store an in-memory copy

**Table 4** The possible states of a distribution region in BBoxDB. Depending on the state, read and write operations are executed on the region or not

| 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 its maximum size and gets split soon. | Yes | Yes |
| `SPLITTING` | The data are spread to the child nodes at the moment. | Yes | No |
| `SPLIT` | The region has been split and the data are spread to the child nodes. | No | No |
| `MERGING` | The region is merging at the moment. | Yes | No |

of this data structure. When the global index is read from ZooKeeper, *watchers* (see Sect. 3.3.1) are established to get notified by ZooKeeper when the data is changed. As soon as a client gets a notification from ZooKeeper, the global index is partially re-read and the in-memory copy is updated.

It takes some time between a change of the ZooKeeper version of the global index and the moment when all clients have updated their in-memory version. When the data is changed at time $t$, it takes up to $t_\Delta$ time until the change is applied on all nodes. In the time between $t$ and $t + t_\Delta$, operations can be sent to the wrong regions. This needs to be handled in BBoxDB.

Most operations in BBoxDB are based on hyperrectangles (see Table 2). Before such an operation is performed, the client needs to determine which nodes are responsible for the area in space. This is done by a query of the local copy of the global index. The local copy of the global index can become outdated. To detect and handle outdated data in the local copy of the global index, the client adds the names of the distribution regions, on which the operation needs to be performed, as parameter to the operation.

For example, when a write operation is sent to a BBoxDB node, the names of the regions, where the operation has to be performed, are also contained in the request. Each server that receives such an operation queries its version of the global index and checks whether its result is equal to the request. If the two results differ, a *reject message* is sent back to the client. On receiving a reject message, the client waits a short period, re-checks the regions on which the desired operation needs to be executed and retries the operation. This is repeated until the operation is successfully completed. If no BBoxDB server sends a reject message back to the client, the operation is successfully executed on all regions.

### 3.5 Redistributing Data

Adding or deleting data can cause an unbalanced data distribution between the nodes; some nodes store more data than others (the fifth challenge from the Introduction). BBoxDB automatically redistributes unevenly distributed data by changing the distribution regions. When a distribution group is created, the minimum and maximum

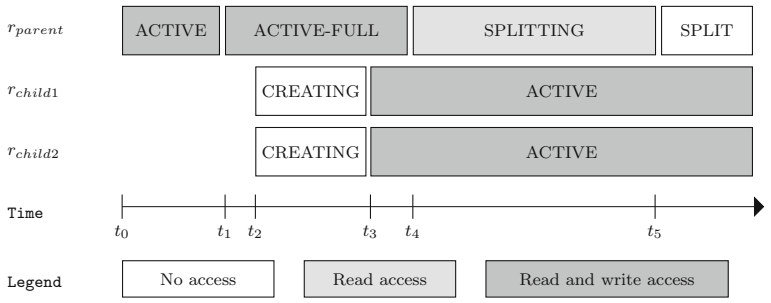



**Fig. 10** The chronological sequence of the states during a distribution region split

size per region (the number of the stored tuples) have to be specified ($t_{overflow}$ and $t_{underflow}$, with $t_{overflow} > t_{underflow}$). When the size of a distribution region is larger than $t_{overflow}$, the region is split and the existing data are redistributed. In the following text, we refer to the region which is split as *parent region* ($r_{parent}$) and the newly created regions as *child regions* ($r_{child}$). We call two or more distribution regions *mergeable* when the space partitioner is able to merge these regions. When the number of stored tuples of some mergeable distribution regions is smaller than $t_{underflow}$, the regions are merged. With the K-D Tree space partitioner, all the leaf nodes of the same parent can be merged.

When distribution regions are merged or split, the stored data need to be redistributed. The data redistribution is performed in the background; at all times, read and write access to the data is possible (the sixth challenge from the Introduction). Every distribution region has a state which is also stored in ZooKeeper (see Sect. 3.4). This state determines, whether read or write operations are sent to this region or not.

**Splitting a distribution region:** Every node checks periodically the size of the locally stored distribution regions. When a distribution region needs to be split, a split point is determined and the new distribution regions are created. Figure 10 depicts the chronological sequence of the states during a split. Table 4 contains an explanation of the states.

By using the K-D Tree space partitioner, the split point is determined by reading samples of the data of all tables of the distribution region $r_{parent}$[11]. The samples are used to choose a point which splits the region into two distribution regions which store an almost equal amount of tuples. The new distribution regions are called $r_{child1}$ and $r_{child2}$ (see time $t_2$ in Fig. 10). The main idea of the different states is that write requests are sent as soon as possible to $r_{child1}$ and $r_{child2}$ and read operations are executed on $r_{parent}$, $r_{child1}$ and $r_{child2}$ until all data are redistributed completely. After the data is redistributed completely, the read operations are only executed on the new created

[11] On a replicated distribution group, several nodes could notice at the same time that the region needs to be split. This could lead to an unpredictable behavior of BBoxDB. To prevent such situations, the state of the distribution region is changed from `ACTIVE` to `ACTIVE-FULL` (see time $t_1$ in Fig. 10). ZooKeeper is used as a coordinator that allows exactly one state change. Only the instance who performs this transition successfully executes the split.

regions. At any time, read and write requests can be handled; the data redistribution does not affect the availability of the data.

After the $r_{child1}$ and $r_{child2}$ are created, a *resource allocator* is used to choose which nodes should store the data of the region[12]. After the nodes are allocated to the child regions, the state is set to `ACTIVE` (see time $t_3$ in Fig. 10). Once the child regions are ready, the state of the node of $r_{parent}$ is set to `SPLITTING` (see time $t_4$ in Fig. 10).

Between $t_3$ and $t_4$, read and write requests are executed on $r_{parent}$, $r_{child1}$ and $r_{child2}$. From time $t_4$ on, read operations are executed on $r_{parent}$, $r_{child1}$ and $r_{child2}$ and write operations are only executed on $r_{child1}$ and $r_{child2}$. In addition, the data is transferred from $r_{parent}$ to $r_{child1}$ and $r_{child2}$. The data of region $r_{parent}$ contains all tuples for the region, but the data might be outdated (write requests are executed only on the new child regions). The data of the regions $r_{child1}$ and $r_{child2}$ contain the most recent version of a tuple, but the data is incomplete (the data redistribution is in progress). To ensure that a client gets the newest version of a tuple, data is read from all regions. The timestamp of the tuple (see Sect. 3.1.2) is used to eliminate outdated tuples from the result set of a query.

As soon as the data are completely transferred, the state in ZooKeeper is changed for $r_{parent}$ to `SPLIT` (see time $t_5$ in Fig. 10). Read and write requests are only sent to $r_{child1}$ and $r_{child2}$ and the data of $r_{parent}$ are deleted. To reduce the required data transfer, one of the newly created regions is stored on the same node(s) (depending on the replication factor). The data can be copied locally on the node without creating network traffic. Algorithm 1 shows the procedure of splitting a region in pseudo code.

**Algorithm 1** Split the global index by using the K-D Tree space partitioner

```
1: procedure SPLITGLOBALINDEX(index, region)
2:     samples ← read samples from all tables in region
3:     splitPos ← determine a good split position by using samples
4:     set the split position of region to splitPos
5:     create two new child regions of region
6:     assign nodes to the new regions depending on the replication factor
7:     set the state of the new regions to ACTIVE
8:     redistribute data from the old region to the new ones
9:     set the state of the old region to SPLIT
10: end procedure
```

**Merging a distribution region:** When multiple regions (e.g., $r_{child1}$ and $r_{child2}$) are smaller than $t_{underflow}$, they are merged into the parent region $r_{parent}$. During the merge operation, almost the same changes as in a split operation are executed in the global index, but in reverse order. First of all, the states of the affected regions are changed. Write operations are only executed on $r_{parent}$ and read operations are executed on all three regions. This is done by setting the state of the parent region

[12] BBoxDB contains different resource allocators, which choose the nodes based on the available hardware such as free disk space, number of harddisks, total memory or total CPUs. The used resource allocator can be chosen when a distribution group is created.

to ACTIVE and the state of the child regions to MERGING. Afterwards, the data are copied from $r_{child1}$ and $r_{child2}$ to $r_{parent}$. When the data transfer is complete, the regions $r_{child1}$ and $r_{child2}$ are deleted and the merge process is done.

Both changes of the global index (splitting and merging) are implemented in a manner, that the stored data is available at any time. In the first step, newly written data are stored in the new distribution region and the old region is changed into a read-only state and the data are transferred from the old into the new region. At this moment, read operations are executed on the old and on the new regions to ensure that the requested data can be retrieved. When the requested tuple is part of the old and the new region, the duplicates are removed from the result set (see Sect. 3.4.2). When all data have been transferred, the read requests are only executed on the new region(s) and the change is done.

**Prepartitioning:** At the moment, when a new distribution group is created, the complete space consists of one distribution region. When a dataset is imported into this distribution group, the space is split and the data is redistributed several times. It takes some splits of the space until all nodes of the cluster have at least one distribution region assigned. When the dataset is known, *prepartitioning* allows to utilize the resources of the cluster better and to omit some expensive data redistributions. BBoxDB ships with a utility, which reads samples from a static dataset and uses these samples to prepartition the space. This ensures that a sufficient number of distribution regions are created and assigned to the nodes of the cluster. When the dataset is imported by using prepartitioning, the data are directly written to several nodes. In contrast to regular distribution regions, the regions created by prepartitioning are not automatically merged. Otherwise, BBoxDB would immediately merge the regions, since no data are stored in them.

### 3.6 The Java-Client

BBoxDB ships with a client implementation for the programming language Java (the *Java-Client*). The Java-Client is available at *Maven Central* [7]; 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 nodes are automatically identified due to the changes in the directory structure of ZooKeeper.

The client uses the *future pattern* [6]; each operation that requires network communication returns a future instead of 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.

Hyperrectangles are used in BBoxDB to describe bounding boxes (see Sect. 3.1.1). The class Hyperrectangle can be used to create such hyperrectangles in Java

code. The constructor of the class takes $2n$ double values to create an $n$-dimensional hyperrectangle. Listing 1 shows some examples of the usage of this class.

**Listing 1** Usage examples of the hyperrectangle class of the Java-client

```
1  // A bounding box for one-dimensional point data
2  Hyperrectangle bbox1 = new Hyperrectangle(1.0, 1.0);
3
4  // A bounding box for two-dimensional non-point data
5  Hyperrectangle bbox2 = new Hyperrectangle(-3.5, 5.0, 0.0, 1.0);
6
7  // A bounding box for three-dimensional non-point data
8  Hyperrectangle bbox3 = new Hyperrectangle(0.0, 5.0, 0.0, 1.0, -2.0, 6.0);
9
10 // A bounding box for four-dimensional point data
11 Hyperrectangle bbox4 = new Hyperrectangle(0.0, 0.0, 1.0, 1.0, 6.0, 6.0, -4.0, -4.0);
```

In Line 2 of Listing 1, a bounding box for one-dimensional point data is created. The start and the end point in the constructor are identical to describe an element with no extend. In Line 5 a bounding box for 2-dimensional non-point data is created. In the first dimension, the interval from -3.5 to 5.0 and in the second dimension the interval from 0.0 to 1.0 is covered by the hyperrectangle. In Line 8 a three-dimensional bounding box for non-point data is created, and in Line 11 a bounding box for four-dimensional point data is created.

Listing 2 shows an exemplary usage of the Java-Client. A new tuple is inserted into the table `group_table1` (line 10), it can be seen that a key (`key1`), a bounding box and some data (the content of the variable `value`) are passed to the `put` operation. The `put` operation is executed asynchronously, due to the future pattern. In line 12, the program waits until the operation is complete. Afterwards, a range query is executed on this table (line 17) and all found tuples are printed (line 23).

### 3.7 Tuple retrieval time complexity

In the introduction, the time complexity of accessing multi-dimensional data in existing data stores is discussed. In the previous section, we have discussed the two-level indexing used in BBoxDB. Operations that have a hyperrectangle as parameter can be executed very efficiently (e.g., retrieving all tuples of a certain region in space)[13]. How efficient these queries are implemented in BBoxDB is discussed in this section.

When a range query is used to retrieve all tuples of a certain area in space, a hyperrectangle is used to describe the query area. In the first step, the global index is queried with the hyperrectangle to determine the distribution regions that are responsible for the area in space. The global index (implemented by a K-D Tree) can be queried in $\mathcal{O}(\log p + u)$ time. $u$ denotes the amount of the result and $p$ the number of all distribution regions in the global index. The result of the index query are the distribution

[13] The efficiency of operations without a hyperrectangle (such as delete operations) is improved by another index. This is discussed in Section 3.10.

**Listing 2** Accessing BBoxDB from the Java-Client

```
1  String table = "group_table1";
2  BBoxDB bboxdb = [...];
3
4  // Insert a new tuple
5  String key = "key";
6  Hyperrectangle bbox = new Hyperrectangle(0.0, 5.0, 0.0, 1.0);
7  byte[] value = [....];
8  Tuple tuple = new Tuple(key, bbox, value);
9
10 ResultFuture insertResult = bboxdb.put(table, tuple);
11
12 insertResult.waitForCompletion();
13
14 // Retrieve all tuples within the given range
15 Hyperrectangle queryRectangle = Hyperrectangle(-0.5, 1.0, -0.5, 1.0);
16
17 TupleListFuture queryResult = bboxdb.getByRectangle(table, queryRectangle);
18
19 queryResult.waitForCompletion();
20
21 // Output all tuples
22 for(Tuple tuple : queryResult) {
23    System.out.println(tuple);
24 }
```

regions on which the retrieval operation has to be executed. Each region is stored on one or more nodes; therefore, the size of the result $u$ is: $0 \leqslant u \leqslant s$ ($s$ is the number of all nodes of the cluster).

These nodes are contacted and the local index on these nodes is queried to determine the tuples that belong to the query area. Querying the local index (implemented by an R-Tree) and retrieving the tuples has a time complexity of $\mathcal{O}(\log \overline{R} + t)$. The symbol $t$ denotes the amount of tuples in the area and $\overline{R}$ denotes the average number of tuples on each node.

The complete retrieval operation has a complexity of $\mathcal{O}((\log p + u) + (\log \overline{R} + t))$. Because each distribution region stores many tuples ($p \ll R$) and the number of nodes in the cluster is normally less than the number of stored tuples ($u \ll t$), the costs for retrieving the tuples on the nodes are dominant. Therefore the complexity can be reduced to: $\mathcal{O}(\log \overline{R} + t)$.

To summarize, existing systems have an average time complexity of $\mathcal{O}(\overline{R} + t)$ because complete buckets or cells of a grid need to be scanned on the nodes that are responsible for a region. The time complexity for a range query in BBoxDB is as follows: $\mathcal{O}(\log \overline{R} + t)$.

### 3.8 Joins and co-partitioned tables

In comparison to data access from a local hard disk, data transfer through a computer network is considerably slower and has a high latency. This immensely reduces the

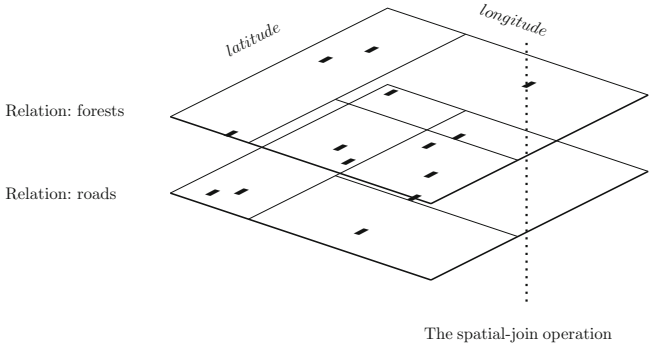



**Fig. 11** Executing a spatial join on co-partitioned two-dimensional data. Both relations are stored in the same distribution group. Therefore, the relations are partitioned and distributed in the same manner

speed of data processing. 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. At the moment a table is created, it must also be determined to which distribution group the table should belong. All tables in a distribution group must have the same dimensions.

On co-partitioned data, distributed spatial joins can be efficiently executed on locally stored data (the seventh challenge of the Introduction). 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 by calling the `join()` operation (see Sect. 3.2).

For example, the two tables *roads* and *forests* store the spatial data of all roads and of all forests of a particular area. Both tables are stored in the same two-dimensional distribution group. The bounding box of each tuple is calculated using the location of the tuple 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 in a co-partitioned manner, all roads and all forests that might intersect are stored on the same nodes (see Fig. 11).

BBoxDB executes the spatial join based on the bounding boxes of the tuples. Depending on the meaning of the stored data, intersecting bounding boxes does not mean that the data really intersects. Therefore, the BBoxDB join result has to be refined by an application that understands the stored data. In the previous example with the two-dimensional spatial data, the application has to check whether or not the reported tuples really intersect. Figure 12 illustrates the situation. In Fig. 12a the two objects have intersecting bounding boxes but do not intersect, in Fig. 12b the objects have intersecting bounding boxes and do intersect.

Decomposing a spatial join into two steps is a common technique. Testing whether two bounding boxes are intersecting is a cheap operation, while calculating if the

**(a)** Spatial data is not intersecting.

**(b)** Spatial data is intersecting.

**Fig. 12** Two two-dimensional spatial objects with intersecting bounding boxes

objects really intersect is an expensive operation. Therefore, the calculation of a spatial join is often divided into a *filter step* and a *refinement step* [51]. The first step detects all possible join candidates, while the second step eliminates the non-intersecting candidates. Because BBoxDB knows only the bounding box of the tuples and does not know how to interpret the value, BBoxDB provides only support for the filter step. The refinement step has to be executed in the application that understands the semantics of the value.

### 3.9 Updates

The ability to process updates efficiently is essential for a data store. An update consists of two operations: (1) a possibly existing old value needs to be invalidated and (2) the new value has to be written. As stated in the introduction, an update operation in a KVS is simply a `put` operation with an already existing key. This works in a regular DKVS as well. The key determines the node that is responsible for the storage of the tuple and the new version is stored on this node. All `put` operations of one key are sent to the same node; this node can invalidate the already stored data. In BBoxDB, the location in space determines the storage place for a tuple. Invalidating the old tuple version becomes more difficult.

For example, we have four tuples $t_1 = (key, bbox1, ...)$, $t_2 = (key, bbox1, ...)$, $t_3 = (key, bbox2, ...)$, $t_4 = (key, bbox3, ...)$. All four tuples have the same key; the tuples $t_1$ and $t_2$ have the same bounding box whereas $t_3$ and $t_4$ have different bounding boxes. The space is partitioned such that $bbox1$ and $bbox2$ are stored on node $n_1$ and $bbox3$ is stored on node $n_2$ (see Fig. 13). To keep the example easy to understand, only point data is used. The update strategies that are presented later in this section do also work on non-point data, even when the data overlaps multiple regions and are stored on multiple nodes (as discussed in Sect. 3.1.3).

Assume that the tuples are stored in the following order: $t_1$, $t_2$, $t_3$, $t_4$. The `put` operations for $t_1$, $t_2$ and $t_3$ are executed on $n_1$. The later `put` operations overwrite the old written value as expected. The `put` operation for $t_4$ is only executed on $n_2$. In this case, only the new value for key is stored on $n_2$ and $t_3$ stays untouched on $n_1$. $t_3$ is not overwritten and still stored on one node and might be part of range queries. This situation could lead to outdated tuples in queries and needs to be corrected. Three strategies exist to solve the problem. The efficiency of the strategies is evaluated in Sect. 4.6.

ON- READ STRATEGY: Outdated tuple versions are detected and removed, when the tuples are read. The client needs to contact all nodes of the cluster and requests the

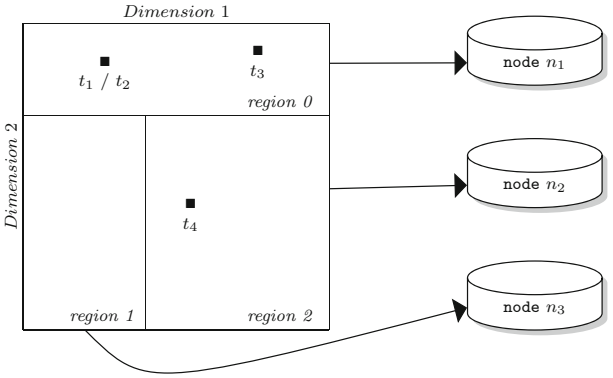



**Fig. 13** Updating the tuple $t$ with the versions $(t_1, t_2, t_3, t_4)$. $t_1, t_2, t_3$ are stored on node $n_1$, $t_4$ is stored on node $n_2$

newest version of a tuple for the particular key. In the final result, only the most recent version of the tuple is included. In our example, $t_3$ is excluded from the result after all nodes are contacted. The on-read strategy can be performed in $\mathcal{O}(s \cdot \log \overline{R})$ time. All $s$ nodes of the cluster need to be contacted and the tuple for the given key needs to be retrieved.

ON- WRITE STRATEGY: This strategy invalidates the old version of a tuple in the moment when a new version is stored. Because the `put` operation knows only the bounding box of the new tuple, the client does not know on which node the old tuple is stored. Therefore, all nodes of the cluster need to be contacted. This can be done in $\mathcal{O}(s)$ time.

INDEX- BASED UPDATE STRATEGY: The problem of the ON- READ STRATEGY and the ON- WRITE STRATEGY is, that the bounding box of the previous stored tuple is unknown. Both strategies contact all nodes of the cluster to compensate this. The performance of both strategies depends on the number of nodes in the cluster and they become slower when the cluster gets larger. In BBoxDB, a further strategy is implemented. An additional index (the *bounding box index*) is created; the maintenance of the index is described in Sect. 3.10. The index maps from a key to the bounding box of the most recent stored tuple version. This solves the problem of the previously described strategies. An index

lookup returns the bounding box of the currently stored tuple. This bounding box is used to contact only the nodes which store the old version of the tuple. Only on these nodes, the old tuple is deleted. This can be done in $\mathcal{O}(t)$ time with $t \leq s$. The number of nodes $t$ depends only on the size of the bounding box of the tuple and the partitioning of the space.

### 3.10 The bounding box index

BBoxDB can maintain an additional index. This index maps the keys of the tuples to the last known bounding box and it is called the *bounding box index*. The index is stored in a 1-dimensional distribution group in BBoxDB. Helper methods are implemented in the Java-Client to maintain the index. Because the index is stored in BBoxDB, it is stored in a scalable and highly available manner.

A big challenge is that multiple clients can execute updates in parallel on the same key of the index. The access to the index entries has to be synchronized to prevent lost updates. To solve this problem, BBoxDB implements locks. A client can lock the corresponding index entry, so other clients have to wait until the holder of the lock unlocks the entry. As soon as an index entry is locked, the client can delete the old stored tuple, write the new tuple and update the index without the risk of a lost update. The `lock()` function of BBoxDB (see Sect. 3.2) is used to lock index entries. The index can be modified in parallel, only the index entries are protected by the lock against concurrent modifications.

Algorithm 2 shows the index-based updates in pseudo code. To simplify the presentation, some special cases (e.g., handling non-existing index entries, the creation of the one-dimensional index relation or handling node failures) are omitted, even if they are handled in the actual implementation. In addition, techniques are implemented to prevent invalid index entries if a client dies in the middle of an update operation.

In line 2–5, the tuple is decomposed into its components. In line 6 the name of the index relation (stored in an one-dimensional distribution group) is determined. In line 7, the bounding box of the index entry is calculated. We use the hash value of the key to determine a point in the one-dimensional space[14]. Beginning with line 8, a *while true loop* is entered. Due to concurrent modifications, it may be necessary to retry the following operations. In line 9, the current index entry is retrieved. The procedure `getIdx` (see Algorithm 3) is used to retrieve the current index entry. In line 11, the algorithm tries to lock the current index entry. If this does not succeed (e.g., another client has modified the index entry between our read and the lock operation), a new round of the while true loop is entered and a new version of the index entry is read. If the index entry could be locked successfully, the new version of the tuple is written (line 14). After that, a deletion marker with a timestamp lower than the new inserted

[14] Other functions for mapping the key to a point in the one-dimensional space can also be used. It is only important that all nodes use the same function for reading and writing index entries.

**Algorithm 2** Index based updates

```
1: procedure INDEXBASEDUPDATE(table, tuple)
2:     key ← tuple.key
3:     bbox ← tuple.bbox
4:     value ← tuple.value
5:     version ← tuple.version
6:     idxTable ← getIndexTablename(table)
7:     idxBbox ← getBBox(key)
8:     while true do
9:         idxEntry ← getIdx(idxTable, key, idxBbox)
10:        idxVersion ← idxEntry.version
11:        lock ← bboxdb.lock(idxTable, key, idxVersion)
12:        if lock == true then
13:            lastTupleBBox ← idxEntry.bbox
14:            bboxdb.put(table, key, bbox, value, version)
15:            bboxdb.delete(table, key, lastTupleBBox, version - 1)
16:            bboxdb.put(idxTable, key, idxBbox, bbox)
17:            return
18:        end if
19:    end while
20: end procedure
```

tuple is written to the necessary nodes (line 15)[15]. The bounding box of the old tuple version is used to identify the nodes that have to delete the old version of the tuple. In line 16 the index entry is updated; the bounding box of the current tuple is stored in the index. This `put` operation also removes the lock on the index entry.

Algorithm 3 describes how the index entry is fetched from BBoxDB. In line 2, the index entry is retrieved with a range query. Key queries in BBoxDB are send to all nodes, whereas range queries need to contact only a few nodes. Therefore, the range query is used to retrieve the index entry. In line 3, all retrieved index entries are iterated. In line 4, it is checked whether the index entry matches the searched key or not. If the key matches, the entry is returned; otherwise the next entry is processed.

Table 5 compares the time complexity of the operations without and with index support. It can be seen that without the index, the operations `delete` and `getByKey` need to contact all $s$ nodes of the cluster. By using the index, the number of nodes can be reduced to $u$ (with $u \leqslant s$). The number of nodes to contact depends on the size of the bounding box of the stored tuple. If the bounding box covers the whole space, all nodes need to be contacted. However, when the bounding box is small, only one or a few nodes need to be contacted to perform the operation. The index does not affect the time complexity of the operations `put` and `getByRect`. These operations require a bounding box as parameter (see Sect. 3.2) to determine which nodes need to be contacted.

[15] As discussed in Sect. 3.1.2, each tuple contains a version timestamp to identify the most recent version of the tuple. To delete a tuple, a deletion marker is stored on disk (see Sect. 3.3.2). Also, deletion markers are stored together with a version timestamp. This behavior is used to store a new version of a tuple at time $t_1$ and to apply a deletion operation later in time which affects only the tuples which have been stored before time $t_1$. This is needed to ensure that a version of the tuple is stored at any time in BBoxDB. Otherwise, the tuple has to be deleted before the new version is stored. Such an implementation would lead to missing tuples in BBoxDB when a read request is executed by a client between the deletion and the put operation.

**Algorithm 3** Get Index Entry

```
1: procedure GETIDX(table, key, idxBbox)
2:     entries ← bboxdb.queryByBB(table, indexBbox)
3:     for entry ∈ entries do entryKey ← entry.key
4:         if key != entryKey then
5:             continue
6:         end if
7:         return entry
8:     end for
9:     return nil
10: end procedure
```

**Table 5** The number of nodes that need to be contacted with and without the bounding box index. $s$ denotes the number of all nodes of the cluster, $u$ the number of nodes whose distribution regions are overlapped by the bounding box

| Operation | Without index | With index |
|---|---|---|
| `put` | $u$ | $u$ |
| `delete` | $s$ | $u$ ($u \leqslant s$) |
| `getByKey` | $s$ | $u$ ($u \leqslant s$) |
| `getByRect` | $u$ | $u$ |

**Table 6** Comparison of a generic key-value-store with our implementation of a key-bounding-box-value store

| Feature | KVS | KBVS (BBoxDB) |
|---|---|---|
| *Data model* | Key-value | Key-bounding-box-value |
| *Optimized for* | One-dimensional point data | $n$-dimensional (non-)point data |
| *Distributes data on* | Key | Bounding Box |
| *Primary retrieval via* | Key | Bounding Box |
| *Key query* | $\mathcal{O}(\log \lvert R \rvert)$ | $\mathcal{O}(\lvert R \rvert)/\mathcal{O}(\log \lvert R \rvert)$ |
| *Multi-Dimensional range query* | $\mathcal{O}(\lvert R \rvert)$ | $\mathcal{O}(\log \overline{R} + t)$ |

The two complexities of a key query in a KBVS depends on whether or not the *bounding box index* is used

Table 6 compares the characteristics of a generic KVS with our implementation.

### 3.11 Executing operations

In the previous sections, we have discussed how BBoxDB works internally. In this section, we describe how a range query (see Sect. 3.2) is performed from a client-side view. In Fig. 14, a two-dimensional space is shown, which is partitioned into three distribution regions. These distribution regions are assigned to the three nodes a, b and c. The query hyperrectangle that is shown in the Figure is used as an example in this section.

When the BBoxDB client is initialized (see Sect. 3.6), it fetches the global index for each existing distribution group and creates an in-memory copy of the index (see Sect. 3.4.2). Besides, the available BBoxDB nodes are read from ZooKeeper and a network connection to all of these instances is opened (see Sect. 3.3.1).

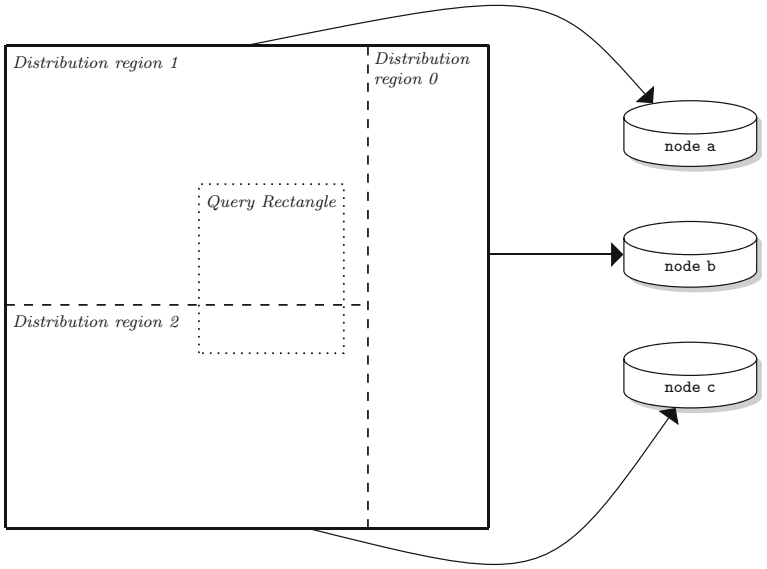



**Fig. 14** Performing a range query in BBoxDB. In the first step, the in ZooKeeper stored global index is read and it is determined which distribution regions and nodes are affected by the range query

To execute the range query, the operation `getByRect(hyperrectangle)` is called on the interface of the BBoxDB client. After the method is called, the client carries out the following steps internally to calculate the query result:

1. The in-memory copy of the global index is read and it is calculated which distribution regions are intersecting with the query's hyperrectangle. In this example, regions 1 and 2 are intersecting.
2. The in-memory copy of the global index also contains the information, which node is responsible for which distribution region. Therefore, the BBoxDB client can determine that node a and node c have to be contacted to execute the query.
3. The open network connections are used to perform the query on these nodes.
4. The nodes are query the local spatial index (see Sect. 3.3.2) with the hyperrectangle from the query and determine the tuples that have bounding boxes that are intersecting with the query's hyperrectangle.
5. The needed tuples are read from the SSTables and send back to the client.
6. The client reads the results from the server and eliminates duplicates (tuples that are part of both distribution regions and stored on both nodes; see Sect. 3.1.3 for more details) before the results are returned.

To keep the example focused, topics such as a changing global index during the query, outdated in-memory copies of the global index are omitted from the description. Details about that can be found in the `BBoxDBCluster` class of the BBoxDB implementation.

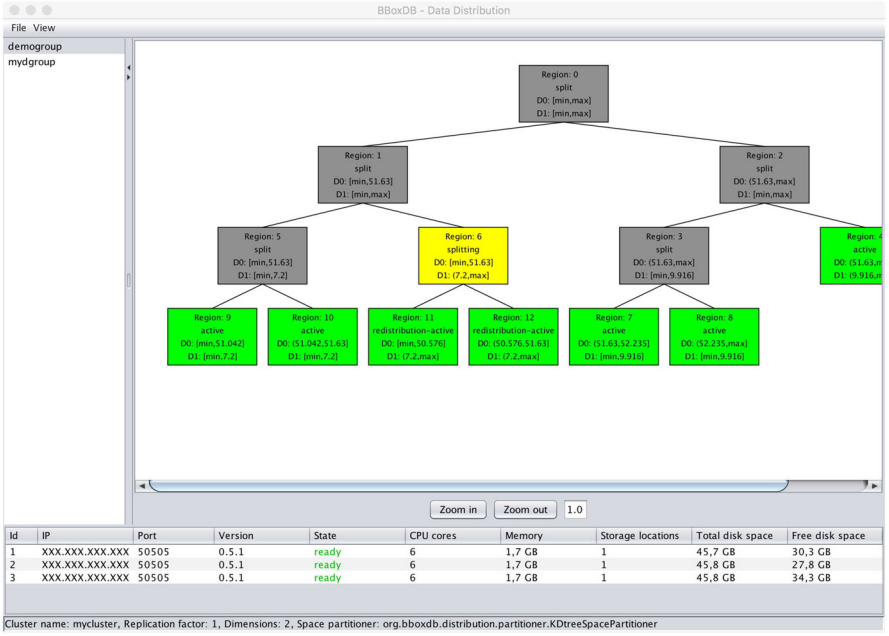



**Fig. 15** The GUI of BBoxDB shows the global index

### 3.12 The GUI

BBoxDB ships with a GUI that visualizes the global index and shows all discovered BBoxDB instances. The installed version of BBoxDB and the state of each instance are also shown. Figure 15 shows the global index and the state of the distribution regions. In the left 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 shown in Fig. 16.

## 4 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.

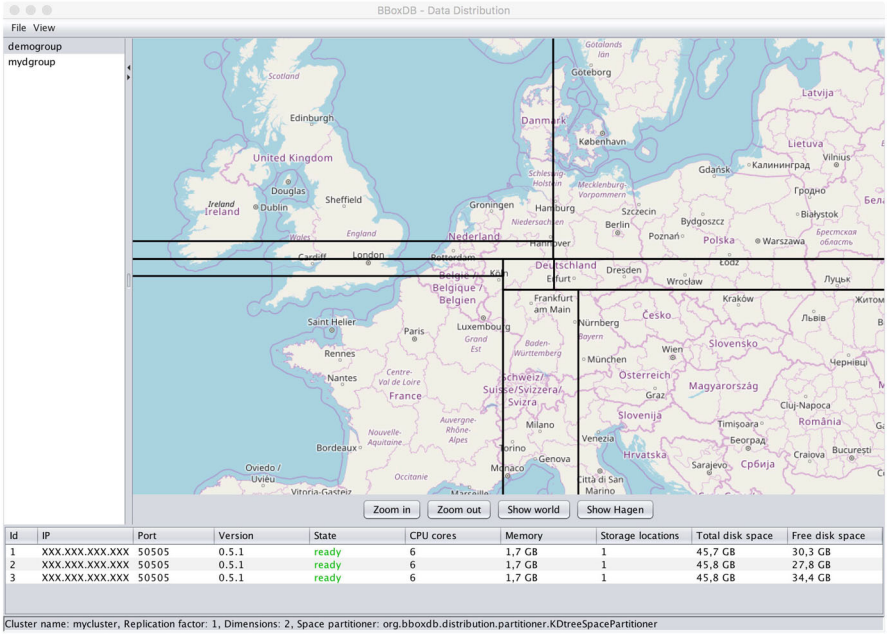



**Fig. 16** The distribution tree as an OpenStreetMap overlay

**Table 7** A summary of the datasets used in the experiments

| Dataset | Data | Dimensions | Type | Elements | Size |
|---|---|---|---|---|---|
| TPC-H | Line item | 1 | Point | 119 994 608 | 15 GB |
| OSM | Tree | 2 | Point | 10 625 658 | 1.7 GB |
| OSM | Forest | 2 | Non-Point | 4 409 678 | 4.7 GB |
| OSM | Road | 2 | Non-Point | 129 696 016 | 59 GB |
| Rome | Taxi | 3 | Point | 21 817 851 | 1.5 GB |
| Rome | Taxi | 3 | Non-Point | 10 335 | 1.5 GB |

### 4.1 Used datasets

For the evaluation, one synthetic and two real datasets with one to three dimensions are used. The characteristics of the datasets are shown in Table 7.

**TPC-H dataset** (one-dimensional): This dataset is generated by the data generator of the TPC-H benchmark [21]. The benchmark data is generated with a scale factor of 20. The table *lineitem* has a size of 15 GB and contains 119 million line items. The attribute *ship date* is converted into a timestamp to generate a point in the one-dimensional space.

**OSM dataset** (two-dimensional): Data from the *OpenStreetMap Project* [50] are used for this dataset; the spatial data of the whole planet is used. The dataset has a size of around 41 GB in the compact and compressed binary encoded *protocol buffer*

*binary format*. GeoJSON excerpts of the dataset, such as trees (point data) or all roads (non-point data), are used in the experiments. The dataset includes 10 million trees (1.7 GB), 4 million forests (4.7 GB) and 52 million roads (59 GB).

**Rome Taxi Dataset** (three dimensional): This dataset contains mobility traces of taxi cabs in Rome, Italy [17]. The dataset consists of the time and the position of approximately 320 taxis collected over 30 days. It has 21 million data points in the three-dimensional space (longitude, latitude, time) and a size of 1.5 GB. The points are used to reconstruct the trajectories of the taxis and to create a non-point data type in the 3-dimensional space (an `mpoint` [32, p. 27]). 10335 trajectories are reconstructed with a maximum length of 8 hours (the daily working time of a taxi driver).

### 4.2 SSTables with a spatial index

As described in Sect. 3.3.2, data is stored in SSTables by BBoxDB. In contrast to regular SSTable implementations, these SSTables are enhanced by a spatial index to speed up spatial queries. As described in Sect. 3.4, this index is called the local index. In the current version of BBoxDB, an R-Tree is used as the local index for the bounding boxes of the stored tuples.

The local index speeds up the spatial queries and prevents full data scans. But as every index, the index needs some resources for creation and maintenance. This experiment examines how much maintenance costs the index needs and how much it speeds up the spatial queries. For the experiment, the datasets are loaded into one BBoxDB instance. The import is performed one time without creating the local index and one time with the creation of the index. The execution time of both data import processes is measured and shown in Fig. 17. After the data are loaded, 100 spatial queries are performed on the data and the average execution time with and without the local index is measured. The query is performed by the operation `getByRect(hyperrectangle)` (see Table 2). To determine the ranges that are used in the range query, 100 samples are chosen randomly from the imported datasets, and their bounding boxes are used in the query. The result of this experiment is shown in Fig. 18.

As in every DBMS, building an index slows down write accesses and speeds up read accesses. This behavior is also confirmed in this experiment. Figure 17 shows that the creation of the R-Tree increases the time for the data import by about 30%. However, as shown in Fig. 18, spatial queries can be accelerated by a factor of about 100 and this type of query is the primary tuple retrieval method used in BBoxDB. Therefore, the index speeds up the most important type of query.

### 4.3 Distributing data

The efficiency of the data partitioning is evaluated in this experiment. BBoxDB splits up stored data into almost equal-sized *distribution regions* using a space partitioner. We compare the dynamic K-D Tree based partitioning used by BBoxDB with a static approach by using a grid (like done in *SJMR* [60] or Distributed Secondo).

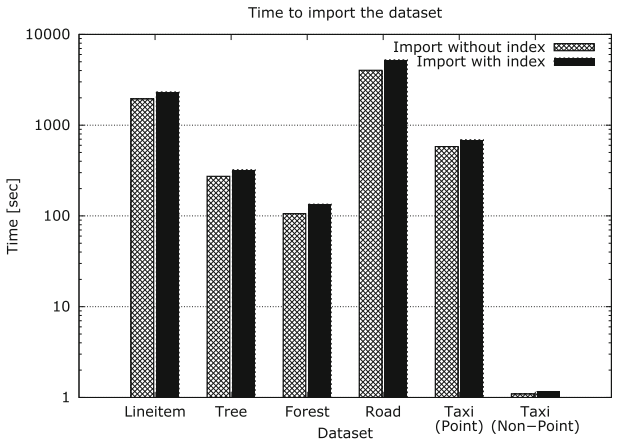



**Fig. 17** Importing the datasets into BBoxDB with and without building the local index

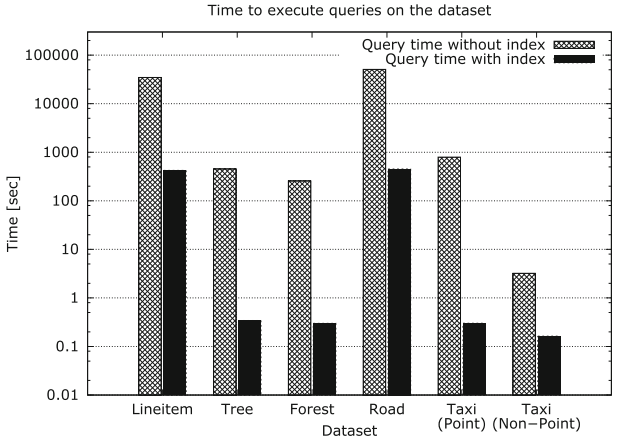



**Fig. 18** The average execution time of a range query, with and without using the local index

The static approach uses an $n$-dimensional grid to partition the space into cells. These cells are mapped to the nodes of the cluster. The data are placed on the grid and stored on the node that is responsible for the cell. Non-point data that belong to multiple cells are duplicated and stored multiple times. To get a more balanced data distribution, more cells than nodes are created and multiple cells are mapped to one node or partition [53, p.5].

We evaluate the efficiency of the data distribution by calculating the *standard deviation* ($\sigma$) of the stored data. The standard deviation describes the deviation in the amount of data stored between the nodes. A lower standard deviation indicates that the nodes store an almost-equal amount of data. A higher standard deviation indicates that there are differences in the amount of stored data. For the experiment, we created several K-D Trees for each dataset with different configurations. We varied the maximum number of stored tuples per node and measured the standard deviation. The grid

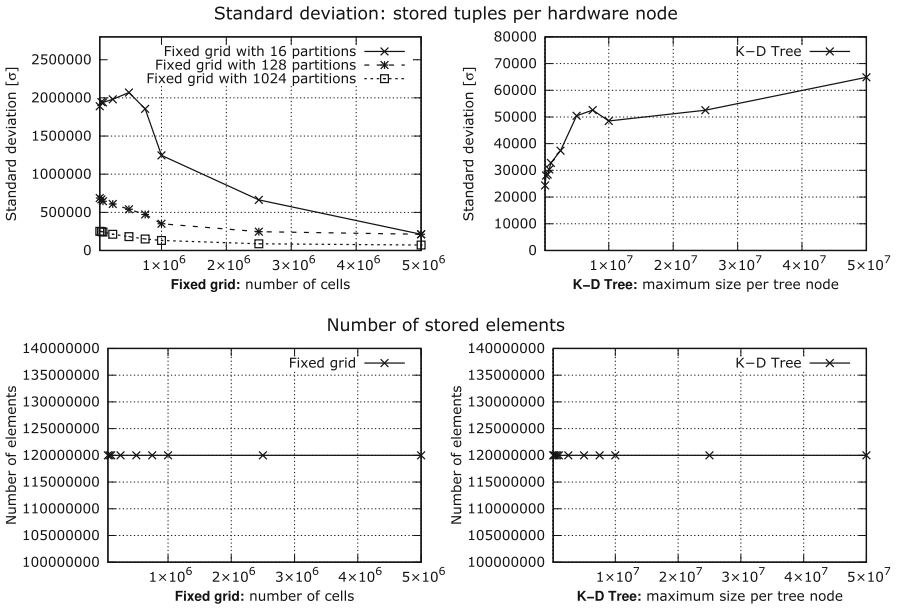



**Fig. 19** The distribution of the TPC-H dataset

based approach has two parameters that determine the distribution of the data: (1) the cell size and (2) the number of partitions to which the cells are mapped. To cover both parameters, we created a grid with different cell sizes and mapped these cells to 16, 128 and 1024 partitions[16].

The experiment was performed on all datasets. In addition to the individual datasets, for the two-dimensional and three-dimensional datasets, all data (point and non-point) were loaded and distributed at once. This shows the behavior of BBoxDB when different objects are stored at the same time in one distribution group and it is called the *mixed dataset*.

The result of the experiments with the different datasets is very similar. To increase the readability of the paper, only three datasets are discussed in greater detail: the TPC-H point data dataset (Fig. 19), the OSM non-point dataset (Fig. 20), and taxi mixed dataset (Fig. 21).

It can be seen in the figures that the dynamic approach by the K-D Tree creates a more equal distribution for most datasets in most configurations; the standard deviation is lower. In general, using a static grid with smaller cells leads to a lower standard deviation, because the data are distributed on more cells. Using a lower limit for the node size in a K-D Tree also leads to a lower standard deviation. Due to the lower limit, the K-D Tree is split more often which generates a more equal data distribution. The figures show also, that for non-point data the K-D Tree has to store fewer tuples than the grid-based approach. The space is divided into fewer partitions and therefore

[16] Systems that use a static grid to work with multi-dimensional data like DISTRIBUTED SECONDO are using more partitions than nodes to allow a dynamic scaling of the cluster. 128 partitions is a common size in a cluster of ten nodes (see [46] for more details).

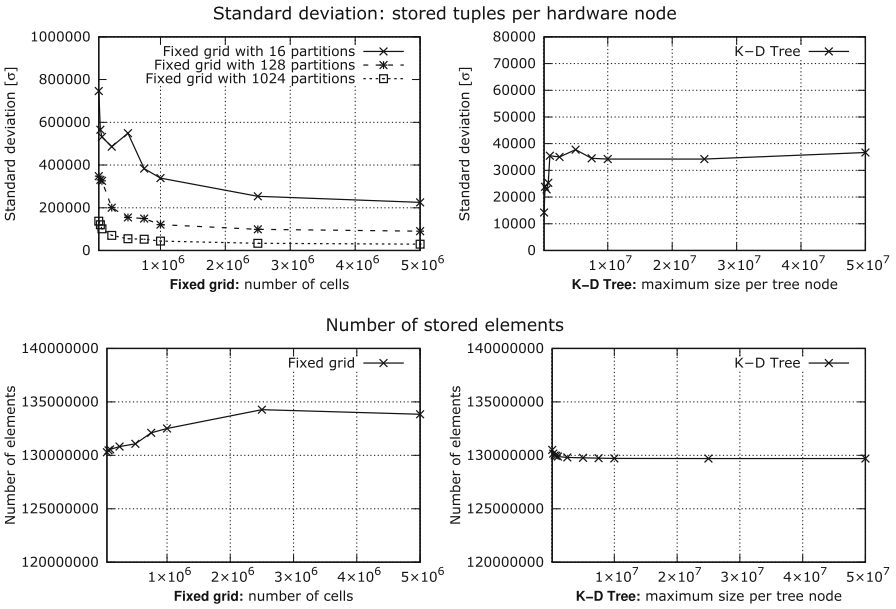



**Fig. 20** The distribution of the OSM road dataset

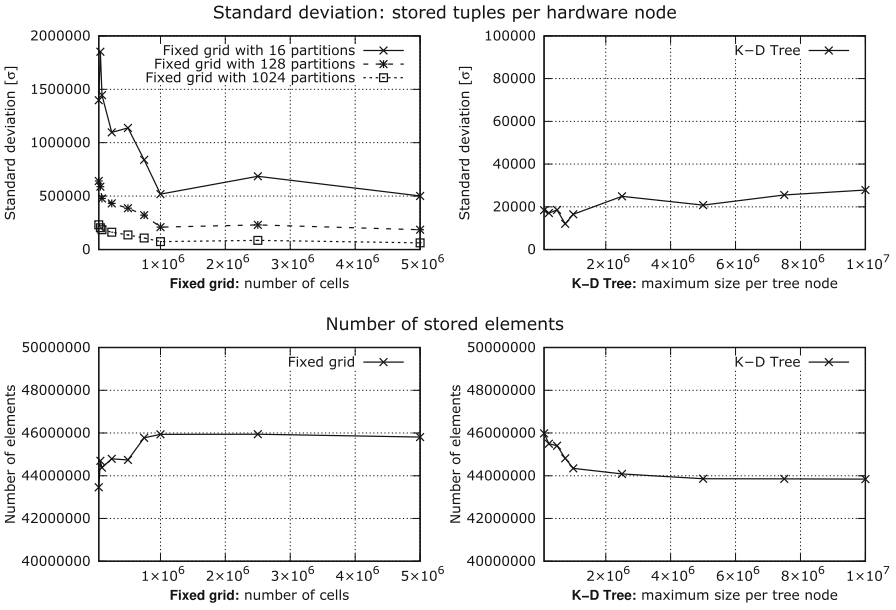



**Fig. 21** The distribution of the taxi mixed dataset

less non-point tuples belong to two or more regions. Such tuples need to be duplicated and stored more than one time. For point data, the amount of stored tuples is identical for the grid-based approach and the K-D Tree in all configurations. Each tuple belongs to precisely one partition, and therefore, no tuple needs to be duplicated.

The standard deviation by using the static grid depends highly on the used parameters. In general, densely and sparsely populated regions are harder to handle. To create a better tuple distribution, the number of cells needs to be optimized for each dataset[17]. Every time the grid is changed, the already stored data has to be be redistributed, which is an expensive operation. The dynamic approach does not require this type of tuning.

### 4.4 Retrieving tuples

This section describes an experiment that evaluates how fast multi-dimensional data can be retrieved in several systems. The approaches of the systems and the execution times are compared with BBoxDB. In the experiment, it was measured how many time it takes to answer a range query on the datasets. The execution time of BBoxDB is compared to other systems like DISTRIBUTED SECONDO, Tiny MD-HBase[18], Spatial Hadoop[19] and a simple baseline approach using Cassandra. In the baseline approach, the datasets are stored in Cassandra and a full data scan is used to retrieve the data[20].

To evaluate the performance of the tuple retrieval operation, the datasets are imported and 100 range queries per dataset are executed. The average execution time of the query is measured. To determine the argument of the range query, 100 random samples per dataset are loaded and the bounding boxes of these samples are used. In addition, for the non-point datasets, the bounding boxes of the samples are enlarged by a factor of 5 in each dimension. Due to the larger bounding boxes, a larger part of the space is intersected by the range query. The name of each dataset is suffixed with the identifier *x5* in the experiments result. The result of the experiment can be seen in Fig. 22.

BBoxDB and DISTRIBUTED SECONDO are the only systems which can handle all datasets. In addition, the performance of Spatial Hadoop and DISTRIBUTED SECONDO are quite similar; BBoxDB instead is a bit faster. BBoxDB, Spatial Hadoop and Tiny MD-HBase are using a K-D Tree to index the stored data. DISTRIBUTED SECONDO instead uses a static grid as an index. The range query in the baseline approach needs a lot more time, because no spatial index does exist and a full data scan is required. The enlarged range queries (increased bounding boxes by a factor of 5) need a bit more time for the execution in all systems due to the larger part of the space that is covered by the query.

[17] A further problem with the grid approach is that the creation of the grid depends on the used dimension. For example, the grid operator in SECONDO needs a concrete implementation for each dimension. At the moment, SECONDO contains implementations for two- and three-dimensional grids. Using a non supported dimensionality leads to additional work for creating an appropriate grid operator. The K-D Tree in BBoxDB can be used immediately for any dimension without any adjustment being necessary.

[18] Tiny MD-HBase is limited to two-dimensional point data, therefore we compare tiny MD-HBase only with this dataset. To import the OSM point dataset, we modified Tiny MD-HBase and added an import function for GeoJSON data. In addition, we added some statistics about the scanned data. Our version of Tiny MD-HBase is available on GitHub [56].

[19] Spatial Hadoop can only handle two dimensional data, therefore range queries were performed only on the two dimensional datasets.

[20] The source code of the baseline approach can be found in the BBoxDB repository at GitHub [13].

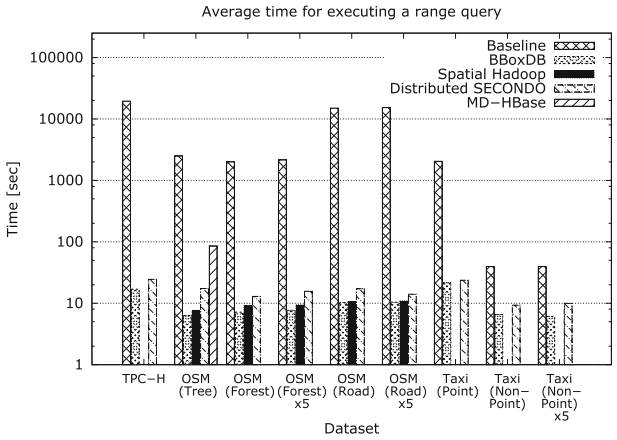



**Fig. 22** The average time to execute a range query on the datasets using different systems

BBoxDB computes the range query based on bounding boxes. The other systems calculate if the range query really intersects with the data in the result set (see Fig. 12). Determining whether two bounding boxes are intersecting is cheap to compute. To compute whether two objects really intersect is much more expensive. To avoid the expensive computation, some algorithms (e.g., spatial joins) use a *filter step* and a *refinement step* (see Sect. 3.8). The first step calculates if the bounding boxes of the elements intersect. When the bounding boxes do not intersect, the elements can not intersect and the second step is omitted. The second step performs the expensive calculation and determine whether or not the elements really intersect.

In a range query, the refinement step is only needed if an element is not completely covered by the query range. In this case, only the bounding box could be intersected but not the element itself (see Fig. 12 for an example). If a bounding box is completely covered by the query range, the element is clearly part of the query result and no refinement is needed.

Considering only the fact that BBoxDB operates on the bounding boxes, the system should be much faster than the other systems. However, there are many differences in the architecture of the systems which determine the execution time of the queries. For example, other systems write the result back to a distributed system (Cassandra for DISTRIBUTED SECONDO and HDFS for Spatial Hadoop). BBoxDB, on the other hand, sends the result back to the requesting client via the network, which takes additional time.

### 4.5 Computing joins

The experiment of this section evaluates the execution time of spatial joins in BBoxDB. The experiment also compares the execution time with other systems. In the experiment, two different spatial joins are executed and the execution time is measured. The first spatial join works on the two-dimensional open street map datasets *Road* and *Forest*. The spatial join reports all roads that lead through a forest; the roads and the

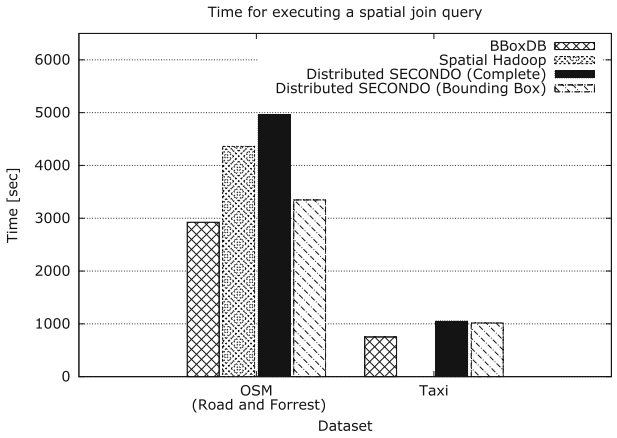



**Fig. 23** The needed time to execute a spatial join query in different systems

forests that are reported by the spatial join have a common point in space. The second spatial join is executed on the three-dimensional taxi non-point data. The spatial join is used to find all vehicles that are approaching closer than 10 meters at any given time. Only the approach and not the overlapping of two vehicles is checked in the join condition. If two vehicles have a common point in space and time then they have collided (which hopefully does not happen).

For better comparison, we implemented a baseline approach which performs the same join by using a KVS and full data scans[21]. The baseline approach executes a nested loop join on in Cassandra stored data and runs for more than four weeks without producing a result. Section 4.4 describes that a full data scan of the forest OSM relation need around 2000 seconds ($\approx$ 33 minutes). The nested loop join of the baseline approach compares each tuple of the first input relation with the second input relation. By using the table road as the first relation and forest as the second input relation, each of the 129 696 016 roads performs a full data scan on the forest relation. This calculation runs for more than 8000 years. This same execution time is also achieved by swapping the relations.

MD-HBase is not capable of performing joins; therefore the system can not be used in this experiment. Spatial Hadoop can handle only the 2-dimensional dataset while DISTRIBUTED SECONDO and BBoxDB can handle all datasets. The result of the experiment is shown in Fig. 23.

As in the range query experiment (see Sect. 4.4), a direct comparison of the execution time of the systems is hard due to their different architecture and the way the join is computed.

The join that is performed by BBoxDB is computed based on the bounding boxes. The join performed by the remaining systems is executed on the real geometries of the stored data. Performing the join based on the bounding boxes is easy to compute, to perform the join on the real geometries is a more expensive operation. BBoxDB is a generic datastore and knows only the bounding boxes of the stored data. The real

[21] The source code of the baseline approach can be found in the BBoxDB repository at GitHub [13].

geometry that is encoded in the value of the tuples can not be decoded and handled by BBoxDB. Therefore, BBoxDB operates on the bounding boxes of the tuples and performs the *filter step* of the spatial join (see Sect. 3.8).

To make the systems more comparable, we used the fact, that DISTRIBUTED SECONDO allows the direct specification of the executed query plans. We performed the experiment with DISTRIBUTED SECONDO two times: (1) one time the spatial joins are executed on the bounding boxes only and (2) one time the spatial joins are executed with the refinement step. In the figure, the experiment without the refinement step is labeled with DISTRIBUTED SECONDO *(Bounding Box)*, the experiment with the refinement step is labeled with DISTRIBUTED SECONDO *(Complete)*[22].

The experiment shows that BBoxDB is capable of performing very fast bounding box based spatial joins. The joins performed by DISTRIBUTED SECONDO (Complete) and Spatial Hadoop are the slowest joins; they are performed on the real geometries. The join performed by DISTRIBUTED SECONDO (Bounding Box) is faster because only bounding boxes are considered in the calculation of the join. The execution time of DISTRIBUTED SECONDO (Bounding Box) is a bit slower than the join in BBoxDB. As discussed in the previous section, the architecture of all systems is different. Distributed SECONDO uses Cassandra as data storage and SECONDO as a query processing engine. Therefore tuples need to be imported from Cassandra into SECONDO when they are processed and exported from SECONDO into Cassandra when they are stored. The import and export step needs some additional time, which makes DISTRIBUTED SECONDO a bit slower than BBoxDB, which can directly operate on the bounding boxes of the data.

### 4.6 Performing tuple updates

In Sect. 3.9 three strategies are described to perform tuple updates. The efficiency of these strategies is evaluated in this section. The experiments in this section are performed on the OSM forest dataset.

#### 4.6.1 Updating tuples with a varying amount of clients

This experiment examines the performance of these strategies when 1000 put operations are executed. In a first set the OSM forest dataset is imported, then the first 1000 tuples of this dataset are updated using the different update strategies. The results of the experiment are shown in Fig. 24.

It can be seen that the ON-WRITE and the ON-READ strategies have an almost similar execution time. The INDEX-BASED UPDATE strategy is much slower. The reason for the behavior are the high coordination costs for updating the bounding box index. Updates on the index need to be coordinated to prevent race conditions.

[22] We also tried to execute the spatial join with Spatial Hadoop only on bounding boxes. We created new datasets witch contained the bounding boxes of the OSM datasets. Afterwards, the spatial join was executed on these datasets. However, the execution time of the join increased by a factor of 5. We assume this is an error in the current implementation and therefore, we don't show the execution time in the figure of the experiment.

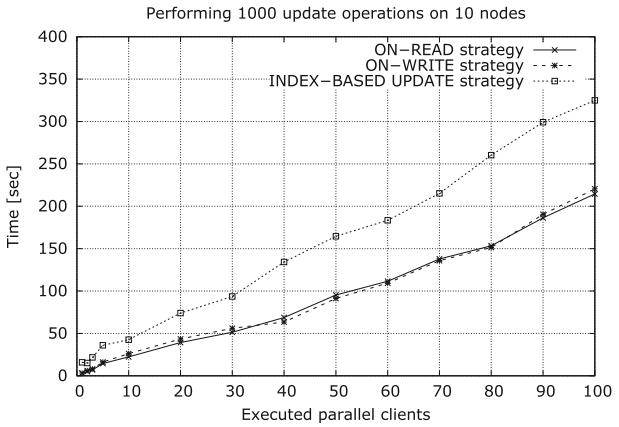



**Fig. 24** Performing updates with a varying amount of clients. Each client performs 1000 update operations

Each index entry needs to be locked before the entry can be safely updated (see Sect. 3.10). In the first step, the old index entry is read by the `get` operation. Afterwards, a `lock` operation for this entry is sent to the server. After executing both operations, the client has to wait for the operation result before the next operation can be performed. In contrast, regular `put` operations (without locking an entry) can be performed asynchronously. The client sends the operation to the server and executes immediately further operations. After a certain time, the server sends the response for this operation back to the client. The asynchronous execution enables the client to execute more operations during the same time.

### 4.6.2 Updating tuples with a varying amount of nodes

In this section a further experiment is described. This time the number of nodes in the cluster is varied and the updates are performed again. To evaluate the behavior, we executed up to 10 BBoxDB installations per physical hardware node of the cluster. Figure 25 shows how the different update strategies perform on a different amount of BBoxDB nodes.

The figure shows that the INDEX- BASED UPDATE strategy has an almost constant execution time; the amount of nodes does not influence the performance of the update operations. The index determines on which nodes the old tuple needs to be invalidated; this is independent of the size of the cluster. On the other hand, the other two strategies operate on all nodes in the cluster. The execution times increase accordingly with an enlargement of the cluster. In our experiment, it can be seen that the increased effort for the INDEX- BASED UPDATE strategy (caused by the maintenance of the index) only leads to an advantage in the execution time of the update operations for a cluster with about 80 nodes and 100 parallel clients. In smaller environments or less parallel operating clients, performing a delete operation on all nodes is faster.

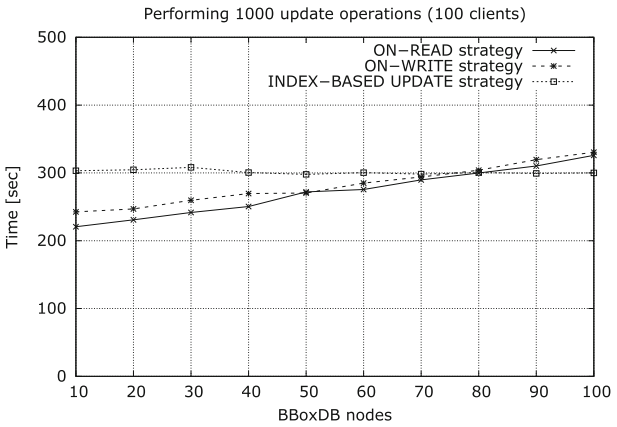



**Fig. 25** Performing the update strategies on a varying amount of BBoxDB nodes. Each client performs 1000 update operations

### 4.6.3 Reading updated tuples

The experiments of the previous sections have shown, that the On- Write and the On- Read strategies have an almost similar behavior regarding the execution time. This does not mean that both strategies behave equally well. The On- Write strategy has already performed the complete update; the new version of the tuple is written and the old version of the tuple is deleted on all nodes. By using the On- Read strategy, in contrast, every read operation has to contact all nodes and requests the newest tuple version for a given key to eliminate the outdated data (see Sect. 3.9). So the costs of the updates are moved into the read operations.

The experiment in this section examines the cost for performing a range query operation on updated tuples. The first 1000 entries of the OSM forest dataset are read and a range query for the corresponding bounding box of each element is performed. The average execution time of these queries is shown in Fig. 26.

It can be seen, that the costs for performing a range query by using the On- Write strategy and the Index- Based update strategy are almost equal. By using these strategies, the tuple updates are performed completely and no additional costs for detecting the outdated tuples are part of the range query operation. By using the On- Read strategy, in contrast, the outdated tuples need to be eliminated during the range query operation. This is performed by executing an expensive `get` operation. BBoxDB uses SSTables as a storage engine; this is a write-optimized data structure. Write operations (e.g., `put` or `delete`) are cheap and performed by adding new data to an in-memory data structure in the first step (see Sect. 3.3.2). Reading a tuple is a more complex operation; the tuple needs to be located and read from disk.

### 4.6.4 Comparison of the update strategy experiments

In summary, we suggest using the On- Write strategy in a smaller cluster to perform tuple updates. Even when the strategy performs a delete operation on all nodes,

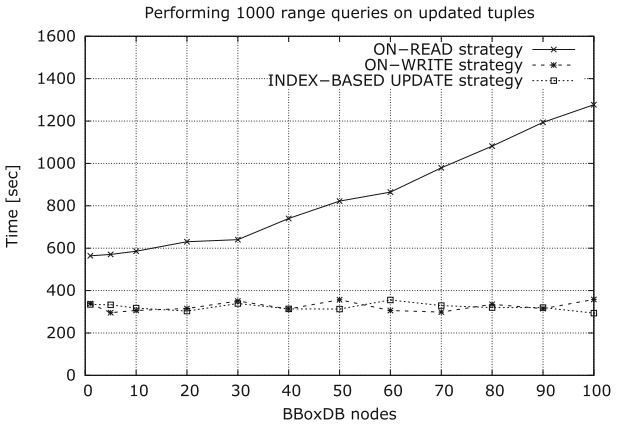



**Fig. 26** Performing tuple reads by using different update strategies and with a varying amount of clients

these delete operations are so cheap that the coordination costs for the INDEX- BASED UPDATE strategy or ON- READ strategy are higher. In a larger cluster, we suggest using the INDEX- BASED UPDATE strategy. This strategy performs updates only on the needed nodes, but the strategy has high coordination costs. The costs for maintaining the index are only paying for themselves in a larger cluster. The ON- READ strategy shows a similar behavior in the experiments like the ON- WRITE strategy. However, the real update costs are moved into an expensive read operation. In all experiments, this strategy shows the worst performance. Therefore, we do not recommend the usage of the ON- READ strategy.

### 4.7 Performing operations while data are redistributed

When a distribution region is split or merged, new distribution regions are created or old ones are deleted. In both cases, the global index needs to be changed. As discussed in Sect. 3.4.2, read and write operations need to be redirected and the in-memory copy of the global index becomes outdated for a short period of time. A client who works with this outdated version might send operations to the wrong nodes and these nodes might reject this operation. The affected operations are retried after a short period of time. This leads to higher execution times of the operations. This experiment examines, how the execution time of range queries (performed by the `getByRect` operation) and `put` operations are affected by a change of the global index. As parameter for the range query, a sample tuple was randomly read from the imported dataset and its bounding is used as parameter for the range query.

Updates of the global index are handled independently from the stored data. Therefore, this experiment is only performed with one dataset. During the import of the OSM road dataset, the execution time of the `put` operations has been recorded during the first distribution region split. In addition, range queries were carried out continuously and their execution time was also recorded during the first distribution region split. Figure 27 shows the execution time of the `put` operations at the top, the required

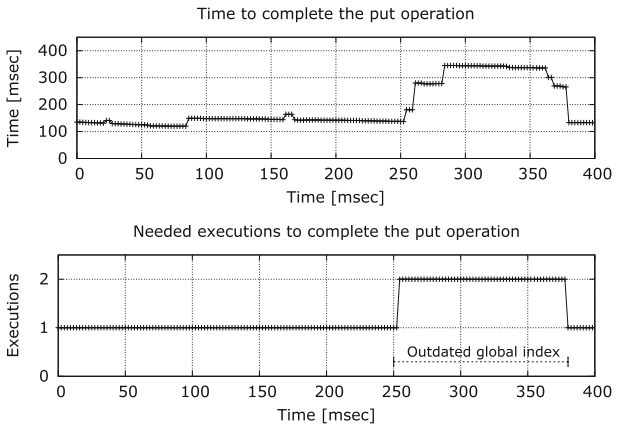



**Fig. 27** Performing put operations while the data are redistributed

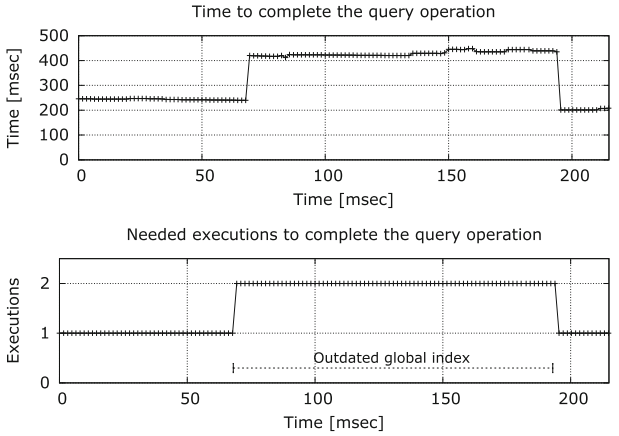



**Fig. 28** Performing range queries while the data are redistributed

number of executions to finish the operation successfully is shown at the bottom of the figure.

In Fig. 28 the execution time and the needed executions are shown when range queries are executed during a split in the global index.

It can be seen in both figures, that for a short period of time two executions are needed to complete the operation successfully. The second execution is needed because the local in-memory copy of the global index is outdated on the client and the server refuses to execute the desired operations on the desired distribution regions. During this time, the total execution time of the operation also increases, because the operation needs to be executed a second time.

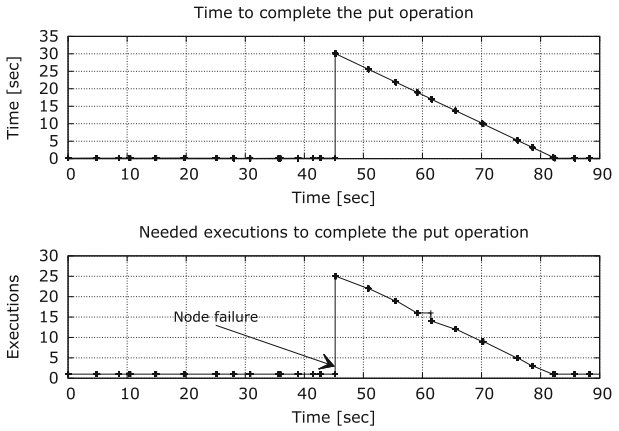



**Fig. 29** Performing put operations during a node failure

### 4.8 Handling node failures

BBoxDB is a highly available system and no single point of failure does exist in its architecture. When data are stored replicated and a node becomes unavailable, operations are executed automatically only on the available replicates. This experiment examines, how node failures are handled and how the execution time of `put` or query operations (performed by the `getByRect` operation) is affected[23].

Just like the experiment in the previous section, the OSM road dataset is used for the experiment and the bounding box of one randomly chosen tuple is used as parameter in the range query. This experiment also consists of two parts. During the first part of this experiment, data are written into BBoxDB. After a particular time, one node is stopped and the execution time and the needed executions are shown. In the second part of the experiment, data are read from BBoxDB. Again, after a certain time, one node is stopped and the execution time and the needed retries are observed. In both parts of the experiment, a replication factor of 2 is used. The results of the experiment are shown in the Figs. 29 and 30. Figure 29 shows the experiment for the `put` operations and Fig. 30 shows the experiment for the query operations.

Figure 29 shows that the execution time and the number of needed retries are increasing after one node has failed. BBoxDB tries to write the data on both replicates. It takes some time for BBoxDB to notice that a node has failed. All BBoxDB nodes create an ephemeral node in ZooKeeper on initialization that automatically gets deleted as soon as the connection to ZooKeeper failed (see Sect. 3.3.1). In our configuration, it takes up to 30 s until ZooKeeper recognizes a connection as dead. During this time period, BBoxDB retries the operation to reach all in ZooKeeper registered BBoxDB

[23] This paper covers only failures in components of BBoxDB. BBoxDB uses ZooKeeper as a coordinator. ZooKeeper itself is also a highly available distributed system, multiple nodes run ZooKeeper in parallel and select one leader. The leader is responsible for performing changes on the stored data. A leader failure results in a new leader election, during this time ZooKeeper clients have to wait before their requests are processed. A discussion of handling failures in ZooKeeper is already contained in papers like [38, p. 11] and not part of this paper.

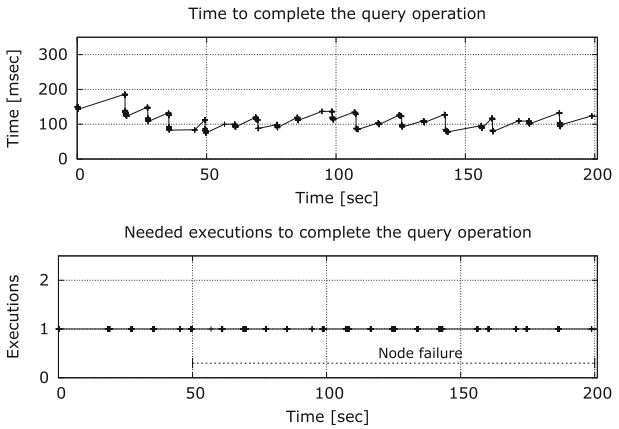



**Fig. 30** Performing range queries during a node failure

nodes, that are responsible for the distribution region. After 30 s, ZooKeeper removes the ephemeral node for the failed node and BBoxDB stops to write data on this node. From this point in time, the put operation succeeded when it is performed successfully the one available replicate. All pending put operations can now be successfully completed in their next execution.

In Fig. 30 almost the same situation is shown. Range queries are executed on a dataset that is replicated two times. One of the nodes fails after some time. However, this time, the node failure does not affect the execution of the queries. The query requests data from all available replicates. As long as a replica is available, the query can be processed successfully.

## 5 Related work

This section presents similar works; these works are explained below in greater detail. Table 8 compares the features of the related work with BBoxDB.

**Distributed systems for BigData.** The *Google File System* (GFS) [30] and its open source counterpart *Hadoop File System* (HDFS) are distributed file systems. HDFS is used by a wide range of applications like *Hadoop* [3] or HBase. Data are split up into *chunks* and stored on *chunk servers*. Chunks can be replicated onto multiple chunk servers. GFS and HDFS provide an append-only filesystem; written data cannot be changed without rewriting the complete file. Additional data structures like SSTables are needed to simulate a read/write capable file system. Systems which are using HDFS directly (e.g., Spatial Hadoop) are limited to the append-only features of HDFS. HDFS and GFS distribute data based on the generated chunks. BBoxDB instead uses the location of the data in an $n$-dimensional space for the placement.

*Apache Cassandra* [42] 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 that this hash function can distribute the data uniformly

**Table 8** Features of the comparable systems

| System | MD data | Non-point data | Updates | Dynamic shards | Co-partitioned data |
|---|---|---|---|---|---|
| *GFS/HDFS* | No | No | Append-only | Yes | No |
| *HBASE* | No | No | Yes | Yes | No |
| *Cassandra* | No | No | Yes | No | Yes |
| *HyperDex* | Yes | No | Yes | Yes | No |
| *ElasticSearch* | 2d | Yes | Yes | Limited | No |
| *GeoCouch* | 2d | No | Yes | Limited | No |
| *MD-HBase* | Any | No | Yes | Yes | No |
| *EDMI* | Any | No | Yes | Yes | No |
| *Pyro* | 2d/3d | No | Yes | Yes | No |
| *HGrid* | 2d | No | Yes | Yes | No |
| *Accumulo* | 3d | Yes | Yes | Yes | No |
| *GeoMesa* | 2d/3d | Yes | Yes | Yes | Limited |
| *GeoWave* | 2d/3d | Yes | Yes | Yes | Limited |
| *Distributed SECONDO* | Any | Yes | Yes | No | Yes |
| *Spatial Hadoop* | 2d | Yes | No | No | Yes |
| *BBoxDB* | Any | Yes | Yes | Yes | Yes |

across the logical ring. If that is not possible, the logical ring becomes unbalanced. Cassandra does not rebalance data automatically; unbalanced data have to be rebalanced manually. BBoxDB redistributes the data dynamically.

*Apache HBase* [35] is a KVS built on top of HDFS; it is the open source counterpart of BigTable [20]. 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. Additionally, BBoxDB supports distribution groups which ensures, that a set of tables is distributed in the same manner (co-partitioned). In HBase the tables are distributed individually.

**NoSQL Databases with Support for Spatial Data.** *MongoDB* [44] is a document-oriented NoSQL database. Documents are represented in JSON. The database also supports geodata, which are encoded as GeoJSON. Queries can be used to locate documents, which are 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* [22] 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* [28]. Querying data that contains or intersects a given region is supported. In contrast to BBoxDB, only two-dimensional data are supported.

*GeoCouch* [27] is a spatial extension for the NoSQL database *Apache CouchDB* [2]. The software 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 have to be created. When the database grows, these shards can be stored on different servers. The database cannot 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* [48] 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 an 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 co-partitioning, 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* [57]—a basic version that illustrates the fundamental concepts of the MD-HBase paper. However, this version is very simple and can only handle small datasets.

*EDMI - Efficient Distributed Multi-dimensional Index* [61] uses a K-D Tree and a Z-order prefix R-Tree to store multi-dimensional point data efficiently in HBase. Like MD-HBase, the software builds an indexing layer on top of HBase. EDMI provides a better query performance than MD-HBase but also supports only point data. In addition, joins, any-dimensional data or co-partitioned data tables are not supported by the software.

*Pyro* [43] is a modification of HBase to store two- and three-dimensional spatial and spatio-temporal data in HBase. The software uses a space-filling curve to encode multi-dimensional point data in a one-dimensional key which is stored in HBase. In contrast, do BBoxDB only two- and three-dimensional point data are supported. Also queries like joins are not covered by the software.

Also *HGrid* [34] is an enhancement of HBase to store 2-dimensional spatial data. Distribution techniques like Quad-Trees combined with a space-filling curve or a grid are used, to map the 2-dimensional data into a data structure that is supported by HBase. However, only 2-dimensional non-point data are supported. Also, this system does not support join queries.

*HyperDex* [24] is a DKVS which supports multi-dimensional point data. Multiple attributes of a tuple are hashed into an $n$-dimensional space. The space is partitioned and mapped to nodes of a cluster. The multi-dimensional hashing solves the problem that the primary access path to the data is only one attribute. In contrast to BBoxDB only point data are supported and the data partitioning is created per table which does not lead to co-partitioned tables.

*Apache Accumulo* [1] is another open source key-value store which is inspired by the architecture of BigTable and uses HDFS to store data and *Apache Zookeeper* for coordinating the distributed system. Techniques such as SSTables and Apache ZooKeeper are used also in BBoxDB.

In [26], an spatio-temporal indexing extension to Accumulo is discussed. The approach employs GeoHashing to map spatio-temporal data to the key-value store. In contrast to BBoxDB, the extension focuses only on spatio-temporal data. It allows one to store point data and objects with an extent. This approach is the primary goal of the *GeoMesa* project [37] [29]. GeoMesa is a spatio-temporal database built on top of existing NoSQL databases such as Cassandra, HBase or Accumulo. Two and

three-dimensional objects with and without an extent are supported. In contrast to the original paper, GeoMesa uses space-filling curves (the *Z-Curve* [45] for point data and the *XZ-Ordering* [16] for data with a spatial extension) to index the data.

The original paper and GeoMesa only allow one to handle three-dimensional data (with the dimensions: longitude, latitude and time). BBoxDB in contrast is not bound to a fixed number of dimensions. In addition to this, GeoMesa adds an index layer to existing NoSQL databases. BBoxDB instead provides a new data store implementation, designed to handle multi-dimensional data and to support range queries. *GeoWave* [59] is also a system which provides an index layer for Accumulo, HBase or Google BigTable. Point and non-point data up to three dimensions are supported. However, in contrast to BBoxDB data of other dimensions are not supported and the data distribution is done per table. Therefore co-partitioned data partitions are not directly supported but the software provides the ability to implement own indexing / partitioning strategies.

DISTRIBUTED SECONDO [46] employs the single computer DBMS SECONDO [31] 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. BBoxDB, in contrast, supports dynamical data partitioning, which leads to an equal data distribution across the nodes. In addition, BBoxDB can work with multi-dimensional data directly, no additional distribution layer has to be implemented.

*Spatial Hadoop* [23] 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. 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.

## 6 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 nodes. The experiments have shown that BBoxDB is capable of splitting various datasets into almost equal-sized distribution regions. BBoxDB supports data replication and does not provide a single point of failure. The software is licensed under the Apache 2.0 license and is available for download from the website of the project [10].

In the upcoming versions, we plan to enhance the compactification tasks and store a particular 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. We plan to implement performance counters (e.g., the number of unflushed memtables or received network operations per second) to get more insights into the behavior of the system. BBoxDB has been examined on a cluster with 10 nodes; we plan to execute more experiments in a cluster with more nodes.

Our current research is dedicated to the question of which algorithms are best suited as space partitioners. The presented K-D Tree (and the implemented Quad-Tree) are only capable to merge certain regions. We are planning to publish further studies regarding this topic in another paper. In addition, we will add more functions to BBoxDB like the handling of data streams and enhance the GUI in a way that the user can directly execute queries. Besides, we plan to support user defined filter, which makes BBoxDB capable of performing operations based on the values of the data.

**Acknowledgements** We are grateful for the free license of *JProfiler*, which *ej-technologies GmbH* provided for the BBoxDB open source project. The profiler helped us to speed up the implementation of BBoxDB significantly.

## References

1. Website of the Apache Accumulo project. https://accumulo.apache.org/, 2017. Online Accessed 05 Oct 2017
2. Website of the Apache CouchDB project, 2018. http://couchdb.apache.org/. Online Accessed 15 April 2018
3. Website of Apache Hadoop project. http://hadoop.apache.org/, 2018. Online Accessed 15 April 2018
4. Apache software license, version 2.0, 2004. http://www.apache.org/licenses/. Online Accessed 15 May 2017
5. Athanassoulis, M., Kester, M.S., Maas, L.M., Stoica, R., Idreos, S., Ailamaki, A., Callaghan, M.: Designing access methods: The RUM conjecture. In Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15–16, 2016, Bordeaux, France, March 15–16, 2016, pp 461–466, (2016)
6. Baker, H.C., Hewitt, C.: The incremental garbage collection of processes. In Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, pp. 55–59, New York, NY, USA, ACM (1977)
7. BBoxDB at the maven repository, 2018. https://maven-repository.com/artifact/org.bboxdb. Online Accessed 15 April 2018
8. Docker image of BBoxDB on DockerHub, 2018. https://hub.docker.com/r/jnidzwetzki/bboxdb/. Online Accessed 15 April 2018
9. Website of Docker Compose, 2018. https://docs.docker.com/compose/. Online Accessed 15 April 2018
10. Website of the BBoxDB project. http://bboxdb.org, 2018. Online Accessed 03 Feb 2018
11. Website of the Docker project, 2018. https://www.docker.com/. Online Accessed 15 Apr 2018
12. The network protocol of BBoxDB. https://jnidzwetzki.github.io/bboxdb//dev/network.html, 2019. Online Accessed 16 Jul 2019
13. Cassandra based baseline approach for performance evaluation, 2018. https://github.com/jnidzwetzki/bboxdb/blob/master/bboxdb-experiments/src/main/java/org/bboxdb/experiments/TestBaselineApproach.java. Online Accessed 21 Oct 2018
14. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
15. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
16. Böhm, C., Klump, G., Kriegel, H.P.: XZ-Ordering: A Space-Filling Curve for Objects with Spatial Extension. In Advances in Spatial Databases, 6th International Symposium, SSD'99, Hong Kong, China, July 20–23, 1999, Proceedings, pp. 75–90 (1999)
17. Bracciale, Lorenzo, Bonola, Marco, Loreti, Pierpaolo, Bianchi, Giuseppe, Amici, Raul, Rabuffi, Antonello: CRAWDAD dataset roma/taxi (v. 2014-07-17). Downloaded from https://crawdad.org/roma/taxi/20140717, July (2014)
18. Cassandra—ALLOW FILTERING explained, 2019. https://www.datastax.com/dev/blog/allow-filtering-explained-2. Online Accessed 15 Jul 2019
19. Cassandra—Create Index, 2019. https://docs.datastax.com/en/archived/cql/3.3/cql/cql_reference/cqlCreateIndex.html. Online Accessed 15 Jul 2019

20. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 4:1–4:26 (2008)
21. Transaction Processing Performance Council. TPC BENCHMARK H (Decision Support) Standard Specification. http://www.tpc.org/tpch/. Online Accessed 22 April 2018
22. Website of Elasticsearch. https://www.elastic.co/products/elasticsearch/, 2018. Online Accessed 23 April 2018
23. Eldawy, A., Mokbel, M.F.: SpatialHadoop: A MapReduce Framework for Spatial Data. In 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13–17, 2015, pp. 1352–1363, (2015)
24. Escriva, R., Wong, B., Sirer, E.G.: HyperDex: A Distributed, Searchable Key-value Store. In Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '12, pp 25–36, New York, NY, USA, ACM (2012)
25. Finkel, R.A., Bentley, J.L.: Quad trees a data structure for retrieval on composite keys. Acta Inf. **4**(1), 1–9 (1974)
26. Fox, A., Eichelberger, C., Hughes, J., Lyon, S.: Spatio-temporal indexing in non-relational distributed databases. In 2013 IEEE International Conference on Big Data, pages 291–299, October (2013)
27. Website of GeoCouch. https://github.com/couchbase/geocouch, 2018. Online Accessed 23 April 2018
28. The Wikipedia article about Geohashing. https://en.wikipedia.org/wiki/Geohash, 2018. Online Accessed 03 Feb 2018
29. Website of the GeoMesa project. http://www.geomesa.org, 2017. Online Accessed 05 Oct 2017
30. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google File System. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pp 29–43, New York, NY, USA, ACM (2003)
31. Güting, R.H., Behr, T., Düntgen, C.: Secondo: A platform for moving objects database research and for publishing and integrating research implementations. IEEE Data Eng. Bull. **33**(2), 56–63 (2010)
32. Güting, R.H., Schneider, M.: Moving Objects Databases. Morgan Kaufmann, Los Altos (2005)
33. Guttman, A.: R-trees: a dynamic index structure for spatial searching. SIGMOD Rec. **14**(2), 47–57 (1984)
34. Han, D., Stroulia, E.: HGrid: A Data Model for Large Geospatial Data Sets in HBase. pp. 910–917, 06 (2013)
35. Website of Apache HBase. https://hbase.apache.org/, 2018. Online Accessed 12 Feb 2018
36. HBase - Secondary Index, 2019. http://hbase.apache.org/book.html#secondary.indexes. Online Accessed 15 Jul 2019
37. Hughes, J., Annex, A., Eichelberger, C., Fox, A., Hulbert, A., Ronquest, M.: Geomesa: a distributed architecture for spatio-temporal fusion. In Geospatial Informatics, Fusion, and Motion Video Analytics V, 94730F, volume 9473 of Proceedings SPIE, pp. 9473–9473–13, (2015)
38. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: 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, USENIX Association (2010)
39. ST_Intersects - Spatial Relationships and Measurements, 2018. http://postgis.net/docs/ST_Intersects.html. Online Accessed 15 May 2018
40. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: 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, pp. 654–663, New York, NY. USA, ACM (1997)
41. Kleppmann, M.: Designing Data-Intensive Applications. O'Reilly, Beijing (2017). ISBN 978-1-4493-7332-0
42. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
43. Li, S., Hu, S., Ganti, R.K., Srivatsa, M., Abdelzaher, T.F.: Pyro: A Spatial-Temporal Big-Data Storage System. In 2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8–10, Santa Clara, CA, USA, pp. 97–109, (2015)
44. Website of MongoDB project. https://www.mongodb.com/, 2018. Online Accessed 23 Feb 2018
45. Morton, G.M.: A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. International Business Machines Company, Ottawa (1966)

46. Nidzwetzki, J.K., Güting, R.H.: Distributed secondo: an extensible and scalable database management system. Distrib. Parallel Databases **35**(3–4), 197–248 (2017)
47. Nidzwetzki, J.K., Güting, R.H.: BBoxDB - A Scalable Data Store for Multi-Dimensional Big Data (Demo-Paper). In Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM '18, pp. 1867–1870, New York, NY, USA, ACM (2018)
48. Nishimura, S., Das, S., Agrawal, D., Abbadi, A.E.: 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, pp. 7–16, Washington, DC, USA, IEEE Computer Society (2011)
49. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (lsm-tree). Acta Inf. **33**(4), 351–385 (1996)
50. Website of the Open Street Map Project, 2018. http://www.openstreetmap.org. Online Accessed 15 Apr 2018
51. Orenstein, J.: A comparison of spatial query processing techniques for native and parameter spaces. SIGMOD Rec. **19**(2), 343–352 (1990)
52. Tamer Özsu, M., Valduriez, P.: Principles of Distributed Database Systems, 3rd edn. Springer, New York (2011)
53. Patel, J.M., DeWitt, D.J.: Partition based spatial-merge join. SIGMOD Rec. **25**(2), 259–270 (1996)
54. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann Publishers Inc., San Francisco (2005)
55. Sprugnoli, R.: Perfect hashing functions: a single probe retrieving method for static sets. Commun. ACM **20**(11), 841–850 (1977)
56. Modified version of Tiny MD-HBase on Github, 2018. https://github.com/jnidzwetzki/Tiny-MD-HBase. Online Accessed 15 Jul 2018
57. The Tiny MD-HBase project on Github, 2018. https://github.com/shojinishimura/Tiny-MD-HBase. Online Accessed 26 Apr 2018
58. Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (2009)
59. Whitby, M.A., Fecher, R., Bennight, C.: GeoWave: Utilizing Distributed Key-Value Stores for Multi-dimensional Data. In Advances in Spatial and Temporal Databases - 15th International Symposium, SSTD 2017, Arlington, VA, USA, August 21–23, 2017, Proceedings, pp. 105–122, (2017)
60. Zhang, S., Han, J., Liu, Z., Wang, K., Xu, Z.: 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, pp. 1–8, (2009)
61. Zhou, X., Zhang, X., Wang, Y., Li, R., Wang, S.: Efficient Distributed Multi-dimensional Index for Big Data Management. In Proceedings of the 14th International Conference on Web-Age Information Management, WAIM'13, pp. 130–141, Berlin, Heidelberg, Springer (2013)