INFORMATIK
BERICHTE

383 – 06/2021

BBoxDB Streams - Scalable Processing of Multi-Dimensional Data Streams

Jan Kristof Nidzwetzki
Ralf Hartmut Güting

FernUniversität in Hagen

Fakultät für Mathematik und Informatik
D-58084 Hagen

BBoxDB Streams – Scalable Processing of Multi-Dimensional Data Streams

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

June 14, 2021

Abstract

BBoxDB Streams is a distributed stream processing system, which allows the handling of multi-dimensional data. Multi-dimensional streams consist of n-dimensional elements, such as position data (e.g., two-dimensional positions of cars or three-dimensional positions of aircraft). The software is an enhancement of BBoxDB, a distributed key-bounding-box-value store that allows the handling of n-dimensional big data. BBoxDB Streams supports continuous range queries and continuous spatial joins; n-dimensional point and non-point data are supported. Operations in BBoxDB Streams are performed primarily on the bounding boxes of the data. With user-defined filters (UDFs), custom data formats can be decoded, and the bounding box-based operations are refined (e.g., a UDF decodes and performs intersection tests on the real geometries of WKT encoded stream elements). A unique feature of BBoxDB Streams is the ability to perform continuous spatial joins between stream elements and previously stored multi-dimensional big data. For example, the dynamic position of a car can be efficiently joined with the static spatial data of a street network.

1 Introduction

Data streams consisting of multi-dimensional elements are ubiquitous. For instance, when the position of a moving object is determined continuously and stored, a data stream is created. Every observed position can be considered as a two-dimensional stream element.

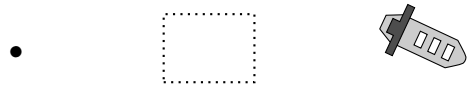
Examples of data streams containing multi-dimensional data are: (1) the price of a stock, (2) the position of a car or a ship, or (3) the position of an aircraft. The first stream consists of price values; each price can be considered as a point in the one-dimensional space. The second stream consists of coordinates in the two-dimensional space (i.e., longitude and latitude of the vehicle). The last stream consists of coordinates in the three-dimensional space (i.e., the longitude, the latitude, and the altitude of the aircraft).

Example. A very simple stream of two-dimensional position data looks as follows: `(54.0044, 8.65926), (53.9336, 8.69052), (53.8972, 8.78316), ....` Each bracket pair represents a stream element, consisting of a *WGS84* [53] coordinate. □

Apart from simple bracket structured data, well-known data formats such as *CSV* or *JSON* can be used for the encoding of the stream elements. This makes it possible to add further information such as the observation time or the id of the entity (e.g., a license plate or a flight number) to the stream elements.

1.1 Multi-Dimensional Data Streams

There are many different types of multi-dimensional data. Such data can have: (1) an extension in space (e.g., the geometry of a ship) or (2) no extension in space (e.g., the coordinates of a point). In the first case, we call the data of the object *non-point data* while in the second case *point data*. Often, multi-dimensional data describe the position of an object of the real-world. The position of an object can be described in the following ways: (1) as a point in space, (2) as an n-dimensional axis-aligned bounding box, and (3) with the full geometry of the object (see Figure 1).



(a) Point in space (b) Bounding box (c) Complete geometry

Figure 1: Three ways to describe the position of an object (e.g., a ship) in the two-dimensional space.

Using the *complete geometry* is the most precise way to describe the position of an object in space. In contrast, the *bounding box*[1] gives only a rough estimation of the location of the object in space. However, the simple structure of a bounding box makes operations like intersection tests fast to calculate; such operations are expensive to compute on the complete geometry of an object.

The most inaccurate way to describe an object is a *point in space*. The whole object, whether it has an extension or not, is described only by an n-dimensional point. Many real-world data streams such as: (1) ADS-B (*Automatic Dependent Surveillance-Broadcast*), which contain the position data of aircraft, (2) AIS (*Automatic Identification System*), which contain the positions of ships, or (3) GTFS real-time (*General Transit Feed Specification*), which contain the position data of public transport vehicles, express the position only with point data[2].

Definition. A *multi-dimensional data stream* S_n of the dimensionality n is a potentially unbounded continuous sequence of *stream elements* e, $S_n = (e_0, e_1, e_2, \dots, e_\infty)$. Each stream element $e = (id, t, value_n, \dots)$ consists at least of an *object identifier*, denoted as id, the *event time* t when the element was produced, and an n-dimensional value $value_n$. □

1.2 Processing Data Streams

Stream processing systems [1] are software systems developed to handle data streams. In contrast to database management systems, which store data first and answer queries afterward, stream processing systems work in exactly the opposite way. In the first step, queries are registered, and afterward, a data stream is processed [68]. Every element in the data stream that fulfills a query predicate is reported. Depending on the stream processing system that is used, the query predicate can contain complex filters or transformations.

The near real-time (*low-latency*) processing of data streams is essential for many areas of application. For example, on a stream of position data, the following queries might be interesting: (1) Is an aircraft about to enter a certain airspace? (2) Which taxis are located near a passenger? (3) Has a convoy of vehicles broken apart and the cars do not drive next to each other anymore? All these queries need to be answered fast so that actions based on these events can be executed. Figure 2 shows a further application from the field of maritime observation.

The figure depicts the real geometries and the bounding boxes of a ship and a reef. A continuous query is registered, which should report all ships that come dangerously close to a reef. To be able to warn the ship about a possible collision, the bounding box of the ship is enlarged by a constant

[1] The axis-aligned minimal bounding box is the smallest box in space that encloses the object completely. In the rest of this paper, the terms *axis-aligned minimal bounding box* and *bounding box* are used as synonyms.

[2] When point data (data without an extension in space) is used, the bounding box degenerates also to a point in space; the point and the bounding box are identical.

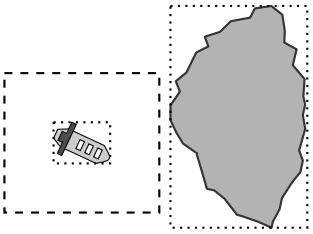


Figure 2: Does a ship collide with a reef? The bounding boxes of the ship and the reef are shown as dotted boxes. In addition, the enlarged bounding box of the ship is shown as a dashed box.

factor[3]. Therefore, the intersection between the enlarged bounding box of the ship and the bounding box reef is reported by the query before the reef is actually hit.

1.3 Challenges

We have identified five major challenges when multi-dimensional data streams are processed:

Scalability: The number of elements in a data stream per time unit can fluctuate significantly; a large number of stream elements can occur within a short period of time. The receiving system needs to process the elements with a low latency delay. A scalable and distributed approach is required, which is capable of utilizing the resources of multiple nodes.

Multi-dimensional data: Objects of any dimensionality can be contained in a data stream. For example, the position of a car is two-dimensional (latitude and longitude); the position of an aircraft also contains an altitude and is three-dimensional. Therefore, the proposed solution should be able to handle n-dimensional data.

Non-Point data: A data stream can contain point and non-point data; the solution should be able to process both types of data (see Section 1.1).

Spatial Joins: For some applications, the stream elements need to be joined with already stored data. For example, the position of a car needs to be joined with a road network to determine in which street the car is currently located. Therefore, the solution needs to provide efficient access to previously stored multi-dimensional big data.

Transformations and Data Distribution: Some continuous queries require the transformation of stream elements (e.g., the enlargement of the bounding box of the spatial join as depicted in Figure 2 in Section 1.2). To execute such a join efficiently, the stream data needs to be distributed to nodes that contain the join partners. Joins in distributed stream processing systems are usually performed in two steps: (1) the data are distributed, and (2) the continuous join queries are executed by the nodes. The enlargement of the stream elements is performed in the continuous join query. This means that the enlargement of the stream elements is calculated after they are distributed, which makes it hard to distribute the data to all needed nodes. However, spatial joins that contain transformations should be supported by a stream processing system.

1.4 Proposed Solution

In this paper, we propose *BBoxDB Streams* [57] as a novel solution to process multi-dimensional data streams. The paper provides the following major contributions:

- A generic stream processing solution for multi-dimensional data, which supports n-dimensional point and non-point data.

[3]Instead of enlarging the bounding box of the ship, the bounding box of the reef could also be enlarged.

- A novel approach for the execution of continuous spatial joins between static multi-dimensional big data and multi-dimensional data streams.
- An efficient way to express queries (e.g., continuous range queries and spatial joins) and tranformations on data streams and stored data.
- A solution to distribute stream elements to nodes, even when transformations are applied and spatial joins should be performed.
- A GUI which can be used to execute queries on real-world data streams.
- A free implementation that is available under an open-source license on GitHub [14].

BBoxDB Streams is an extension of *BBoxDB* [56]; a distributed key-bounding-box value store, optimized for the handling of multi-dimensional data. To the best of our knowledge, BBoxDB is the first freely available data store that is capable of handling n-dimensional point and non-point big data efficiently; data can be retrieved in $\mathcal{O}(\log n + k)$ time. In contrast to a key-value store, BBoxDB stores each value together with a bounding box. The space is partitioned dynamically into distribution regions, and these regions are assigned to the nodes of a cluster. In BBoxDB, operations are executed primarily on the bounding boxes of the data. *User-defined filters* (UDFs) are used to enhance the generic query processor [55]; they refine the query results and decode custom data formats. In BBoxDB Streams, UDFs are used to decode the various data stream formats and process the geometries of the stream elements (see Section 3.6).

The rest of the paper is organized as follows: Section 2 gives an overview of the related work. Section 3 contains the details of BBoxDB that are required for the understanding of BBoxDB Streams. Section 4 describes our solution for the handling of data streams. Section 5 shows how queries can be expressed and executed. Section 6 contains a performance evaluation of BBoxDB Streams, while Section 7 concludes the paper.

2 Related Work

The handling of data streams, the execution of continuous queries, and the calculation of joins has related work in the areas of: (1) *rule engines* (the evaluation of query predicates), (2) *database management engines* (storing large amounts of data and allow the efficient data retrieval), and (3) *stream processing systems* (the handling of continuously changing values).

The related work of these areas is discussed in the following sections. The related systems are compared with BBoxDB (for the data storage part) and BBoxDB Streams (for the stream handling part).

2.1 Rule Engines

Rule engines apply *condition and action pairs* (usually expressed as if-then statements) on datasets. One of the first rule engines was *PLANNER* [37]. PLANNER is a programming language that allows the user to provide problem-solving primitives together with hierarchical control structures. The *Official Production System 5* (*OPS5*) [25] is a *rule-based production language*. OPS5 uses two types of memory. The system stores: (1) data in the *working memory*, and (2) if-then rules in the *production memory*. When the content of the working memory matches the condition of a rule, the actions of the rule are applied. The logical programming language *PROLOG* [19] is also a rule processing system. The language allows to define *facts* and *rules*. A calculation is performed by running a query over the defined facts and rules. Newer approaches such as *Jess* [38] allow to define and evaluate rules in the programming language Java.

BBoxDB and rule engines share in common the fact that predicates (or rules) are evaluated on datasets. In contrast, rule engines are not optimized to store and access large amounts of data. In addition, the concept of data streams and the comparison of streams with stored data are not part of

these systems. BBoxDB is instead designed to store and access large amounts of multi-dimensional data efficiently.

2.2 Database Management Systems

For more than a decade, distributed storage and retrieval of large amounts of data (*big data*) has been an important topic in research. The related systems of these areas are discussed in the following sections.

2.2.1 Traditional Database Management Systems

Many *database management systems* (DBMS) that are used today focus on the relational data model; such systems are called *relational database management systems* (RDBMS) [59, 75]. They are optimized for ad-hoc query processing, transactions, and consistency. However, these features make it hard to scale well across a cluster of nodes [17]; most RDBMS are only capable of running on one node.

Compared to the relational data model, BBoxDB uses simpler key-bounding-box-value tuples. The software supports queries such as range queries or spatial joins on n-dimensional data (see also the discussion in Section 2.2.3). The queries are simpler than the queries supported by RDBMS, but BBoxDB stores the data in a distributed way and re-distributes unevenly partitioned data automatically in the background. Such concepts are not supported by most DBMS. Therefore, BBoxDB can work on larger datasets.

Extensible DBMS like *SECONDO* [33] do support more data models, such as nested relations or graphs. In addition, SECONDO can be easily extended by own operators or data models (called *algebra modules* [32]).

User-defined filters (see Section 3.6) extend the query processor of BBoxDB. This is a simpler but similar concept as the algebra modules used in SECONDO or user-defined functions in the open-source DBMS MySQL [50]. In BBoxDB Streams, these UDFs can be used to process data streams with custom data formats like GeoJSON (see Section 3.6.2 for an example).

2.2.2 Key-Value Stores

Key-value stores (KVS) belong to the family of NoSQL systems. They use a simple data model, consisting of *key* and *value* tuples. To store large amounts of data, they can be implemented as a *distributed key-value store* (DKVS), such as *Cassandra* [43] or *HBase* [5].

In a DKVS, the data are distributed across a cluster of nodes and each node stores only a partition of the data. A *partitioning function* like a *range-* or a *hash-partitioning function* is applied to the key of the tuple to determine to which node the tuple belongs.

(D)KVS provide simple methods to manage large amounts of key-value pairs. For storing data, they provide an operation such as `put(table, key, value)`. The given value is stored in a table under a particular key. For retrieving data, another operation such as `get(table, key)` is provided. This operation retrieves the stored value for a key from a table. (D)KVS focus on data storage, data streams are not supported, and the features to execute queries on the data are limited.

2.2.3 Multi-Dimensional Data in KVS

Most (D)KVS are optimized to handle one-dimensional data; handling n-dimensional data is laborious in these systems. Figure 3 depicts the problem for one- and two-dimensional data. In the figure, we assume that the shown customer record is retrieved only via the *customer_id* attribute. Therefore, this attribute can be used as the key in the KVS. Choosing the key for a road (two-dimensional non-point data) is much more difficult.

(D)KVS work with one-dimensional keys, and these keys are the only access path to the data. Because of this, the key has to support the query processing. For example, when the road's name (e.g., *Road 66*) is chosen as key for a tuple that stores the spatial data of a road (see Figure 3 (b));

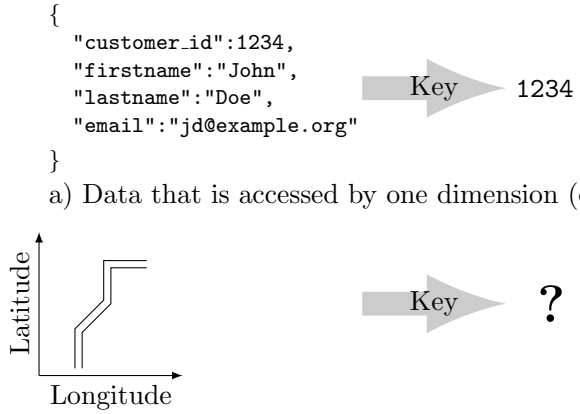


Figure 3: Determining a proper key for data in a (D)KVS can be a hard problem.

the road can be accessed by the name efficiently. However, the key does not contain information about the location of the road in space. It is impossible to deduce the needed keys to retrieve the data for a query such as *Which roads are located in the following area?* and in such cases an expensive full data scan has to be performed.

In general, multi-dimensional range queries are not directly supported in regular (D)KVS; a full data scan must be performed to answer such queries. A full data scan is an expensive operation; the complete dataset has to be read from disk. To answer queries efficiently, full data scans have to be avoided.

Linearization [49] can be used to encode the location of one-dimensional point data into a key. Systems like *MD-HBase* [58] are using this technique to work with such data. MD-HBase is a multi-dimensional extension of HBase, which allows the efficient storage of two-dimensional points. A K-D Tree or a QuadTree partitions the space, and linearization is employed to generate ids for these partitions. An additional indexing layer maps from these partition ids to HBase buckets, where the data are stored. Queries like range queries are performed by determining the required partitions and performing a scan of the affected buckets. BBoxDB works with n-dimensional data and can also store non-point data. In addition, no bucket scans are performed in BBoxDB. The local index (see Section 3.4) allows retrieving the required tuples directly.

The source code of MD-HBase is not publicly available. With *Tiny MD-HBase* [70] an open-source implementation does exist. *Tiny MD-HBase* is a sample implementation that shows the implementation aspects of the system. However, this version is not designed for handling large amounts of data.

Systems such as *EDMI - Efficient Distributed Multi-dimensional Index* [79], *Pyro* [44], and *HGrid* [36] are also enhancements of HBase which use an additional index layer to store multi-dimensional data in HBase. However, operations such as spatial joins or continuous queries are not supported by these systems. In addition, these systems only support point data.

HyperDex [23] is a distributed key-value store that supports multi-dimensional point data directly. However, non-point data are not supported by HyperDex. In [27], a spatio-temporal indexing extension to the key-value store *Apache Accumulo* [3] is discussed, which is based on *GeoHashing* [30]. However, this extension can handle only spatio-temporal (three-dimensional) data and does not support data streams. The *GeoMesa* project [31, 39] has developed a spatio-temporal database built on top of existing NoSQL databases like Cassandra, HBase, or Accumulo. Two- and three-dimensional point and non-point data are supported. Also, GeoMesa allows the handling of data streams. However, the software supports only very simple queries on data streams.

2.2.4 Further Approaches

Array databases such as *Rasdaman* (raster data manager) [12], *SciDB* [67], or *SciQL* [78] allow the processing of multi-dimensional arrays (data cubes). In addition to these dedicated software systems, raster data extensions for relational database management systems such as *PostGIS Raster* [65] or *Oracle GeoRaster* [62] exist. Array databases allow the handling of data like maps (two-dimensional) or satellite image time series (three-dimensional). These systems are optimized for storing and retrieving data. Handling data streams or continuous queries is not covered by all these systems.

Parallel SECONDO [34] and *Distributed SECONDO* [54] are distributed versions of SECONDO. They also allow the handling of multi-dimensional data. However, the distribution of the data is based on a static grid, and uneven partitions are not re-balanced automatically. In addition, continuous queries are not supported by these systems.

MapReduce [20] is an approach for the processing of large amounts of data on a cluster of unreliable nodes. With *Apache Hadoop* [4], an open-source implementation of the MapReduce algorithm does exist. Specialized extensions for the processing of spatial data (like *SpatialHadoop* [22]) or Spatio-Temporal Data (like *ST-Hadoop* [2]) were developed. However, these systems do not support to process data streams.

2.3 Stream Processing Systems

This section compares popular stream processing systems and common architecture patterns with BBoxDB Streams. Besides, other approaches for the handling of multi-dimensional stream data are discussed.

2.3.1 First Stream Processing Systems

The *ST*anford st*RE*amdat*A* *M*anagement (*STREAM*) system [10] was one of the first publicly available systems to evaluate continuous queries over data streams. The system uses a centralized architecture and does not focus on the special characteristics of multi-dimensional data. In contrast, BBoxDB is a distributed system that is specialized in the handling of multi-dimensional data.

The *Tapestry system* [68] introduces the concept of *continuous queries*. Continuous queries are registered by a user and evaluated as soon as new data are stored. Tapestry is an append-only database, and the continuous queries are formulated in the *Tapestry Query Language*, which is similar to SQL. In contrast to BBoxDB, data can not be deleted, and the system can only utilize the resources of a single node.

2.3.2 Current Stream Processing Systems

The Apache project hosts four of the most widespread stream processing systems: *Apache Flink* [28], *Apache Spark Streams* [51], *Apache Kafka* [52], and *Apache Storm* [9].

Modern stream processing systems allow splitting up the stream into so-called *windows*. A window is a small partition of stream elements (e.g., based on the number of elements or the time) that are processed at once. On these windows, operations such as aggregations can be executed. Some systems allow to build up distributed tables of previous stream elements or with the most recent stream value (e.g., *KTables* in Apache Kafka [8]). These tables are stored in a distributed manner, but features such as the dynamic re-partitioning of these tables are not supported. However, stream processing systems provide the ability to store the result of the stream processing into *data sinks* such as Cassandra. These stream processing systems are not optimized to compare the elements of the streams with larger previously stored datasets. These systems do not support operations such as geometric indexing or spatial joins. BBoxDB instead re-distributes datasets dynamically in the background without interrupting access to the data. The system supports multi-dimensional indexing and spatial joins are supported out of the box. BBoxDB Streams allows performing continuous spatial joins between the elements of a data stream and stored data.

2.3.3 Data Streams and Continuous Joins

Joining previously stored data with stream elements is an important topic for many areas of application. In general, joins in stream processing systems can be divided into: (1) *windowed joins* and (2) *unwindowed joins* [1, p. 254]. In a windowed join, the data of the stream is joined with a *fixed range* (e.g., a fixed time range of one day) or a *sliding range* (e.g., the last 100 elements of the stream) of the previously received data. In an unwindowed join, the data of the stream are joined with the complete history of the stream.

Performing joins on streams is discussed in papers like [41, 64]. Both papers propose *Distributed Hash Tables* [45] to distribute the tuples of a data stream in a way that joins can be efficiently executed. In contrast to BBoxDB Streams, these systems focus only on one-dimensional point data. Systems such as *DEDUCE* [71] allow the usage of MapReduce jobs to access large amounts of static data in stream processing systems. These systems allow joining data streams with static data; however, they focus also on one-dimensional key-value pairs, which MapReduce can process. Topics such as spatial data are not covered by these systems.

A unique feature of BBoxDB Streams is the efficient continuous unwindowed spatial join between a data stream and previously stored n-dimensional big data.

We have chosen Apache Kafka for a more detailed comparison to discuss the differences between BBoxDB Streams and a typical stream processing system in-depth. The architecture of Apache Kafka and the implemented join types are discussed subsequently. The comparison is also valid for other typical stream processing systems.

In Apache Kafka, a data stream is called *KStream*. Each element of a KStream consists of a key and a value. The key is used to partition the stream elements across the nodes of the cluster. Each node processes only a partition of the whole data stream. A *KTable* is an "*abstraction of a changelog stream*" [8]; for each key it contains the most recent value. Just like the data stream, KTables are partitioned across the nodes of the Kafka cluster. *GlobalKTables* are similar to KTables, with the difference that they are not partitioned; each Kafka node stores a full copy of the table. Therefore, GlobalKTables allow joins over non-key stream attributes. Regardless of how the stream data are partitioned, the join partners are present on the nodes.

Apache Kafka supports five different join operations [7]: (1) the *KStream-KStream Join*, (2) the *KTable-KTable Join*, (3) the *KStream-KTable Join*, (4) the *KStream-GlobalKTable Join*, and (5) the *KTable-to-GlobalKTable Join*. The KStream-KStream join is a windowed join; the remaining joins are unwindowed.

Similar to a key-value store, Kafka works with simple data types for the key and the value of the stream elements (e.g., `byte`, `string`, `int`, `long`) [6]. However, this leads to the same problems as discussed in Section 2.2.3 when multi-dimensional data or non-point data have to be processed. Determining a proper key is problematic for such kind of data. Therefore, it is problematic to distribute a stream of n-dimensional data across a cluster of nodes in a way that KStream-KTable joins can be performed. To perform this type of join, the stream elements have to be partitioned in the same way as the KTable.

As an alternative, the KStream-Global-KTable join can be performed. However, in this case, the table is not partitioned and has to be stored on each Kafka node completely. The resources of the nodes limit the size of the table. Therefore, such joins can only be performed on small tables.

The spatial join of BBoxDB Streams can be compared to the *KStream-KTable Join* of Apache Kafka. The data of the table containing the join partners of the stream elements are partitioned across the cluster of nodes; the stream is partitioned in the same way. So, the stream elements are processed by the same nodes that store the join candidates. The difference between Kafka and BBoxDB Streams is that BBoxDB Streams uses a partitioned n-dimensional space and n-dimensional bounding boxes of the stream elements to distribute the data. Apache Kafka uses only a one-dimensional key to distribute the data, which leads to the problems discussed above.

2.3.4 Data Streams and Multi-Dimensional Data

The systems *PLACE* [48] and *Tornado* [46] are built to handle data streams of spatial data. PLACE focuses on spatio-temporal data streams and supports continuous queries, and implements operations like spatial joins. However, the system is not a distributed system. Therefore, the system can only employ the resources of one node, to handle the data stream and to calculate the continuous queries. Tornado focuses on spatio-textual datasets (e.g., geo-tagged text messages from Twitter). The distributed architecture allows the system to process data streams and register continuous queries. In contrast to BBoxDB, the system can handle only point data with two spatial dimensions and one temporal dimension. In addition, both of these systems are not publicly available, and they focus only on two- and three-dimensional data.

In [76] an extension of Apache Storm for the handling of spatial data streams is proposed. However, the paper focuses only on two-dimensional point data. In contrast, BBoxDB Streams can handle n-dimensional point and non-point data.

The paper [77] focuses on the handling of continuous spatial joins on moving objects. However, it does not cover topics such as the handling of previously stored n-dimensional big data or the distributed computing on a cluster of nodes.

When a data stream of position data is processed, the continuous queries are evaluated on each position update. This requires a certain amount of resources. To reduce the needed resources, concepts such as the *safe region* [66] were introduced. A safe region is a region in space where the object can move without changing the result of a continuous query; this reduces the number of calculations required. The safe region gives only a rough estimation of the location of an object. Applications that need the exact position of the object can not work with safe regions. BBoxDB is a highly scalable system that is designed to handle large amounts of data and updates. Concepts to reduce the number of updates or query re-calculations are not implemented at the moment. Instead, more resources can be added easily, and the existing data are re-distributed in the background without interrupting the service. Besides, the GUI of BBoxDB is used to visualize the stream data in real-time. Therefore, all changes need to be processed by the registered continuous queries (see Section 4.8).

2.3.5 Software Architecture

The *lambda architecture* [47] is a software architecture pattern that deals with the low-latency processing of data streams. A stream-processing system and a batch-processing system are used in parallel to get the best of both worlds: accurate and up-to-date results. The stream-processing system is used to get the most recent but inaccurate results. The batch-processing system re-processes the complete data periodically. The output of this system is a bit outdated but accurate. For the batch processing component, the complete data history has to be available. The drawback of the architecture is that the same logic has to be implemented twice: (1) in the batch-, and (2) in the stream-processing system.

A more recent approach is the *kappa architecture* [42]. The batch processing engine is passed, and so, the duplicate implementation of the logic is omitted. The data are still stored in an append-only way, but everything is treated as a data stream. The stream processing engine periodically re-processes the historical and the most recent data. The history data are read entry per entry and processed as a regular data stream.

In BBoxDB, continuous spatial join queries are performed between stream elements and previously stored data. The software can store the stream elements in a persistent way. In this situation, the spatial joins are performed between current stream elements and all previously stored stream elements. This means that BBoxDB Streams is inspired by the kappa architecture.

3 BBoxDB Basics

BBoxDB is a distributed key-bounding-box-value store designed to handle multi-dimensional big data. In contrast to regular key-value stores, which store key-value pairs, BBoxDB stores each value together with an n-dimensional bounding box. The system is optimized for the handling of low-dimensional (e.g., spatial and spatial-temporal) data[4]. BBoxDB is licensed under the Apache 2.0 license and is available for download on the website of the project [14]. More information about the concepts can be found in papers such as [55, 56].

Building a highly-available distributed system is a complex and error-prone task. Failures such as node or network outages have to be handled properly. *Apache ZooKeeper* [40] is a software that was developed to simplify the implementation of distributed systems. The software provides a simple tree-oriented structure, which can be used to realize more complex tasks. BBoxDB uses ZooKeeper for tasks such as service discovery (i.e., which BBoxDB nodes are available) or sharing information (e.g., which part of the space is handled by which node).

3.1 Tuples and Consistency

Definition. A tuple t consists of a *key*, a *value*, a *bounding box* and a *version*; $t = (key, bounding\ box, value, version)$. (1) The *key* identifies the tuple, (2) the *bounding box* is used as a generic description of the location of the tuple in the n-dimensional space (see Section 3.3), (3) the *value* contains the data, and (4) the *version* is used to identify the newest version of the tuple[5]. Table 1 lists the attributes of a tuple. □

Name	Description
Key	The *key* is a `string` that identifies the tuple.
Bounding box	The *bounding box* is an n-dimensional hyperrectangle consisting of `double` values.
Value	The *value* is a `byte array` that contains the data of the tuple.
Version	The *version* is a `long` which identifies the most recent version of a tuple.

Table 1: The attributes of a tuple in BBoxDB.

BBoxDB uses *eventual consistency* [72] to deal with outages such as network partitions or unavailable replicates. The timestamp of the tuple is used to keep track of the most recent version of a tuple. Eventual consistency means that all replicates become eventually synchronized with the last version of the data when no updates are made.

3.2 Operations

BBoxDB has some similarities to key-value-stores. But in contrast to a KVS, BBoxDB uses slightly different operations. New data are stored with the `put(table, key, hrect, value)` operation. In addition to the already described parameters, an n-dimensional bounding box (a hyperrectangle; see Section 3.3 for a detailed description) has to be specified when the data are stored.

Data are retrieved using the operation `queryByRect(table, hrect)`, which retrieves all tuples whose bounding box intersects with the query bounding box. Besides, further operations such as a spatial join (operation `join(table1, table2, hrect)`) are implemented. Table 2 lists BBoxDB operations that are used in this paper[6].

BBoxDB is a generic data store that can store any kind of data (e.g., a geometry encoded in GeoJSON or WKT format, or a trajectory encoded in CSV format). Each stored value is a plain

[4] In general, BBoxDB can handle data of any dimension. However, due to the "*the curse of dimensionality*" [15, p. ix] and the increasing volume of hyperectangles (used as bounding boxes, see Section 3.3) the data model can become

Most important operations of BBoxDB		
`put`	table × string × hrect × bytes	→ table
`delete`	table × string	→ table
`getByRect`	table × hrect	→ stream(tuple)
`join`	table × table × hrect	→ stream(tuple)

Table 2: The most important operations of BBoxDB.

array of bytes for the datastore. Only the bounding box of the value is encoded in a format that BBoxDB can decode. Therefore, the query processor takes only the bounding boxes of the tuples into consideration. Some operations, such as spatial joins on polygons, need to decode the stored values to produce the correct query result. User-defined filters (UDFs) can decode the value and can refine the bounding box based operations (see Section 3.6).

3.3 Bounding Boxes

In BBoxDB, each value is stored together with an n-dimensional bounding box. To calculate the bounding box for a value, the minimum and maximum coordinates have to be determined for each dimension. BBoxDB provides some helper functions which calculate a bounding box for common data formats like GeoJSON, WKT, or GTFS.

Definition. The *n-dimensional axis-aligned minimum bounding box*, which is simply called bounding box in this paper, is represented by an n-dimensional hyperrectangle consisting of $2n$ values of the datatype `double`. The value $2(i-1)$ describes the lowest included coordinate in the dimension i, while the value $2(i-1)+1$ describes the highest included coordinate in the dimension i. For example, the tuple $(0.5, 2.5, 0.2, 3.0)$ describes a two-dimensional hyperrectangle. In the first dimension, the range $[0.5, 2.5]$ and in the second dimension, the range $[0.2, 3.0]$ are covered (see Figure 4). □

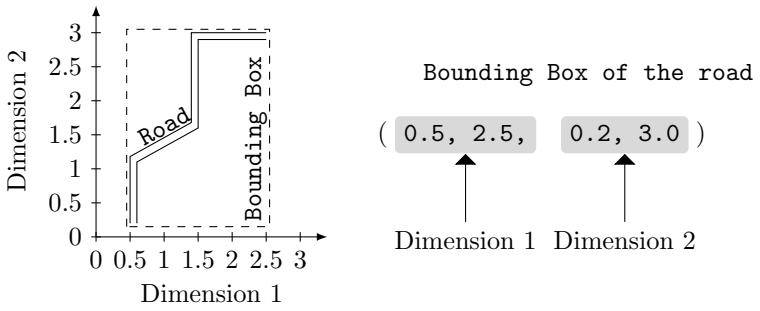


Figure 4: The bounding box for a two-dimensional non-point entity (e.g., a road).

3.4 Data Distribution

BBoxDB stores tuples in tables. Tables of the same dimensionality can be grouped together in a *distribution group*. To address a table in BBoxDB, the complete name consisting of the name of the distribution group and the table has to be specified. For example, `dgroup_table1` denotes the table `table1` of the distribution group `dgroup`.

By splitting the space, BBoxDB splits the data of a distribution group into almost equal-sized partitions (*distribution regions*) and spreads the data of these partitions across a cluster of nodes. Therefore, each node stores only a part of the whole dataset.

inefficient in the high dimensional space.

[5]The version can be provided by the user; if no version is provided the current timestamp is used.

[6]BBoxDB provides further operations for the management of distribution groups and tables. These operations are discussed in papers such as [56, pp. 14ff.].

Geometric data structures (such as the K-D Tree [16] or the Quad-Tree [24]) are used as *space partitioner* to partition the space into distribution regions. The space is split and merged based on the actual distribution of the stored tuples. The used partitioning algorithm can be chosen when the distribution group is created.

The *global index* contains two types of information: (1) the current *partitioning* that is generated by the space partitioner (*space* $\rightarrow$ *distribution region*) and (2) the *assignment* of these partitions to the nodes of the cluster (*distribution region* $\rightarrow \mathcal{P}(nodes)$)[7].

On the nodes, data are indexed by an *R-Tree* [35]; this data structure is called the *local index*. The local index maps from the n-dimensional space to the stored tuples (*space* $\rightarrow$ *tuples*). Both indexes are stored in ZooKeeper.

Figure 5 shows an example of stored tuples, their bounding boxes, and the partitioned space. The mapping between the space and the nodes is the global index. All the tuples whose bounding box belongs to multiple distribution regions (e.g., *Tuple G* in the figure) are duplicated and stored multiple times.

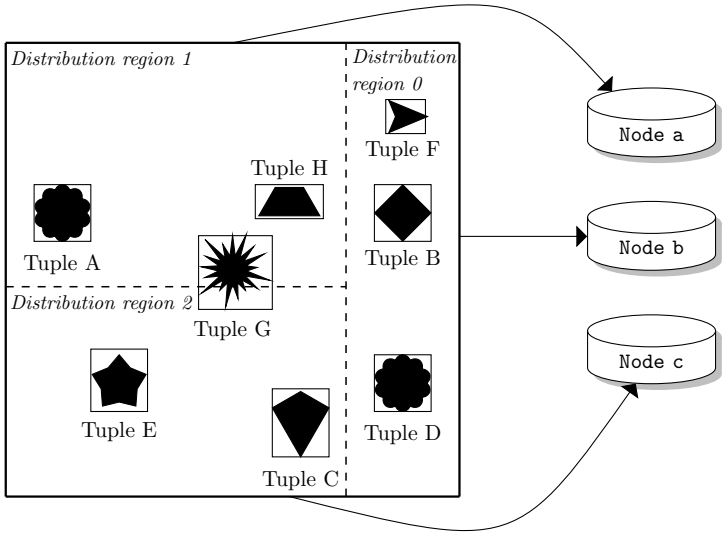


Figure 5: BBoxDB partitions the space into distribution regions and assigns these regions to the nodes of a cluster. The different symbols of the tuples represent the different values. The box around the symbol represents the bounding box of the value.

BBoxDB is designed to handle growing and shrinking datasets. To ensure an equal data distribution, the data are re-partitioned and re-distributed dynamically. When a distribution group is created, an upper- and a lower threshold (t_{upper} and t_{lower}) are defined. Each node of the BBoxDB cluster calculates the size of the locally stored distribution regions periodically. When a distribution region becomes larger than t_{upper}, the region is split. When a region becomes smaller than t_{lower}, the region is merged. BBoxDB re-distributes the data in the background without interrupting access to the data[8].

When a new distribution group is created, the entire space is covered by one distribution region. By storing data and the re-partitioning of the space performed by BBoxDB, the space is partitioned into more and more distribution regions. It takes some time until the space is partitioned in enough partitions that each node can store at least the data of one distribution region. Nodes that do not store any distribution regions are idle. To utilize the nodes directly from the beginning and to reduce the amount of data re-distribution tasks, BBoxDB allows one to *pre-partition* the space of a new distribution region into n distribution regions by a provided sample. BBoxDB determines the distribution of the data from the sample and creates matching partitions. Because no data are

[7] When replication is used, one distribution region is mapped to multiple nodes.

[8] See [56, pp. 23ff.] for a detailed discussion of this functionality.

stored in the distribution region, the partitions can be created and assigned to the nodes without re-distributing any data. When data are stored after the region is pre-partitioned, the data is distributed directly to all nodes of the cluster.

3.5 Efficient Data Access

The two-level index structure of BBoxDB, consisting of (1) the global index, and (2) the local index allows the efficient retrieval of tuples. Most of the operations in BBoxDB take a hyperrectangle as a parameter. The hyperrectangle is compared with the global index to determine which distribution regions are affected by the operation. The nodes that are responsible for these regions are contacted, and the operation is performed on these nodes. The local index on the nodes is used to identify all the local stored tuples that intersect with the hyperrectangle[9].

All tables of a distribution group share the same *global index*. This means that the data are spread in the same manner (*co-partitioned*) across the nodes of the cluster; the same regions in space are stored on the same nodes.

Definition. For a join $\bowtie_p$, we call two partitioned tables $R = \{R_1, ..., R_n\}$ and $S = \{S_1, ..., S_n\}$ *co-partitioned* iff $R \bowtie_p S = \bigcup_{i=1,...,n} R_i \bowtie_p S_i$. □

On co-partitioned data, spatial joins (performed by the `join()` operation) can be efficiently performed. No data need to be transferred through the network; all join partners are stored on the same node. The `join()` operation also takes a hyperrectangle as a parameter. This hyperrectangle determines the area in space where the spatial join is performed. The global index is employed to determine the distribution regions on which the operation needs to be performed. These nodes are contacted and the local index on these nodes is used for an *index nested loop join*. Figure 6 illustrates a spatial join on two co-partitioned tables.

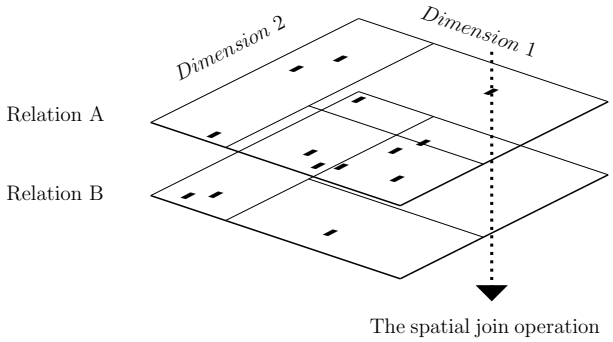


Figure 6: Executing a spatial-join on two co-partitioned tables.

3.6 User-Defined Filters

As a generic data store, BBoxDB is unable to interpret the bytes of the stored values. The bounding box is stored in a standardized format that BBoxDB can interpret. Therefore, the query processor can only perform operations on the bounding boxes of the data.

Example. Performing a spatial join on spatial data, which only considers the bounding boxes, leads to incorrect results. Intersecting bounding boxes is a necessary criterion but not a sufficient criterion for a spatial join (see Figure 7). □

To solve this problem, *user-defined filters* (UDFs) are supported by the query processor of BBoxDB. These filters contain the knowledge to decode a certain data format (e.g., GeoJSON encoded values) and to perform a certain operation on the data. UDFs refine the bounding box-based

[9]Operations without a hyperrectangle as a parameter can also be efficiently executed. This is realized with an additional *bounding box index*, which is discussed in [56, pp. 32ff.].

(a) Two non-intersecting spatial objects.

(b) Two intersecting spatial objects.

Figure 7: Two spatial objects (solid line) with intersecting bounding boxes (dashed line). In (a), the spatial objects do not intersect, while in (b), the spatial objects do intersect.

output of the generic query processor by applying a further filter step on the values (see Figure 8). These filters are deployed to the nodes of the cluster and executed directly in the query processor. Therefore, UDFs are executed in a distributed manner on different nodes, and the data is filtered before it is transferred to the client.

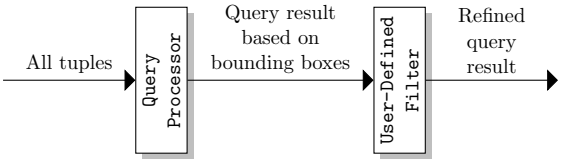


Figure 8: Refining the bounding box based result of the generic query processor with a user-defined filter. In a spatial join, a user-defined filter can consider the real geometries and let only the intersecting geometries pass.

3.6.1 Implementation Details

UDFs are developed by the user of the system. Only the user who has stored the data knows how to interpret the values of the data. However, BBoxDB contains some pre-defined UDFs for common data formats (e.g., for decoding GeoJSON data or WKT). UDFs are written in Java and they can use existing libraries. The filter are deployed and executed on the BBoxDB nodes. A UDF is a class which implements the interface `UserDefinedFilter`, which is provided by BBoxDB. The interface contains two methods that need to be implemented by every UDF (see Listing 1).

Listing 1: The interface UserDefinedFilter.

```
1 public interface UserDefinedFilter {
2
3  public boolean filterTuple(Tuple tuple, byte[] customData);
4
5  public boolean filterJoinCandidate(Tuple tuple1, Tuple tuple2, byte[]
       customData);
6 }
```

- The method `filterTuple` in Line 3 is used to refine range queries: $f(t) \in \{true, false\}$. For each tuple that has a bounding box which intersects with the query rectangle the method is called.
- The method `filterJoinCandidate` in Line 5 is used to refine spatial join queries: $f(t1, t2) \in \{true, false\}$. The method is called with two join candidates that have intersecting bounding boxes.

When these methods return `true`, the tuple is part of the final query result. Otherwise, the tuple is not part of the final result. In addition, both methods accept a user defined value `customData`,

which can be used for further operations (e.g., test a property like the name of the road in the GeoJSON encoded value).

3.6.2 A UDF for GeoJSON Encoded Data

In this paper, many examples work with GeoJSON encoded data. In this section, a UDF is discussed, which decodes GeoJSON encoded data. The UDF can be used: (1) to refine range queries and (2) to refine bounding box-based spatial joins to spatial joins on the real geometries.

The range query refinement is done by performing an intersection test between the real geometry of a tuple and a provided geometry. The spatial join refinement is done by performing intersection tests on the real geometries of the elements. Elements that have only intersecting bounding boxes and no intersecting geometries (see Figure 7) are removed from the result. The *ESRI Geometry API for Java library* [69] is used by the `UserDefinedGeoJsonSpatialFilter` UDF to perform the intersection test. The UDF is used later in Section 5 to refine continuous queries. For a good understanding of the examples, this UDF is described in this section in greater detail.

The UDF performs the following filter tasks on range queries:

- When the `filterTuple` method is called with only a tuple, all tuples can pass the filter.
- When the `filterTuple` method is called with a tuple and a GeoJSON element as *custom value*, an intersection test is performed between these objects.

The UDF performs the following filter tasks on spatial join queries:

- When the `filterJoinCandidate` method is called with two tuples, the performed operation depends on the type of the GeoJSON geometries:
  - When both geometries are regions, an intersection test is performed.
  - Otherwise (e.g. for a point and a region), a distance test of the geometries is performed. Geometries that are closer than five meters are treated as intersecting and can pass the filter. The distance test for lines or points is implemented for situations where a position of a car (a point) should be compared with a road (a line). Due to measurement tolerances, the geometries do not really intersect but they become close (see Listing 14 in Section 5.3 for an example). The distance of 5 meters is the default value and can be changed by the user.
- When the `filterJoinCandidate` method is called with two tuples, and a custom value, the same calculation as described above is performed. In addition, it is assumed that the custom value is in the format `key:value`. GeoJSON elements can contain a property map of key-value pairs[10]. The filter tests that the provided key and value are contained in at least one of the property maps of the tuples. For example, with the custom value `name:road66` the name of the road is restricted to *road66*. By specifying `bridge:yes` the road has to be a bridge, and by specifying `lanes:4` the road has to have four lanes. Only tuples with a matching property and intersecting geometries can pass the filter.

In addition to the `UserDefinedGeoJsonSpatialFilter`, BBoxDB contains a more strict version of the filter called `UserDefinedGeoJsonSpatialStrictFilter`. This UDF performs only the real intersection test of both geometries; no distance check is performed. This filter can be used when it is required to test that a point is really inside of another geometry, like the position of a car (a point) in a forest (a region).

It is a common pattern in spatial join algorithms to evaluate the bounding box of an object in the first step and evaluate the full geometry only if required. For example, the calculation of a spatial join is often divided into a *filter step* and a *refinement step* [63]. The filter step is cheap to

[10] In the examples of this paper, we work on data fetched from the OpenStreetMap project. Details of the properties of these objects can be found in the wiki of the project [61].

calculate and detects all possible join candidates by intersecting bounding boxes. The refinement step is more expensive to calculate, works on the real geometries, and eliminates all join candidates that do not really intersect.

4 BBoxDB Streams

The introduction lists several challenges in the handling of multi-dimensional data streams (see Section 1.3). To solve these problems, we have developed a new data stream processing solution called BBoxDB Streams. The system is an extension of the key-bounding-box-value store BBoxDB. BBoxDB is used for data distribution and storage. Features such as the decoding of stream elements or the support for continuous queries are part of our BBoxDB Streams implementation.

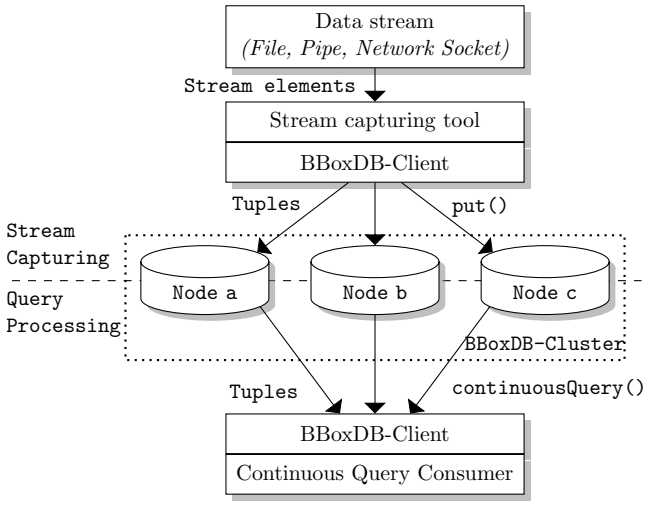


Figure 9: Handling a data stream with BBoxDB Streams. The data stream is captured, converted into tuples, and written to the nodes of the BBoxDB cluster. Afterward, the continuous queries are executed and the results are delivered to the clients.

The handling of data streams consists of two major tasks: (1) the data stream needs to be captured and handled, and (2) continuous queries need to be evaluated. Both tasks are fulfilled by our implementation and described in the next sections. The upper part of Figure 9 depicts the capturing of data streams (which will be described in Section 4.1). The middle part depicts the handling of multi-dimensional data in BBoxDB, which was already described in Section 3. The lower part in the image contains the execution of queries (see Section 4.2) and the visualization of results (see Section 4.8). The types of queries that can be executed is covered in Sections 4.3, 4.4, 4.5 and 4.7. Section 4.6 discusses strategies to distribute the stream elements to the nodes of the cluster.

4.1 Capturing Data Streams

To capture a data stream, the stream has to be: (1) read from an input source, (2) decoded and converted into a data format that can be handled by BBoxDB Streams, and (3) partitioned and distributed to the nodes that are responsible for the tuples.

4.1.1 Decoding and Processing Streams

BBoxDB Streams contains a *stream capturing tool*. The stream capturing tool allows the user to read data streams from input sources like *network sockets*, *named pipes*, or *files*. The goal of the stream capturing tool is (1) to read data from an input source (*input source* $\rightarrow$ *string*), and (2) to decode the content and generate a stream of BBoxDB compatible tuples (*string* $\rightarrow$ *tuple*). To decode the data of the input sources, BBoxDB Streams contains decoder for *GeoJSON*, *ADS-B*, or *GTFS*

real-time. Therefore, many streams can be handled out of the box. Support for additional data formats or stream inputs can be easily implemented by creating a new Java class and implementing the interface `TupleBuilder`.

The stream capturing tool uses the BBoxDB client library for the communication with the BBoxDB cluster. The client library is written in the programming language Java. This library honors the global index and redirects operations in a proper way when changes in the global index occur (see [56, pp. 30 ff.]). After the capturing tool is started, the following steps are executed:

- **Read Data:** The data stream is read continuously from the input source.
- **Parse Data:** The read data are parsed, and the stream elements are decoded and converted into tuples, consisting of a key, a bounding box, a value, and a version (see Section 3.1).
- **Ordering:** A version for the tuple is determined. When the stream elements contain a version field, this value is used. Otherwise, the current timestamp is used as a version. Taking the original version honors the order of the tuple at the creation time. By generating a new version, the receiving order of the tuples is taken.
- **Distribution:** The tuple is distributed to the nodes of the BBoxDB cluster. To perform this action, the capturing tool executes the `put()` operation together with a table name. Afterward, the tuple is sent to the nodes which are responsible for storing data for the region in space which is described by the bounding box of the tuple. Due to the fact that a table name is associated to the stream elements in this step, continuous queries can be registered on these stream elements by specifying the table name.
- **Continuous Queries:** When a node receives a new tuple, the tuple is processed and the execution of the registered continuous queries is performed (see Figure 9 for an illustration).
- **Store:** After the tuple is processed, the tuple can be stored on disk or discarded. The desired action for the tuples can be configured by the user of the system in the stream capturing tool [11].

4.1.2 Data Storage and Index Updates

The capabilities of BBoxDB are used to store the elements of the data stream. How the storage structures work and the two-level index structure (see Section 3.4) is affected by new data is described in this section.

Local Index: *Memtables* and *String Sorted Tables* (SSTables) [18] are employed as data structures to store data; both data structures are optimized for writes. Memtables are located in memory, and SSTables are located on disk. Tuples are stored in a Memtable first. When a threshold is reached, the tuples are sorted by key and written to disk as an SSTable. Tuples are never updated; instead, new versions of a tuple are stored. Old versions of the tuples are removed periodically in a cleanup task called *compactification*.

SSTables are optimized for key-based retrieval operations. For efficient hyperrectangle-based retrieval operations (e.g., range queries), an R-Tree is created for each Memtable and SSTable (the local index).

Global Index: The global index determines which node is responsible for which distribution region (a partition of the space). Updating the global index is an expensive operation because the data between the nodes need to be re-distributed and transferred between the nodes. Therefore, the global index is only updated when a partition becomes unbalanced. Two threshold values (the upper and the lower limit) are defined when a partition is considered as unbalanced. These values

[11] As in a KVS, tuples are overwritten when a new tuple with the same key is stored. In addition, BBoxDB supports the automatic deletion of tuples after a particular time (*time to live-based tuple removal*). This feature can be used to ensure that the persistent data of the stream elements is automatically removed after some time when the element is no longer contained in the stream.

are configured when the distribution group is created. In addition, it can be configured if tuples or the amount of the stored data is used as the size of the region. The global index also determines how the data stream is partitioned and distributed in the BBoxDB cluster. Therefore, when the global index is changed, the stream is distributed in a different manner. The global index is split in a way that the data is evenly distributed across the cluster. Therefore, the data stream is also distributed in a way that reflects the distribution of the stream elements in the n-dimensional space.

The data re-distribution was already implemented in BBoxDB. Also, the handling of read- and write-operations during the data re-distribution was implemented. These implementations are re-used by BBoxDB Streams. More about the storage management and the data re-distribution of BBoxDB can be found in [56].

4.2 Performing Continuous Queries

BBoxDB Streams enhances BBoxDB by two operations for the handling of continuous queries: (1) `continuousQuery(queryPlan)` and (2) `cancelQuery(id)` (see Table 3). The first operation registers a new query while the second operation cancels a previously registered continuous query.

Operations of BBoxDB Streams		
`continuousQuery`	queryPlan	$\rightarrow$ id $\times$ stream(tuple)
`cancelQuery`	id	$\rightarrow$ bool

Table 3: The operations implemented for BBoxDB Streams.

The operation `continuousQuery` takes a query plan as parameter. This query plan describes how the tuples of the stream are processed and which result tuples are returned by the continuous query. How the execution of the query plan is done is described in Section 4.4. How a continuous query plan can be constructed (e.g., which transformations are performed and the specification of the area where the query is registered) is described in Section 5. The operation returns an id of the continuous query together with a stream of result tuples. The query id can be used as a parameter for the operation `cancelQuery` to stop the execution of the continuous query. Listing 5 of Section 5.1 demonstrates the usage of these operations.

These operations are fully integrated in the BBoxDB client and also track the state of the global index (the data distribution). When a continuous query is registered and the data distribution is changed (e.g., a distribution region is split or merged), the query is automatically re-registered on the new distribution regions.

4.3 Tuple Transformations

Tuple transformations are an important part of the evaluation of continuous queries. BBoxDB Streams supports two types of transformations: (1) *Modifying Transformations* and (2) *Filter Transformations*.

4.3.1 Modifying Transformations

Modifying transformations allow the user to change the tuples of the stream (see Section 4.4); they are applied on the bounding box b of a tuple and return a modified bounding box b'. Depending on the transformation, addition parameter can be passed to the transformation: $f(b[r_1, \ldots, r_{2n}], \ldots) = b'[r'_1, \ldots, r'_{2n}]$. f is the modifying transformation, b is the original bounding box, b' is the transformed bounding box, $[r_1, \ldots, r_{2n}]$ and $[r'_1, \ldots, r'_{2n}]$ are the values of the n-dimensional bounding boxes (see Section 3.3).

The following modifying transformations are supported in BBoxDB:

Enlarge Bounding Box by Factor: This transformation enlarges the bounding box by a constant factor c. In each dimension the extension is calculated and multiplied by c. Half of the

enlargement is subtracted from the start coordinate of the bounding box and the other half is added to the end coordinate. So, the bounding box is enlarged and location of the center remains unchanged: $f(b[r_1, r_2, \ldots, r_{2n}], c) = b'[r_1 - \frac{c(r_2 - r_1)}{2}, r_2 + \frac{c(r_2 - r_1)}{2}, \ldots, r_{2n} + \frac{c(r_{2n} - r_{2n-1})}{2}]$.

Enlarge Bounding Box by Value: The bounding box b is enlarged by a constant value v in each dimension. As in the factor transformation, half of the value is subtracted from the start coordinate and half the value is added to the end value in each dimension: $f(b[r_1, r_2, \ldots, r_{2n}], v) = b'[r_1 - \frac{v}{2}, r_2 + \frac{v}{2}, \ldots, r_{2n} + \frac{v}{2}]$.

Enlarge WGS84 Bounding Box by Meter: Enlarge the two-dimensional bounding box b that uses WGS84 coordinates by a certain value v in meters. The extension on the latitude axis is specified as e_{lat}; the longitudinal extension is specified as e_{lon}. These meter based values are converted into the proper changes of the WGS84 coordinates e'_{lat} and e'_{lon}. The bounding box is changed as follows: $f(b[r_1, r_2, r_3, r_4], v) = b'[r_1 - \frac{e'_{lat}}{2}, r_2 + \frac{e'_{lat}}{2}, r_3 - \frac{e'_{lon}}{2}, r_4 + \frac{e'_{lon}}{2}]$.

4.3.2 Filter Transformations

Filter transformations decide whether a tuple t can pass the filter or not: $f(t, \ldots) \rightarrow \{true, false\}$. Depending on the filter, additional parameters (e.g., name of a key to filter) can be passed to the filter function. Filter transformations can be applied on the stream and on the previously stored tuples.

When the filter returns `true`, the tuple can pass the filter; otherwise, the query processing for this tuple is stopped and the tuple will not be part of the query result. The following filter transformations are supported in BBoxDB:

Filter by Key: If the key of a tuple is equal to a certain value v, `true` is returned; `false` otherwise: $f(t, v) \in \{true, false\}$.

Filter by Bounding Box: If the bounding box of the tuple does intersect with a constant bounding box b, the filter function returns `true`; `false` otherwise: $f(t, b) \in \{true, false\}$.

Filter by User-Defined Filter: This is a generic filter operation which delegates the real filter operation to the user-defined filter u with the user-defined value v (see Section 3.6) and returns the result of the UDF: $f(t, u, v) \in \{true, false\}$.

4.4 Continuous Queries

BBoxDB Streams supports two types of queries: (1) *continuous range queries* and (2) *continuous spatial join queries*. These queries are discussed in the following subsections.

4.4.1 Continuous Range Queries

The continuous range query q_{cr} is used to filter stream elements that intersect with a given query rectangle. The stream elements can also be transformed and filtered (see Section 4.3). The query performs a selection on the n-dimensional data stream S_n and returns all stream elements that intersect with the range α where the query is registered and matches the selection function σ_{cr}:

$$q_{cr}(S_n, \alpha, \sigma_{cr}) = \{s \mid s \in S_n \ \wedge \ s \cap \alpha \neq \emptyset \\ \wedge \ \sigma_{cr}(s, \tau, \theta, \beta)\}$$

Table 4 describes the notations of the equation. Section 5 discusses the elements of the query in greater detail and gives some examples.

After a tuple s of a data stream is received by a BBoxDB node, the continuous queries are executed. When the bounding box of the tuple intersects with the area in space α where the continuous query is registered, the following steps are executed:

Symbol	Description
S_n	The n-dimensional data stream.
s	A tuple of the data stream S_n.
α	The hyperrectangle in space in which the query is registered.
τ	The query hyperrectangle.
θ	A set of transformations that are applied to the stream tuples.
β	Is a boolean value which determines whether the query reports positive or negative matches.

Table 4: The notations of the continuous range query.

1. The transformations θ for the bounding box of the tuple s are executed.
2. After the transformations are applied and no filter has stopped the execution of the query (see Section 4.3.2), the bounding box of the tuple is compared with the query hyperrectangle τ.
3. If the bounding box of the tuple and the query hyperrectangle do intersect and positive matches should be reported β, the tuple is sent to the query client. The same is performed if the bounding boxes do not intersect, and negative matches should be reported.

Example. With this type of query, the ships of a static region of the ocean can be observed. The position data of the ships is written to a table; on this table, the continuous range query is registered. The region where the query is registered α and the query rectangle τ are identical. β is set to *true* so that all ships that are intersecting with the query rectangle are reported. So, all ships that enter the region are detected by the query. Transformations on the stream elements θ are not performed in this example. □

Using the continuous range query also more complex queries can be executed. This is shown in the following example.

Example. Again, all ships that are heading to an island should be reported. But in addition, ships that are in the direct neighborhood of the island should be ignored by the query. For instance, these are ships waiting for a free port in the harbor, or these are ships that are maneuvering in the harbor area. The situation is depicted in Figure 10. The continuous query should report all ships that are in the light grey area; *Ship1* should be detected and *Ship 2* should be ignored.

Again the data stream containing the position of the ships is written to a table. On this table, the query is registered in the area α. This is the part of the ocean that is observed, the area around the island is τ, β is set to *false*. Therefore, only ships are reported by the query that are inside of α but not intersecting with τ. The query reports the ships that are inside of the region $\alpha \setminus \tau$. □

4.4.2 Continuous Spatial Join Queries

The continuous spatial join query q_{cj} is used to compare stream elements with previously stored multi-dimensional data. Like the continuous range query, the continuous spatial join query can filter and transform the elements of the stream. The query performs a spatial join between the n-dimensional data stream S_n and the table R_n of the same dimensionality in the area α using the selection function σ_{cj}:

$$\begin{aligned} q_{cj}(S_n, R_n, \alpha, \sigma_{cj}) = \{(r,s) \mid\ & r \in R_n \ \wedge\ s \in S_n \\ & \wedge\ s \cap \alpha \neq \emptyset \\ & \wedge\ \sigma_{cj}(r, s, \theta, \lambda)\} \end{aligned}$$

Table 5 describes the notations of the equation. Section 5 discusses the elements of the query in greater detail and gives some examples.

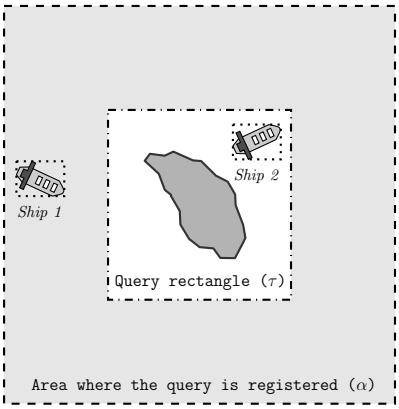


Figure 10: A continuous range query that determines all ships that are heading to an island. Ships in the direct neighborhood of the island are ignored.

Symbol	Description
S_n	The n-dimensional data stream.
R_n	The n-dimensional table of stored tuples.
s	A tuple of the data stream S_n.
r	A tuple of the table R_n.
α	The hyperrectangle in space in which the query is registered.
θ	A set of transformations that are applied to the stream tuples.
λ	A set of transformations that are applied to the previously stored tuples. Only filter transformations are supported.

Table 5: The notations of the continuous spatial join query.

For the previously stored tuples, only filter transformations are supported; applying modifying transformations is not supported. Otherwise, the stored tuples would need to be accessed to calculate the enlargement of the resulting bounding boxes (e.g., multiply the length the bounding box by a value of 2). Since it is not known in advance how the transformation will change the bounding box and whether the resulting bounding box is relevant for the query afterward, the transformation has to be applied to all stored tuples. The calculation would be very inefficient because all already-stored tuples have to be loaded. Therefore, modifying transformations on previously stored tuples are not implemented in BBoxDB Streams; modifying transformations can only be applied to the stream elements.

Like the continuous range query, the continuous spatial join query is executed every time a stream tuple s is received by a BBoxDB node. If the bounding box of the tuple and the range where the query is registered α do intersect, the query is executed. The following steps are executed:

1. The transformations θ on the bounding box of the stream tuple are applied.
2. Afterward, the transformed bounding box is used for a range query on the table R_n. Details of this operation are discussed in Section 4.6.
3. For each result tuple t of the range query, the persistent tuple filter transformations λ are applied.
4. If the tuple t is not eliminated by a filter operation the intersection between the bounding box of t and the bounding box of the stream tuple s is calculated.

5. If the bounding boxes do intersect, both tuples are sent to the query client (see Section 5.1 for a detailed example). Due to the transformations λ, the bounding boxes might no longer be intersecting.

Example. With this type of query, the problem that is depicted in Figure 2 on Page 3 can be solved. In the figure, collisions between ships and the reefs of the ocean should be detected before they occur. The stream S_n contains the position of the ships and the table R_n contains the spatial data of the reefs. The observed region of the ocean is α. The transformation of the stream elements θ is used to enlarge the bounding box of the ships to detect a possible collision before it occurs. Transformations on the persistently stored data λ are not used in this example. □

4.5 Distributing Stream Elements

A challenge in the scalable handling of data streams is the efficient distribution of the stream elements to the nodes of the cluster (see the *stream capturing* part in Figure 9 on Page 16). To execute efficient continuous spatial join queries, the stream elements need to be spread to the nodes that store the join partners. Listing 2 shows in pseudocode how the stream elements are distributed in BBoxDB Streams.

Listing 2: Spreading stream elements to the nodes.

```
1 for(e ∈ stream) {
2     b = bounding box of e
3     r = distribution regions that are intersecting with b
4     n = nodes that are storing r
5     forward e to the nodes n
6 }
```

According to the global index, the stream elements are distributed to all nodes responsible for the area of the bounding box of the tuple. This means that the stream elements are partitioned in the same way as the tables of the distribution group. This leads to the same partitioning of the stream elements and the stored data of this distribution group; the tables and the stream elements are co-partitioned.

Technically, the distribution is implemented as follows:

1. The stream is converted into tuples and stored in a table in BBoxDB by calling the `put()` operation.
2. The BBoxDB client queries the global index of the distribution group with the bounding box of the tuple. The tuple needs to be sent to all distribution groups, which intersect with the bounding box of the tuple (Line 2 and 3 in Listing 2).
3. The nodes that are responsible for these distribution regions are determined. The BBoxDB client contains a robust implementation that handles changes in the global index (i.e., splits or merges of distribution regions, or failed and new started nodes) automatically (Line 4).
4. The tuple is sent to all of these nodes that process the tuple (Line 5).
5. After the tuple is sent to the required nodes, the registered continuous queries are executed (see Section 4.4).

The architecture of BBoxDB Streams is highly scalable. The basic algorithms for the scalability are already implemented in BBoxDB. BBoxDB re-partitions the space automatically in the background if the data are unevenly distributed. This is performed without interrupting read or write access to the data (see [56] for a more detailed discussion of this functionality). BBoxDB Streams re-use these algorithms for the distribution of the stream elements. BBoxDB Streams automatically

adapts changes of the global index and registers already started continuous queries on newly created (split or merged) distribution regions.

When the stream contains an area in space in which many stream elements are located, the nodes that are responsible for this area have to do more work than other nodes. The nodes have to process a large number of stream elements in the continuous queries, and they have to store the stream elements on disk. Storing the stream elements on disk leads to growing distribution regions. BBoxDB recognizes these regions and they are split automatically after some time (see Section 3.4). The split of a distribution region leads to a changed distribution of the stream. The dense area is now distributed and handled by more nodes.

4.6 Data in Different Distribution Regions

To perform continuous spatial joins between a data stream and previously stored data, the stream elements and the previously stored data need to be co-partitioned. This means that the stream elements are spread to the nodes that store the possible join partners (see Sections 3 and 4.5). Transformations (e.g., enlargements of the bounding box) can be applied to the bounding boxes of the stream elements (see Section 4.3). Enlargement transformations are executed on the BBoxDB node that executes the continuous query. The transformation of the bounding box can lead to the situation where the enlarged bounding box intersects with another distribution region. In this case, the stream element and the join partner are located on different nodes. This behavior is shown in Figure 11.

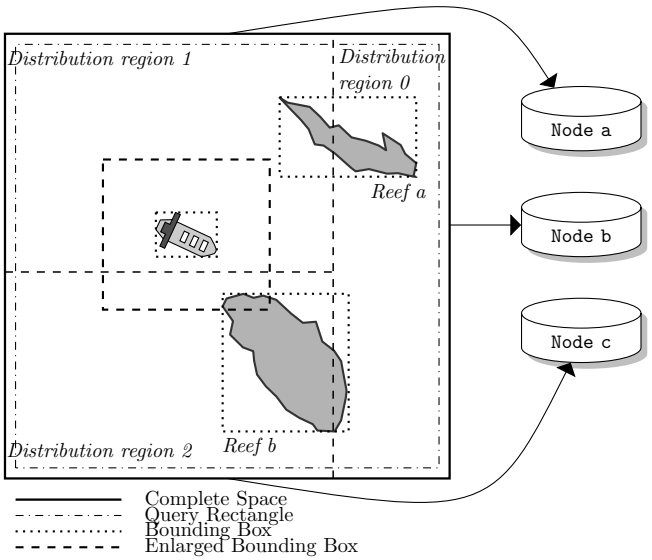


Figure 11: A stream of position data of ships is joined with the reefs of the ocean. The position data of the ship belongs to *Distribution region 1* and is distributed to *Node a*. The spatial data of the *Reef b* belongs to *Distribution region 0* and *Distribution region 2* and is stored on *Node b* and *Node c*.

In the figure, the spatial data of the ocean are stored in a cluster of BBoxDB nodes. A continuous spatial join covering almost the entire space is used to find all ships that are about to hit a reef. To get notified about this before the ship has actually hit the reef, the bounding box of the ship is enlarged by a constant value.

As described in Section 4.1, the stream elements are distributed to the nodes that are responsible for the region in space. In the figure, the position of the ship belongs to the *Distribution region 1* which is stored on *Node a*. The continuous query for the stream element is performed on this node, and the bounding box of the ship is enlarged. With the enlarged bounding box, a spatial join is performed. However, the enlarged bounding box also belongs to the *Distribution region 2*. In this

region, the join partner *Reef b* is located. If the stream element is only joined with the local data on *Node a*, the intersection between the bounding boxes of the ship and *Reef b* will not be detected.

In general, when a transformation enlarges a bounding box of a stream element, the enlarged bounding box can intersect with additional distribution regions. The join partners of these regions need to be included in the join to calculate the correct result. The strategies of the following subsections are implemented in BBoxDB to solve the problem.

4.6.1 Fetch Data From Nodes – FETCH

This strategy fetches the missing join partners from other nodes via the network. When the bounding box of a stream element intersects with another distribution region, all data stored in this area are fetched from the nodes. Listing 3 shows the algorithm of this strategy. The advantage of this strategy is that no special distribution of the stream elements is required. The drawback is that data have to be transferred through the network when the continuous spatial join query is evaluated. Transferring data through a network has high latency, and the network bandwidth can become a bottleneck.

Listing 3: Fetching the missing join partners via the network.

```
1  localRegion = the region of the space that is stored locally
2
3  for(e ∈ stream) {
4    for(q ∈ queries) {
5      bbox = apply q.θ to the bbox of e
6      nonLocalData = bbox \ localRegion
7      joinPartners = fetchDataOfRegion(nonLocalData)
8    }
9  }
```

Example. Figure 12 depicts the situation, the light grey area is fetched via the network by *Node a* from *Node c*. Therefore, the spatial data of *Reef b* is transferred to *Node a* and the intersection between the reef and the enlarged bounding box is detected. □

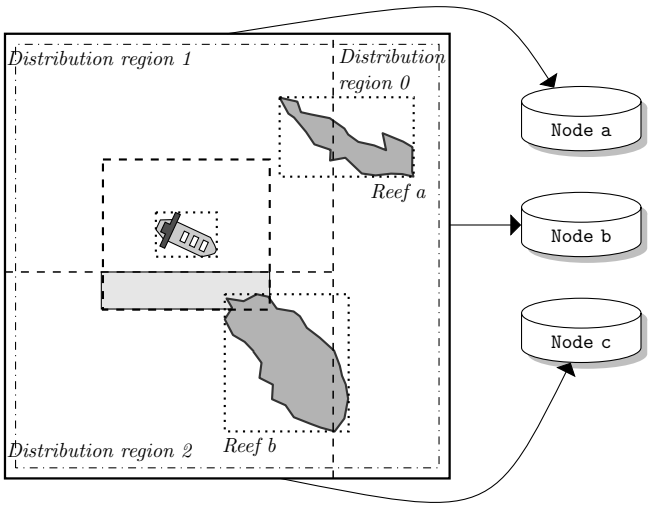


Figure 12: By using the *FETCH* strategy, the data of the light grey area is fetched via the network during the spatial join.

4.6.2 Enlarge By Static Padding – STATIC

In the moment, when a stream element is processed by the stream capturing tool (see Figure 9), the bounding box that is used for the distribution is enlarged by a static padding. The enlargement of the bounding box ensures that the tuple is distributed to all required nodes. On these nodes, the execution of the continuous query is also performed and the join with the locally stored data is executed. The static padding has to be calculated and specified by the user. The drawback of this strategy is that continuous queries that perform tuple transformations that are larger than this enlargement still lead to incorrect results.

Example. When the stream element of the ship is distributed in Figure 11, the bounding box of the ship is enlarged by the same padding as used by the transformation in the continuous query. Therefore, the ship is distributed to *Node a* and *Node c* and the possible collision is detected by the continuous join. □

4.6.3 Enlarge By Dynamic Padding – DYNAMIC

The strategy is similar to the *STATIC* strategy. The main difference is that the enlargement of the bounding boxes is automatically determined. The enlargement is determined by calculating the largest enlargement of all currently registered continuous queries. When a user registers a continuous query that uses a bigger enlargement than the already registered queries, the stream capturing tool automatically uses the biggest enlargement. The strategy is illustrated in Figure 13.

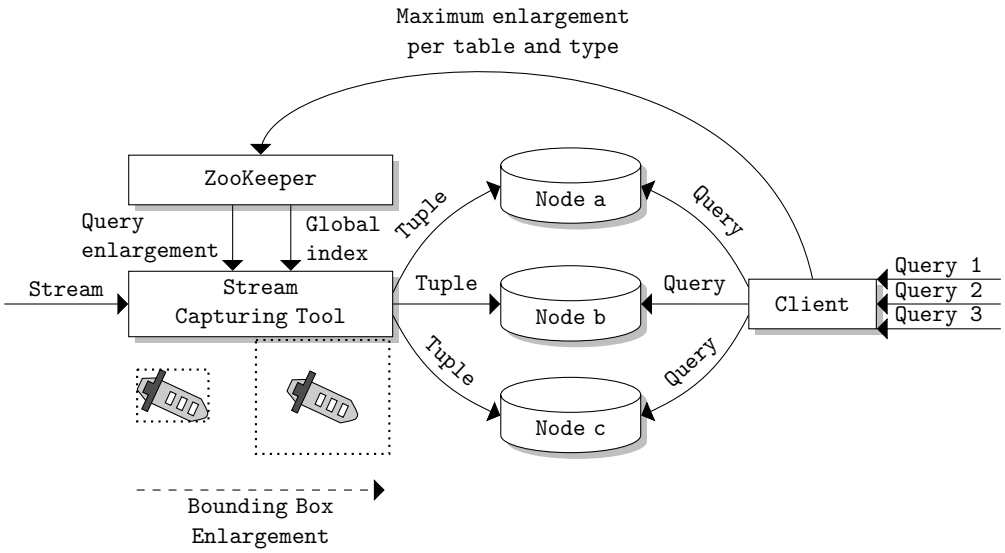


Figure 13: To distribute the stream elements to the proper nodes, the bounding boxes are enlarged in the stream capturing tool, and the distribution is performed based on the enlarged bounding boxes. To determine the needed enlargement, each BBoxDB client registers the maximum enlargement of all of his queries per table and enlargement type in ZooKeeper. The maximum enlargement is used by the stream capturing tool to enlarge the bounding box.

Each BBoxDB client tracks its continuous queries. When a new continuous query is executed, the BBoxDB client determines the maximum enlargement of all of his continuous queries per table and per modifying transformation type (see Section 4.3.1). These enlargement values are stored in ZooKeeper for every running BBoxDB client (see Section 3).

As soon as the continuous query is canceled, the maximum enlargement of all remaining queries is recalculated and updated properly in ZooKeeper. In ZooKeeper, the enlargements are stored as *ephemeral nodes* [40, p. 3]. This means that this node is automatically deleted as soon as the BBoxDB client disconnects or crashes. Therefore, outdated enlargements are automatically removed.

The stream capturing tool takes the maximum of all stored enlargement values per transformation type from ZooKeeper and uses these values to enlarge the bounding boxes of the stream elements. Due to the *watcher* functionality of ZooKeeper [40, p. 3], the stream capturing tool gets notified about changes automatically and adapts the maximum enlargement.

Listing 4 contains the algorithm of the dynamic padding in pseudocode. The maximum enlargement of the supported modifying transformations is applied, and the bounding box of these bounding boxes is determined[12]. The resulting bounding box is the maximum enlargement of the bounding box of the tuple, after all transformations of the currently registered continuous queries are applied. The tuple is distributed according to this bounding box.

Listing 4: Spreading stream elements to the nodes using dynamic padding.

```
1  for(e ∈ stream) {
2      b = BBox.of(e)
3
4      eb1 = enlargeByMaxFactorPadding(b)
5      eb2 = enlargeByMaxValuePadding(b)
6      eb3 = enlargeByMaxMeterPadding(b)
7
8      enlargedBBox = BBox.of(eb1, eb2, eb3)
9
10     r = distribution regions that are intersecting with enlargedBBox
11     n = nodes that are storing r
12     forward e to the nodes n
13 }
```

The advantage of this strategy is that the stream elements are always distributed with the correct padding. The drawback is a more complex implementation and the necessity to determine the maximum enlargement of all registered continuous queries.

The advantages and the drawbacks of these strategies are discussed in the evaluation in Section 6.9.

4.7 Performing Queries in Multiple Regions

A bounding box of a stream element can intersect with multiple distribution regions. In this case, the element is replicated and sent to multiple BBoxDB nodes (see Section 4.6). Also, a query rectangle can intersect with multiple distribution regions. In this case the continuous query is registered on several nodes (see Figure 16).

The duplication of the stream elements and the execution of the query on multiple nodes in parallel could lead to the situation that a tuple is contained multiple times in a query result, which is incorrect. To prevent this, a hash map in the BBoxDB client is used to detect and filter duplicates (see the lower part of Figure 9). The hash map is built over the keys and version timestamps of the query result tuples. When the client detects an already known tuple, the tuple is discarded before it is reported to the user.

4.8 The GUI

The GUI of BBoxDB provides information about the cluster, the data distribution, and allows one to perform queries. BBoxDB Streams enhances the GUI in a way such that continuous queries are supported.

The GUI is optimized to handle two-dimensional GeoJSON encoded data. On the GUI, a user can define a query rectangle and execute queries. BBoxDB Streams integrates the support for

[12]The distribution of the stream elements is performed based on a axis parallel n-dimensional hyperrectangle in BBoxDB. The union of the individual bounding boxes can create a more complex polygon. Therefore, the bounding box of these bounding boxes is calculated, which is the smallest rectangle which encloses all individual bounding boxes.

continuous queries into the GUI. With the enhancement, continuous range queries and continuous spatial join queries can be executed. The GUI executes the specified continuous query and processes the received result tuples. The geometries of the tuples are shown as an overlay over the map. In addition to the location, the stream elements contain further information. Placing the mouse cursor over an element opens a tooltip, containing all the additional information that is contained in the GeoJSON object (e.g., the *altitude* of an aircraft or the *trip id* of a bus). The area of the GUI under the map shows details about the used cluster (i.e., IP, software version, available disks, disk space, CPUs).

Real-world queries can be performed and observed using the GUI. The stream capturing tool of BBoxDB is capable to decode some real-world streams, such as ADS-B encoded data (i.e., aircraft position data) or GTFS real-time encoded data (e.g., public transport vehicles). Queries such as *Which aircraft is currently in the airspace over Berlin?* (Figure 14) or *Which public transport vehicle drives currently through a forest?* (see Figure 15 and Listing 11 on Page 33) can be performed.

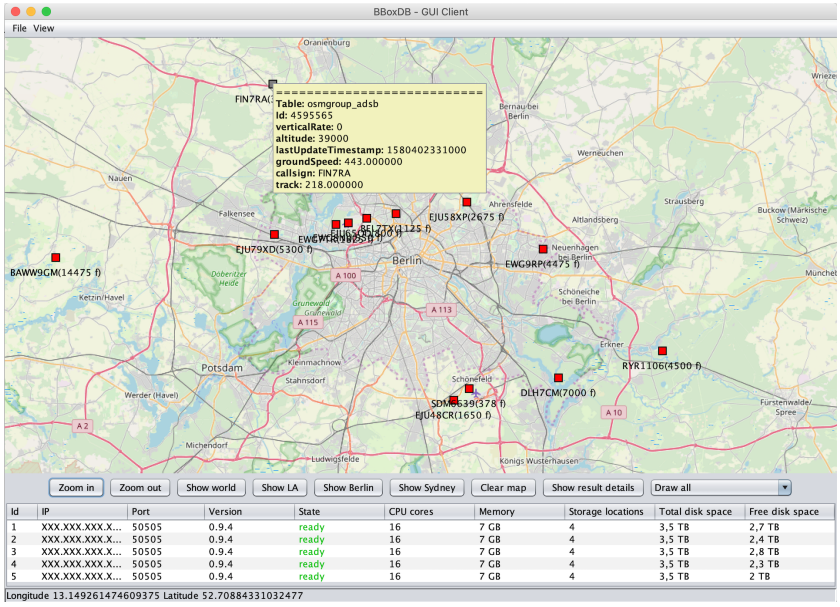


Figure 14: Visualizing aircraft traffic over Berlin, Germany in the BBoxDB GUI.

In addition queries, such as *Which buses are located on the Elizabeth Street in Sydney?* (see Listing 12 on Page 33) can be performed. More details about these data streams can be found in Section 6.1. Additional real-world use-cases of the GUI of BBoxDB Streams are described in the demo paper [57].

5 Continuous Queries

This section describes how continuous queries can be expressed and executed. Besides, some example queries are discussed. Before a continuous query can be executed, a query plan needs to be created. In BBoxDB Streams, the query plan can be specified directly in the programming language Java. A *query plan builder* ensures that the created query is syntactically correct.

After the query plan is built, it is serialized into JSON (see Figure 16) and submitted to the required BBoxDB nodes. As an alternative to the query plan builder, the query plan can be directly written in JSON (see Section 5.6). However, the query builder performs some basic checks, and we recommend using this class to create the query plans.

5.1 Execute Continuous Queries

To support continuous queries, the BBoxDB Java client was enhanced by two methods for the handling of continuous queries (see Section 4.2). These methods are used in Listing 5. In the listing,

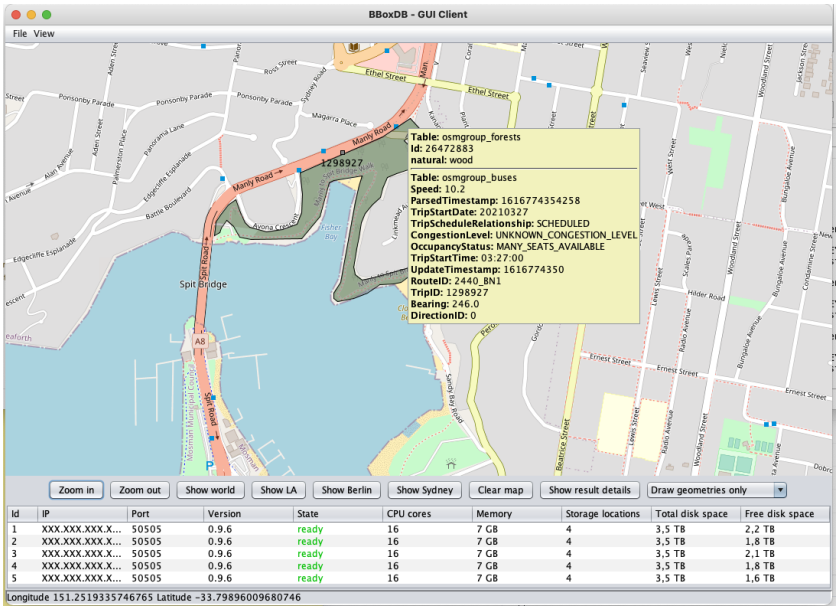


Figure 15: A continuous spatial join between the forests and public transport vehicles in Sydney, Australia.

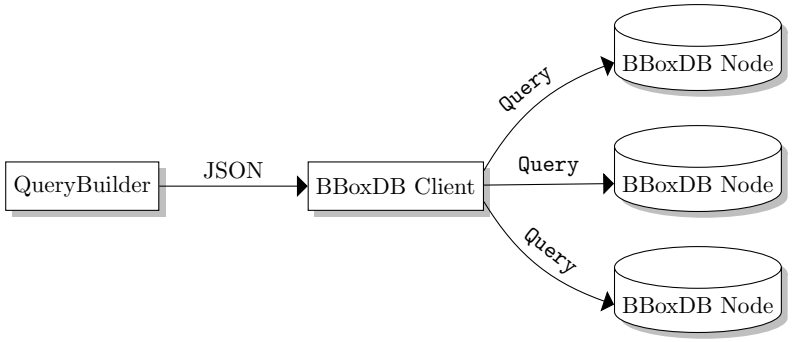


Figure 16: The query plan is built by the QueryBuilder, serialized into JSON, and registered on the required BBoxDB nodes.

the creation, registration, execution, and the cancellation of a continuous query is shown.

In Line 2, the query plan is created and serialized into JSON. To keep this example clear, the actual specification of the query plan is omitted; in the following listings, the calls to build a query are shown. In Section 5.4 the methods of the `QueryPlanBuilder` are described in detail.

The `QueryPlanBuilder` allows the creation of continuous queries by using the *builder pattern* [29, pp. 97ff.][13]. In Line 5, the created query plan is passed to the BBoxDB client and registered on all required nodes. A *future*[14] is returned that can be used to fetch the results of the query. In Line 8, the result tuples are fetched. When no unprocessed tuple is available, the iterator blocks and waits until the next tuple is available. When a result tuple is available, the tuple is assigned to the variable `tuple`. In Line 9, the tuple can be accessed and processed by the actual application code. In Line 13, the *query id* of the continuous query is determined, and in Line 14, the *query id* is used to cancel the running query.

The result type of a continuous query is a `MultiTuple` (see Line 8). A `MultiTuple` contains an ordered set of tuples. Depending on the query type, each `MultiTuple` consists of one or more tuples.

[13]The builder pattern is often used in complex object oriented software systems to provide a convenient way for creating objects that have complex constructors.

[14]BBoxDB uses the *future pattern* [11]; complex operations return a future instead of a concrete value. The calling application code can wait until the concrete result is computed 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 as much as possible.

Listing 5: Create and register a continuous query.

```
1  // Build query
2  ContinuousQueryPlan queryPlan = QueryPlanBuilder.[...].build();
3
4  // Register query
5  TupleListFuture queryFuture = bboxDBClient.continuousQuery( queryPlan);
6
7  // Handle query results
8  for(MultiTuple tuple : queryFuture) {
9    [...]
10 }
11
12 // Cancel query
13 UUID queryId = queryPlan.getQueryId();
14 bboxDBClient.cancelContinuousQuery(queryId);
```

For range queries, the `MultiTuple` contains only one tuple; this is the stream tuple that matches the query hyperrectangle. For spatial join queries, the `MultiTuple` contains two tuples: (1) the tuple of the stream and (2) the tuple from the static dataset. The `MultiTuple` abstraction can also be used by further operations to return more than two result tuples.

5.2 Building Continuous Range Queries

To build a continuous range query with the `QueryPlanBuilder`, the method `createQueryOnTable` has to be called with the table name on which the continuous query has to be registered (see Line 2 of Listing 6).

Listing 6: Building a continuous range query.

```
1 ContinuousQueryPlan qp = QueryPlanBuilder
2   .createQueryOnTable("dgroup_table")
3   .forAllNewTuplesInSpace(2.0, 3.0)
4   .enlargeStreamTupleBoundingBoxByFactor(2.0)
5   .compareWithStaticSpace(1.0, 4.0)
6   .build();
```

The area in space where the query is registered (symbol α in Table 4 on Page 20) is passed to the `forAllNewTuplesInSpace` method (Line 3). In this example, this is a one-dimensional hyperrectangle. Transformations of the stream elements θ are specified by methods like `enlargeStream TupleBoundingBoxByFactor` (Line 4). The method `compareWithStaticSpace` takes the query hyperrectangle τ as a parameter (Line 6). The query building is finished by calling the `build` method (Line 7) which returns the query plan serialized to JSON.

A continuous range query can report positive or negative matches β. A positive match is an intersection between the stream element and the query rectangle; otherwise, it is a negative match. The behavior of the query can be specified with the methods `reportNegativeMatches` and `reportPositiveMatches`. If none of these methods are called, positive matches are reported.

The `QueryPlanBuilder` implements the *fluid interface pattern* [26, pp. 343ff.]. All methods can be written *fluently* in a chain. The next method is called directly on the result of the last method (e.g., in Listing 6 the method `enlargeStreamTupleBoundingBoxByFactor` is directly called on the result of the method `forAllNewTuplesInSpace`).

5.3 Building Continuous Spatial Join Queries

Continuous spatial join queries can also be built with the `QueryPlanBuilder`. Building such queries is similar to the building of continuous range queries. Listing 7 contains an example of a continuous

spatial join query.

Listing 7: Building a continuous spatial join query.

```
1 ContinuousQueryPlan qp = QueryPlanBuilder
2   .createQueryOnTable("dgroup_table1")
3   .forAllNewTuplesInSpace(3.0, 4.0)
4   .spatialJoinWithTable("dgroup_table2")
5   .enlargeStoredTupleBoundingBoxByFactor(2.0)
6   .build();
```

The methods called in Lines 1–3 are equal to the last example. In contrast to the range query, the method `spatialJoinWithTable` is called (Line 4), which takes a table name as parameter. This is the table of the join partners (symbol R_n in Table 5 on Page 21). Every stream element is compared with the tuples of the table. In spatial join queries, transformations on the stored tuples λ can also be applied. This can be done by methods like `enlargeStoredTupleBoundingBoxByFactor` (Line 5). The building of the query is finished by calling the method `build` (Line 6).

5.4 The Query Plan Builder

This section describes all methods of the `QueryPlanBuilder`. This class is used to construct continuous query plans in BBoxDB Streams. The available methods are listed in Table 6 with a description. Calling a method multiple times overrides the older value. An exception are the methods `addStreamFilter(..)` and `addJoinFilter(..)` which can be called multiple times to add additional filters.

5.5 Query Examples

This section shows some query examples and the wide range of practical problems that can be solved with BBoxDB Streams.

5.5.1 Is the price of the stock "Apple" above € 100?

This is a query in the one-dimensional space. The price of a stock is treated as a point in space and the stream is written into the table `dgroup_stock` (see Listing 8). The query is registered on this table (Line 2). The query observes a price of the stock of the company *Apple* (Line 3) and produces query results as soon as the stock is traded for more than € 100 and less than € 9999 [15] (Lines 4 and 5).

Listing 8: Does a stock raise above a certain price?

```
1 ContinuousQueryPlan qp = QueryPlanBuilder
2   .createQueryOnTable("dgroup_stock")
3   .filterStreamTupleByKey("Apple")
4   .forAllNewTuplesInSpace(100.0, 9999.99)
5   .compareWithStaticSpace(100.0, 9999.99)
6   .build();
```

5.5.2 Do two vehicles drive side by side?

In this scenario, two cars are equipped with GPS receivers. They send their positions every few seconds to a BBoxDB cluster. The query in Listing 9 is used to test if the vehicles with the ids `vehicle1` and `vehicle2` are closer to each other for more than 10 meters.

[15] The upper limit of € 9999 is used because the bounding box needs to have an upper end, any large number can be used here (e.g., `Integer.MAX_INT`).

Method	Parameter	Description
`createQueryOnTable(..)`	String	This method specifies on which data stream the query is created (see S_n in Table 5 on Page 21).
`forAllNewTuplesInSpace(..)`	Hyperrectangle	The region in space where the query is registered (α).
`enlargeStreamTupleBoundingBoxByValue(..)`	Double	Enlarge bounding box of the stream tuple by a certain value (θ).
`enlargeStreamTupleBoundingBoxByFactor(..)`	Double	Enlarge bounding box of the stream tuple by a certain factor (θ).
`enlargeStreamTupleBoundingBoxByWGS84Meters(..)`	Double × Double	Enlarge the bounding box of the stream tuple by a certain amount of meter. The bounding box of the stream tuple has to be two-dimensional and WGS84 coordinates have to be used in the bounding box (θ).
`filterStreamTupleByKey(..)`	String	Filters the stream tuple by the given key (θ).
`filterStreamTupleByBoundingBox(..)`	Hyperrectangle	The stream tuple and the provided hyperrectangle have to intersect (θ).
`addStreamFilter(..)`	UDF × UDF-Value	The stream tuple has to pass the provided UDF (θ).
`build(..)`	-	This methods finishes the construction of the query plan and returns the constructed plan.
Continuous range query specific methods		
`compareWithStaticSpace(..)`	Hyperrectangle	The query rectangle (τ).
`reportPositiveMatches(..)`	-	Stream tuples that do intersect with the query rectangle should be reported (β).
`reportNegativeMatches(..)`	-	Stream tuples that do not intersect with the query rectangle should be reported (β).
Continuous spatial join query specific methods		
`spatialJoinWithTable(..)`	String	The name of the table of the join partners (R_n).
`filterStoredTupleByKey(..)`	String	Filters the join partner of the table by the given key (λ).
`filterStoredTupleByBoundingBox(..)`	Hyperrectangle	The join partner of the table and the provided hyperrectangle have to intersect (λ).
`addJoinFilter(..)`	UDF × UDF-Value	The stream tuple and the join partner have to pass the provided UDF (λ).

Table 6: The methods and parameters of the QueryPlanBuilder that can be used to create a query plan.

Listing 9: Do two vehicles drive side by side?

```
1  ContinuousQueryPlan qp = QueryPlanBuilder
2    .createQueryOnTable("dgroup_car")
3    .filterStreamTupleByKey("vehicle1")
4    .enlargeStreamBoundingBoxByWGS84Meter(10.0, 10.0)
5    .spatialJoinWithTable("dgroup_car")
6    .filterStoredTupleByKey("vehicle2")
7    .build();
```

For each stream element, it is tested that it contains a position update for *vehicle1* (Line 3). The two-dimensional bounding box of the vehicle is extended by 10 meters in both dimensions (Line 4). Then a spatial join is executed in this area (Line 5). The join partners are filtered by the key; only tuples with the key *vehicle2* (Line 6) are valid join partners. The query returns results as long as *vehicle1* and *vehicle2* are closer than 10 meters.

The spatial join is performed between the stream elements of `dgroup_car` and the elements of the table `dgroup_car` (Lines 2 and 5). This is a self-join between the current position of a car and the historical positions.

In the query, the current position of *vehicle1* is joined with the last known position of *vehicle2*. The query also works when the roles of *vehicle1* and *vehicle2* are swapped. In this case, the current position of *vehicle2* would be joined with the last known position of *vehicle1*.

5.5.3 Which ships are heading to an island?

This example is the same as shown in Figure 10 on Page 21. The ships heading to an island should be reported while the ships that are already near the island should be ignored by the query. The query is registered on a larger bounding box and negative matches for a given query rectangle are calculated. Listing 10 contains the calls required to construct such a query.

Listing 10: Which ships are heading to an island?

```
1  ContinuousQueryPlan qp = QueryPlanBuilder
2    .createQueryOnTable("dgroup_ships")
3    .forAllNewTuplesInSpace(1.0, 2.0, 1.0, 2.0)
4    .compareWithStaticSpace(1.2, 1.8, 1.2, 1.8)
5    .reportNegativeMatches()
6    .build();
```

In this example, the query is registered on the table *ships* (Line 2). In Line 3, the space where the query is registered is defined. The query rectangle is specified by calling the method `compareWithStaticSpace` in Line 4; this is the bounding box τ of the island. A slightly larger space α is passed to the method `forAllNewTuplesStoredInSpace`. All ships that are inside of this space but are not inside the bounding box of the island are reported by the query. These are the ships that are heading toward the island.

5.5.4 Which buses are driving through a forest in Sydney?

This example performs a spatial join between (1) a data stream, which contains the positions of several buses, and (2) an already stored dataset, which contains the spatial data of the forests of Australia. The spatial join is used to join the position of the bus with the forests. The result of the query are all buses that are currently inside of a forest. To ensure that the bus is really inside the polygon of the forest and not only inside the bounding box, a UDF is used to refine the query (see Figure 7 on Page 14). The refinement is done by the UDF, which is discussed in Section 3.6.2.

Listing 11 shows the building of the query plan. This query will also be used in the experiments on recorded bus trajectories and the spatial data of the roads of Sydney in Section 6.1.

Listing 11: Which buses are driving through a forest?

```
1  UserDefinedFilterDefinition udf
2   = new UserDefinedFilterDefinition(
3   "UserDefinedGeoJsonSpatialStrictFilter", "");
4
5  ContinuousQueryPlan qp = QueryPlanBuilder
6    .createQueryOnTable("dgroup_bus")
7    .forAllNewTuplesInSpace(
8      150.56, 151.34, -34.09, -33.60)
9    .spatialJoinWithTable("dgroup_forest")
10   .addJoinFilter(udf)
11   .build();
```

In Lines 1–3, a transformation with a user user-defined filter is created. The query is evaluated when new data are stored in the table `dgroup_bus` (Line 7). For every new tuple in the region of Sydney (Line 8) a spatial join with the table `dgroup_forest` is performed (Line 9). In Line 10, the UDF is referenced to refine the spatial join result.

5.5.5 Which buses are driving currently on the "Elizabeth Street" in Sydney?

This query is identical in many aspects to Listing 11. Again the positions of buses are joined with static data. This time, the spatial data of the roads in Australia are used. In contrast to the last query, it is not just a spatial join that is performed. In this query, one of the join partners has to contain a certain property value. As described in Section 3.6.1, the `customData` value of the `UserDefinedGeoJsonSpatialFilter` is used to check the property map for a certain value.

Listing 12: Which buses are driving on the Elizabeth Street?

```
1  UserDefinedFilterDefinition udf
2   = new UserDefinedFilterDefinition(
3   "UserDefinedGeoJsonSpatialFilter",
4   "name:Elizabeth Street");
5
6  ContinuousQueryPlan qp = QueryPlanBuilder
7    .createQueryOnTable("dgroup_bus")
8    .forAllNewTuplesInSpace(
9      150.56, 151.34, -34.09, -33.60)
10   .spatialJoinWithTable("dgroup_road")
11   .addJoinFilter(udf)
12   .build();
```

In Lines 1–4, the UDF is created and initialized. The name of the street and the associated key of the property map (`name:Elizabeth Street`) is passed as custom data to the UDF (see Section 3.6.2). In this example, the properties of both join candidates are tested to determine whether or not the key and the value are contained in the property map of one of the GeoJSON objects. If one of the join candidates matches, the intersection test on the real geometries is executed[16].

5.6 Serializing Queries to JSON

After a query plan has been built by the `QueryPlanBuilder`, it is serialized into JSON, and sent to the required BBoxDB nodes. The serialization is done to transform the query plan into a structure that can be sent through a network. To understand the structure of a JSON encoded query plan, two examples are discussed in this section.

[16]The position of the bus is a point in space, the geographical data of the road a line. As discussed in Section 3.6.2, a distance test is performed for these data types to refine the query. Join candidates that are closer than 5 meters are treated as intersecting.

5.6.1 Is the price of the stock "Apple" above € 100?

Listing 13 shows the JSON that is generated from the continuous range query of Listing 8 of Section 5.5.1.

Listing 13: Does a stock raise above a certain price?

```
1  {
2   "type":"bboxdb-query-plan"
3   "query-type":"range-query",
4   "query-range":"[[100.0,9999.99]]",
5   "stream-table":"dgroup_stock",
6   "compare-rectangle":"[[100.0,9999.99]]",
7   "report-positive":true,
8   "stream-transformations":[
9    {
10    "name":"key-filter",
11    "value":"Apple"
12   }
13  ],
14 }
```

In Line 2, the key `type` is set to the value `bboxdb-query-plan`. That value indicates that the JSON contains a BBoxDB query plan. Without this line, the BBoxDB nodes do not try to deserialize the query plan. In Line 3, the query type is defined; in this example, a continuous range query is executed. Line 4 defines a one-dimensional query rectangle (€100 – €9999); this is the area in space where the query is registered. Line 5 determines the table on which the query is registered. Line 6 defines the query rectangle, which is used to test whether or not a new tuple of the table qualifies for the continuous range query. Line 7 determines that the tuples are returned by the query, which do intersect with the query rectangle. Lines 8–11 contain the transformations of the stream elements. In this example, a filter on the key of the tuple is applied, which ensures that all result tuples must have a key that is identical to *Apple*.

5.6.2 Which buses are driving currently on the "Elizabeth Street" in Sydney?

Listing 14 shows the query plan JSON that is generated from the continuous spatial join with a user-defined filter of Listing 12 of Section 5.5.5.

In Line 3, it is defined that a continuous join query should be performed. Line 4 defines the range in space where the query is registered, and Line 5 defines on which table the query is registered. Line 6 defines the name of the data stream. Line 7 defines that the tuples of the table are not transformed. Lines 8–14 contain the definition of the UDF, while Line 15 determines that no transformations are applied to the stream tuple.

6 Evaluation

The evaluation of BBoxDB Streams is performed on a cluster of five nodes. These nodes 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 Java 8 on a 64 bit Ubuntu Linux. Unless stated otherwise, before the experiments are performed, the space is pre-partitioned (see Section 3.4) into 40 distribution regions to utilize all nodes of the cluster. Since not all distribution regions might be equally utilized, more distribution regions than nodes are created. In the used environment, each of the five nodes is responsible for eight regions, which leads to an almost equal utilization of the nodes (see Section 6.10 for a detailed discussion).

The pre-partitioning is performed with samples from the stored datasets to generate a good data distribution. Taking random samples is not possible when a data stream is processed sequentially.

Listing 14: Which buses are driving on the Elizabeth Street?

```
{
 "type":"bboxdb-query-plan"
 "query-type":"continuous-join",
 "query-range": "[[150.56,151.34]:[-34.09,-33.6]]",
 "join-table":"dgroup_forest",
 "stream-table":"dgroup_bus",
 "table-transformations":[],
 "join-filter":[
  {
   "filter-class":
    "UserDefinedGeoJsonSpatialFilter",
   "filter-value":"name:Elizabeth Street"
  }
 ],
 "stream-transformations":[],
}
```

How a data stream can be used to pre-partiton the space is discussed and evaluated in Section 6.10. The data streams are imported (unless stated otherwise in the experiment) by one instance of the stream capturing tool. This is done in the same way a real-world data stream would be read from a network socket, with the exception that the data is read from a disk.

In most of the experiments, the time to process a stored data stream is measured. These experiments show how a variation of some parameters (e.g., number of nodes, number of queries) affect the time to process the data stream entirely.

In a real-world scenario, stream elements have to be processed at the speed at which they are delivered. So, the design of the experiments differs from the real-world scenario. However, we have chosen to process a stored data stream and measure the processing time because: (1) the same stream elements are processed in all experiments, which makes it possible to compare the experiments. (2) The processing time is more meaningful than just saying that a certain amount of resources is not enough to fully process the stream.

Usually, BBoxDB Streams skips stream elements when not enough resources are available to process the stream at the required speed. For the following experiments, this behavior of the software was changed. All stream elements have to be processed. When not enough resources are available, the stream capturing tool has to wait for the processing of further stream elements. The change in the stream processing makes it possible to measure the processing time of the complete data stream.

6.1 Used Datasets

For the evaluation of BBoxDB Streams, three two- and three-dimensional stream datasets are used. Two stream datasets are captured from real-world data sources, and one stream dataset is synthetically generated. These datasets are described in detail in the following subsections. Besides, one static dataset is used for the evaluation of the spatial join between a data stream and previously stored data.

6.1.1 BerlinMOD Dataset

BerlinMOD [21] is a benchmark for spatio-temporal database management systems. The benchmark contains a data generator that generates trips of moving vehicles within the Berlin (Germany) metropolitan area. For the evaluation, the dataset was calculated with a scale factor of 5.0. The trips are stored as CSV data on disk, and WGS84 coordinates are used. The *moving object id* of the vehicle (`Moid`) is used as the key when the data is written to BBoxDB, and the two-dimensional position of the vehicle is used to calculate the bounding box of the elements.

6.1.2 NSW Transport Dataset

The government of the state of New South Wales in Australia operates the *NSW open data portal* [74]. On this portal, real-time data about buses, ferries, metros, and trains of the Sydney (Australia) metropolitan area are published. A GTFS encoded real-time feed of the data can be subscribed. The elements of the feed contain, among other things, an id, a position, the speed, and a tour number (see the tooltip in Figure 15). In this paper, we use the position data of the buses since they provide the majority of the elements in the stream. The stream content was captured for a whole week (21 January 2020 - 28 January 2020) and stored as GeoJSON. The id of the trip (`TripID`) is used as the key when the data is written to BBoxDB, and the two-dimensional position of the bus is used to determine the bounding box.

6.1.3 ADS-B Dataset

An aircraft continuously determines its position. The position is broadcasted periodically via radio as *automatic dependent surveillance – broadcast* (ADS-B) transmissions. In addition to the position, the transmissions contain the *altitude*, the *callsign*, the *heading*, and some more information. These ADS-B transmissions can be captured with an antenna on the ground. However, an ADS-B receiver captures only the transmissions in a radius of a few miles around the antenna. Websites such as `adsbhub.org` [73] provide a service to aggregate the feeds of several individual stations into a global feed.

For creating the ADS-B dataset, the global ADS-B data feed of the website adsbhub.org was captured for a whole day (16 February 2020) and the data are stored as CSV on disk.

ADS-B messages are typed. Depending on the message type, different information is contained in the message. Three messages of different types need to be read to construct one stream element. The *callsign* of the aircraft is used as the key when the data is written to BBoxDB. The two-dimensional position of the aircraft and the altitude are used to calculate the three-dimensional bounding boxes of the aircraft that are contained in this dataset.

6.1.4 Open Streetmap Dataset

This dataset is a copy of the planet dataset of the *Open Streetmap Project* (OSM) [60]. The dataset contains the spatial data of the whole world. In contrast to the last three datasets, this dataset consists of static data. The spatial data of the roads (denoted as *OSM roads*) and forests (denoted as *OSM forests*) are used for the evaluation of BBoxDB Streams.

For the experiments, the dataset was converted into GeoJSON[17]. The GeoJSON elements contain *properties* (key-value pairs) with additional information about the objects. For example, roads are annotated with the *maximum speed*, the *name*, and the *coating*. The *id* of each element is used as the key of the tuple when the data is written to BBoxDB. The two-dimensional bounding box of the geometry is used as bounding box of the tuple.

6.1.5 Summary

The described datasets are used in the evaluation of BBoxDB Streams. Table 7 summarizes the properties of these datasets.

The stream datasets are stored in different formats. The NSW transport dataset is stored as GeoJSON values, while the ADS-B dataset and the BerlinMod datasets are stored as CSV values. Different parsers are used to read the data. In addition, each dataset contains a different amount of additional information, which leads to different sizes per tuple. Table 8 contains an overview about the different element sizes.

[17] BBoxDB includes a converter which converts *OpenStreetMap Protocolbuffer Binary Format (.osm.pbf)* encoded data into GeoJSON (see [13] for more details about the converter).

Dataset	Type	Dim.	Covered Area	Total Elements	Different Elements	Size	Real Time [Seconds]
BerlinMod	Stream	2d	Berlin	259 841 059	4 473	23 GB	5 184 000
NSW transport	Stream	2d	Sydney	49 538 352	44 406	21 GB	604 800
ADS-B	Stream	3d	Planet	56 299 586	81 047	16 GB	86 400
OSM roads	Static	2d	Planet	146 060 493	146 060 493	67 GB	-
OSM forests	Static	2d	Planet	5 187 592	5 187 592	5.4 GB	-

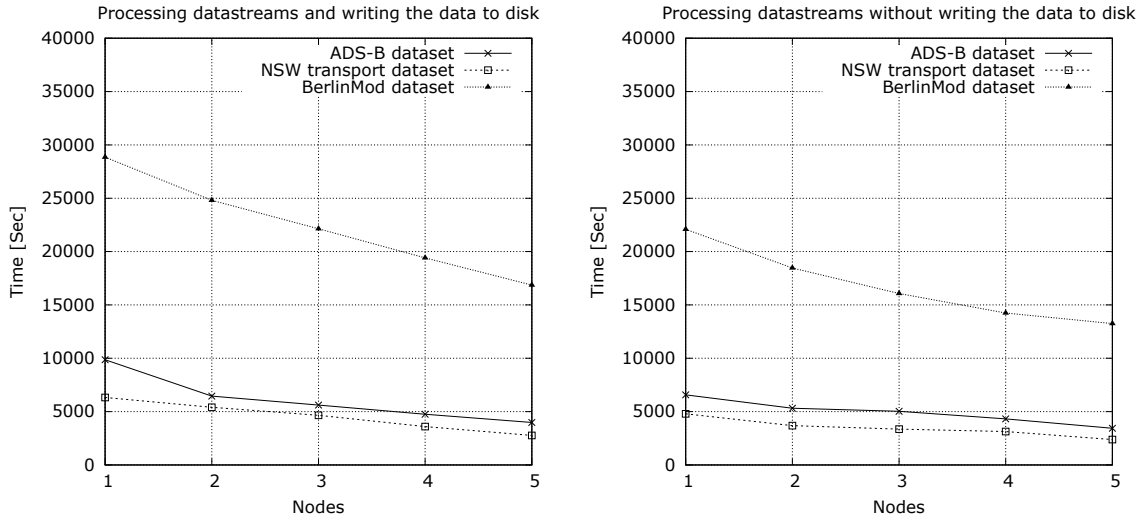
Table 7: The datasets that are used in the experiments. The column total elements denotes the absolute number of elements in the dataset; the number of different elements denotes how many different elements (e.g., cars or aircraft) are contained.

Dataset	Average Size (Byte)
BerlinMod	92
NSW transport	446
ADS-B (per message)	96
ADS-B (per tuple)	289

Table 8: The average size per element and stream dataset.

6.2 Processing Data Streams

In the first experiment, the data streams are imported into the cluster of BBoxDB nodes, and the time to perform the data import is measured. In addition, the number of BBoxDB nodes is varied between the experiments, and the data is imported: (1) one time with writing the data to disk and (2) one time without writing the data to disk. In the experiment, only the data stream is imported and processed by BBoxDB Streams. No continuous queries are registered during the stream processing. The evaluation of the continuous query will be part of the following experiments. The result of this experiment can be seen in Figure 17.



(a) Writing the data streams to disk.

(b) Without writing the data streams to disk.

Figure 17: Importing the data streams into a cluster of BBoxDB nodes. The number of nodes was varied during the experiments.

It can be seen in the figure that decreasing the number of nodes increases the needed time to import the data stream. The reason is that with fewer nodes, fewer resources for processing the stream are available. It can also be seen that writing the data to disk takes some time, and only processing the stream elements without writing the data to disk can speed up the processing.

The system utilization of the cluster was observed during the experiment. At any time, all nodes have some free disk, CPU, and memory resources. Therefore, only a small speed-up factor can be achieved in the experiment. The limiting resource is the parsing and distributing of the input stream. However, in this experiment, no continuous queries are registered. Therefore, the nodes do not have to evaluate such queries. This will be covered in the following experiments (see Section 6.8), and the data of this experiment shows the basic system behavior. Processing multiple data streams is discussed in the experiment of Section 6.3.

6.3 Importing Multiple Data Streams in Parallel

In the last experiments, only one data stream was imported. The workload to parse the data and to write the data to the BBoxDB cluster was only performed by one node. BBoxDB Streams is able to process multiple data streams in parallel. For the preparation of this experiment, the stored data streams are split into two to five pieces. The stream capturing tool is executed on different nodes, and the partial streams are imported in parallel. Therefore, the work to parse the data streams and to write the data to the network is parallelized. In the experiment, the data is written to the disks of the cluster, and no continuous queries are registered. As in the last experiment, the time to process the complete dataset is measured. The result of the experiment is shown in Figure 18.

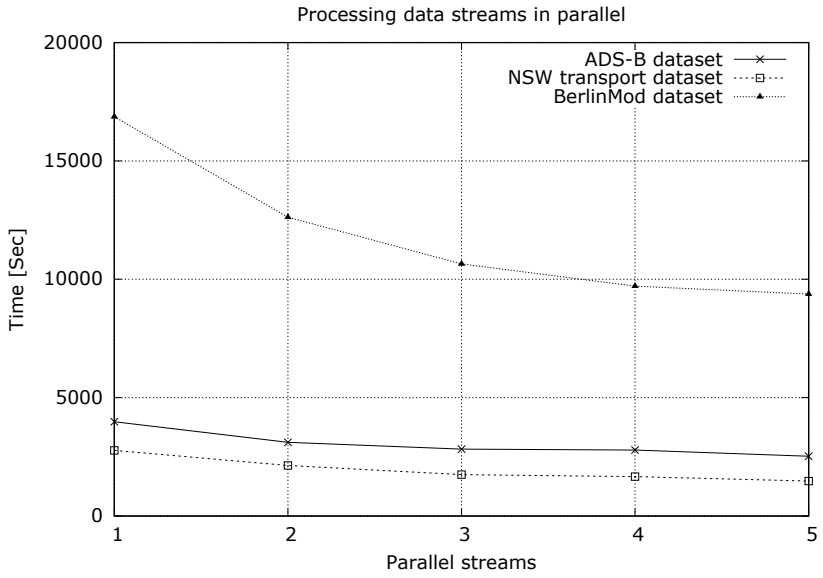


Figure 18: Importing data streams in parallel using multiple instances of the stream capturing tool.

It can be seen in the figure that the time to import the complete data streams decreases when more instances of the stream capturing tool are executed. Using more instances, the workload to parse the data and communicate with the BBoxDB cluster (e.g., determine to which nodes a tuple needs to be sent and serialize the tuple into a byte stream) is distributed between multiple nodes.

In addition, each BBoxDB node handles each connection (i.e., the connections from the stream capturing tool) in a separate thread. This thread reads the bytes from the network connection, parses the data, performs the requested operations (e.g., inserting a tuple), and writes the answers (e.g., acknowledging a successful operation) back to the client. Executing multiple instances of the stream capturing tool leads to more connections and a higher degree of parallelism on the BBoxDB nodes.

However, expensive tasks (e.g., the evaluation of the continuous queries or writing the data to disk) are performed in separate threads, and executing multiple instances of the stream capturing tool can not speed up these operations.

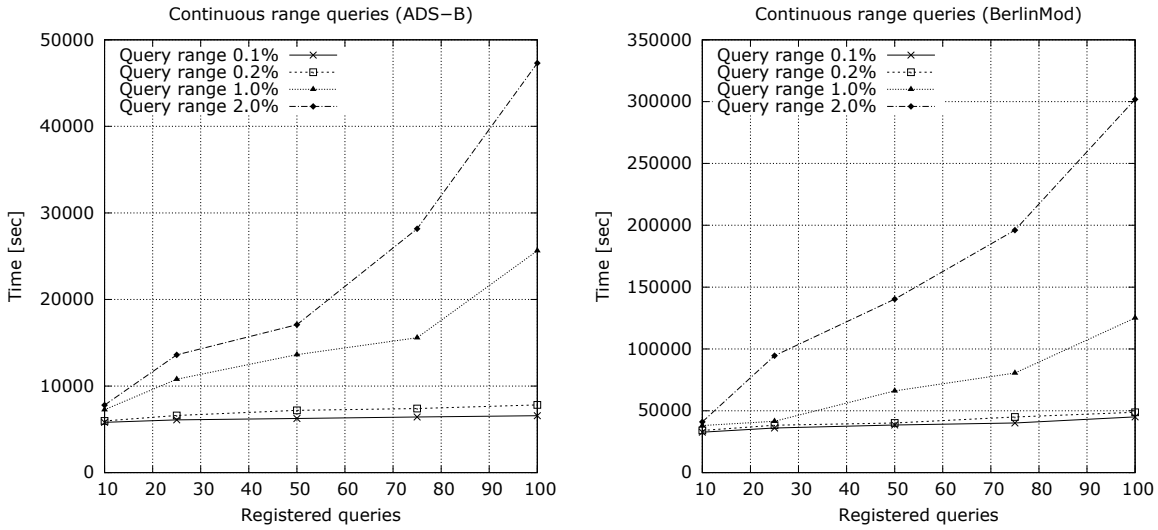
Processing the BerlinMod data stream benefits mostly from parallelization. The elements of the data stream have the smallest tuple size (see Table 8 on Page 37). Due to the small tuple size, more

network operations are required to process this data stream than the other data streams.

6.4 Continuous Range Queries on Data Streams

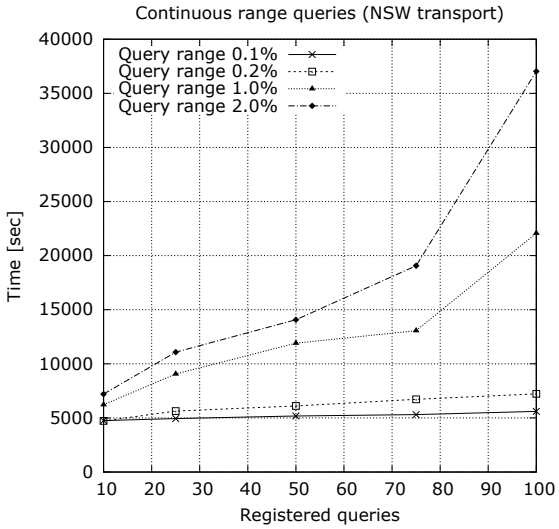
In this experiment, continuous range queries on data streams are performed, and the execution time to process the datasets is measured. During the experiment, the amount of queries and the size of the query range are varied.

In the first step of the experiment, the bounding box of the stream dataset is determined. The query range in the experiment covers a certain percentage (0.1% – 2.0%) of this bounding box. To make the measured execution times comparable, only one bounding box is created per size and reused in all experiments. Afterward, the bounding box is used to register a certain amount (10 – 100) of continuous range queries. The result of the experiment is shown in Figure 19.



(a) ADS-B dataset.

(b) BerlinMod dataset.



(c) NSW transport dataset.

Figure 19: Performing continuous range queries on the datasets with a varying query range and number of registered queries.

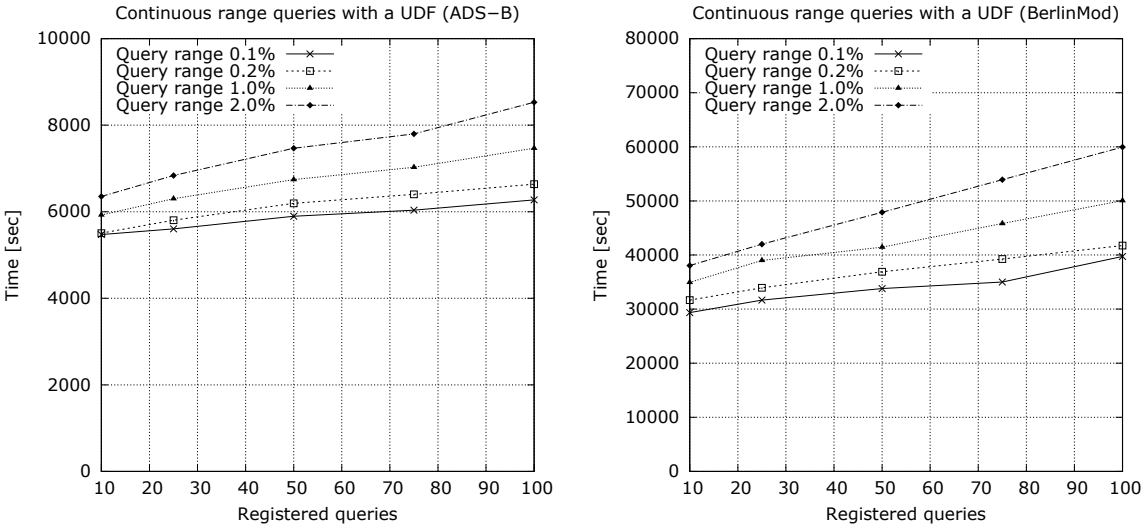
It can be seen in the result of the experiment, that with an increasing number of queries and increasing query range the needed time to import the data stream also increases. The reason for this behavior is that with more registered queries, more query predicates need to be evaluated. With an increasing query range, the registered queries are called more often and they produce more

results that have to be created and sent back to the query client. For example, with 100 registered continuous range queries and a query range of 2% per query, $\approx 2n$ result tuples are produced by processing a stream of n elements. For example, 1 332 861 131 tuples are returned in total by 100 range queries with a query range of 2% per query on the NSW transport dataset consisting of 49 538 352 elements.

6.5 Continuous Range Queries and User-Defined Filters

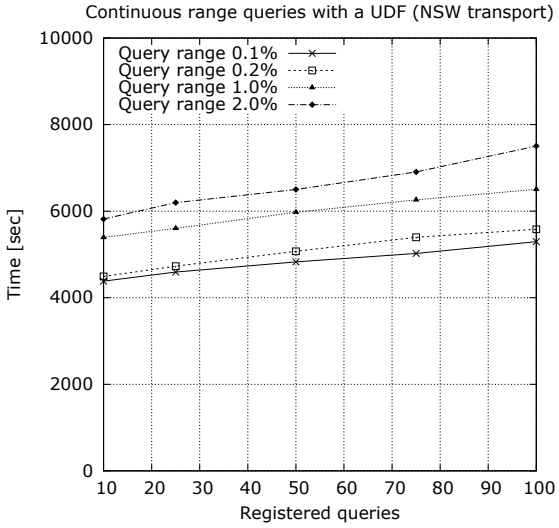
By using a User-Defined Filter (UDF), the query processor of BBoxDB can be extended (see Section 3.6). UDFs can decode the value of the tuple and filter the tuples based on this value.

In this experiment, the UDF from Section 3.6.2 is used to filter the GeoJSON values based on a property value. Therefore, the values of the stream elements are to be parsed into JSON objects. Afterward, the stream of the ADS-B dataset is filtered for the call sign `QTR3WM`, the BerlinMod dataset is filtered for the vehicle id `4093`, and the NSW transport dataset is filtered for the route id `2436_N60`. The result of the experiment can be seen in Figure 20.



(a) ADS-B dataset.

(b) BerlinMod dataset.



(c) NSW transport dataset.

Figure 20: Performing continuous range queries with a filter UDF on the datasets with a varying query range and number of registered queries.

Like in the last experiment, a varying amount of queries is registered with varying query ranges

and the time to process the stream datasets is measured. In this experiment, evaluating the UDF causes some additional work because the values of the stream need to be parsed. However, the additional filter set reduces the number of result tuples significantly.

In the ADS-B dataset, only 2 013 tuples can pass the UDF, in the BerlinMod dataset 24 608 tuples can pass the UDF, and in the NSW transport dataset 76 129 tuples can pass the UDF.

It can be seen in the result of the experiment that the needed time to process the data stream is lower than in the last experiment without using the UDF. Besides, the execution time does not rise as much with an increasing amount of queries as in the last experiment. This is caused by the filtering step of the UDF and, therefore, the reduced amount of results of the continuous queries. This leads to fewer result tuples that need to be produced, transferred through the network, and consumed by the continuous query client. This saves more resources than the additional effort to parse and evaluate the JSON values in the UDF.

6.6 Continuous Spatial Joins on Data Streams

In this experiment, the behavior of BBoxDB Streams is evaluated when continuous spatial joins are performed. The experiment is performed by using the *BerlinMod* dataset and by using the *NSW transport* dataset. The spatial joins are performed between the stream elements and the forests and roads of the OSM dataset. The *ADS-B* dataset is not used in the spatial join experiments. The bounding box of the positions of the aircraft are three-dimensional, and the bounding boxes of the OSM dataset are two-dimensional. Therefore, the datasets can not be joined directly[18].

During the experiment, the number of queries and the size of the region where the spatial join is performed is varied. The join is performed on the bounding boxes of the elements. A spatial join that uses a UDF and works on the real geometries of the data will be discussed in Section 6.7. The result of the join between the BerlinMod stream and the OSM dataset, and the join between the NSW transport dataset and the OSM dataset is shown in Figure 21.

The results show that an increasing amount of registered continuous queries and an increasing amount of query range increase the needed time to process the data stream. This is caused by the increasing amount of elements that are processed by the continuous queries and the increasing amount of result tuples produced by the queries. In general, a spatial join takes more time to calculate than a range query. This is caused by the disk access to retrieve the needed join partners.

The join result sizes of the spatial joins will be discussed in more detail in Table 9, which is contained in the following experiment.

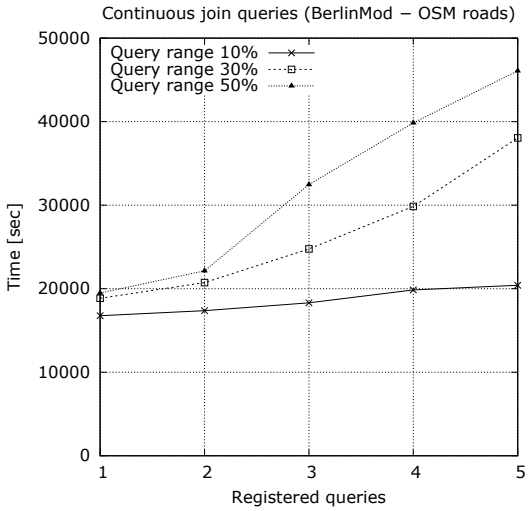
6.7 Continuous Spatial Joins with User-Defined Filters

In this experiment, a spatial join between two stream datasets and the OSM dataset is performed. In contrast to the last experiment, the spatial join is refined by the UDFs, which are discussed in Section 3.6.2. These UDFs decode the GeoJSON encoded elements and refine the spatial join by performing an intersection test on the real geometries.

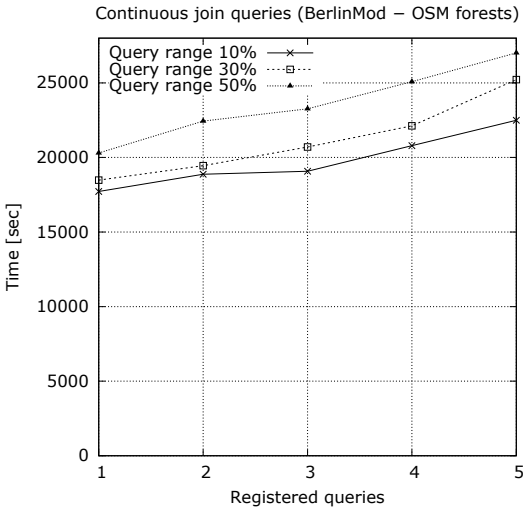
The spatial join between the stream dataset and the roads is refined by the `UserDefinedGeoJson SpatialFilter` UDF, which allows up to a five meter distance between the position of the stream element and the road. The spatial join between the stream dataset and the forests is refined by the `UserDefinedGeoJsonSpatialStrictFilter` which requires that the position of the stream element is inside of the forest.

The decoding of the GeoJSON elements and calculating the intersection performed by the UDFs take some additional time, compared with the last experiment. However, the UDFs remove many join candidates. This reduces the size of the result and the number of tuples that need to be

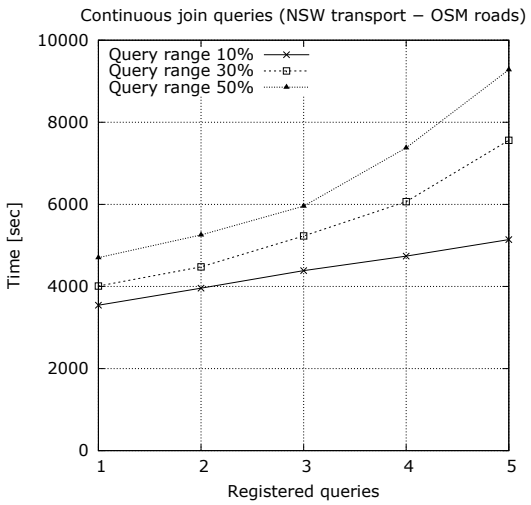
[18] However, the bounding box of the aircraft can be reduced to two dimensions by only using the position to calculate the bounding box. With these two-dimensional bounding boxes, a spatial join between this position and the roads and the forests could be performed. Even though the spatial join could be calculated, the results of the join are more constructed than an actual result because the flying aircraft is not really *inside* of a forest or *on* particular street. Therefore, we decided not to use the ADS-B dataset for the evaluation of the spatial joins.



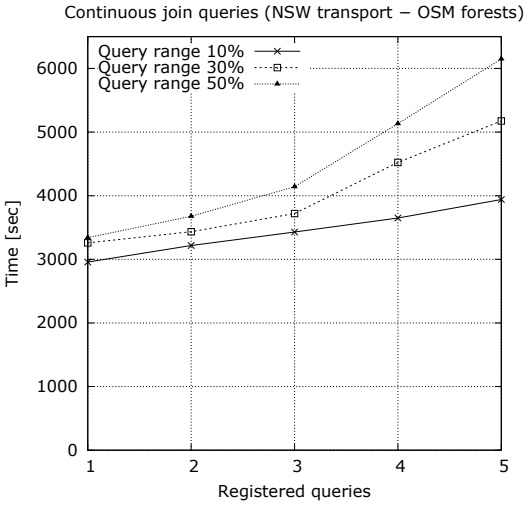
(a) Spatial join of BerlinMod and OSM roads.



(b) Spatial join of BerlinMod and OSM forests.



(c) Spatial join of NSW transport and OSM roads.



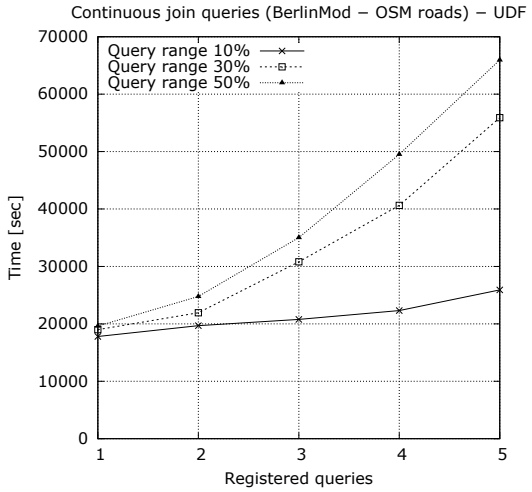
(d) Spatial join of NSW transport and OSM forests.

Figure 21: Performing continuous spatial join queries with a varying query range and number of registered queries.

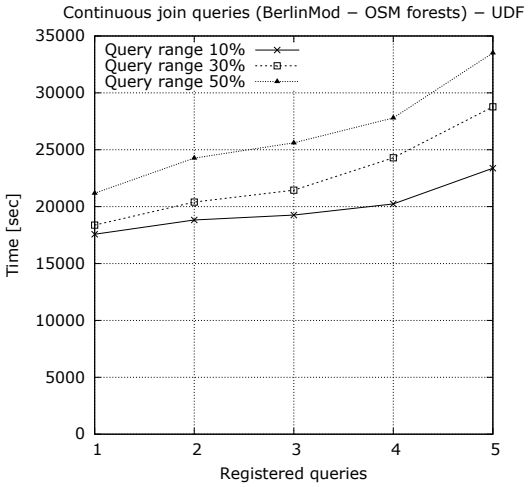
transferred to the query client (see Figure 7 on Page 14). The result of the experiment is shown in Figure 22.

The needed time to execute the continuous spatial join is higher as in the last experiment. This is caused by the additional computations performed by the UDFs. For the stream positions and the forest (a point and a polygon), the UDF has to calculate whether or not the position is inside or outside of the polygon of the forest. For the roads (a point and a line), the distance is calculated. Table 9 shows how many result tuples are produced with and without the refinement performed by the UDFs.

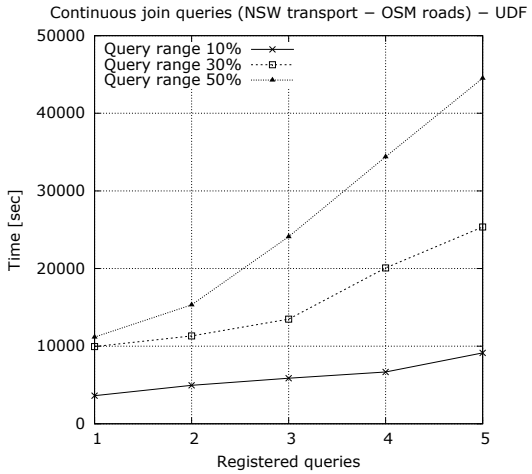
By joining the stream elements with the roads, the cardinality of the result is larger than the cardinality of the stream dataset. This is caused by multiple join partners with overlapping bounding boxes. When the `UserDefinedGeoJsonSpatialFilter` UDF refines the result, the cardinality of the result decreases. The UDF lets all join candidates pass, with have geometries that are closer than five meters. Not all stream positions have a road located in this area. Therefore, the cardinality of the refined join result is smaller than the cardinality of the stream dataset.



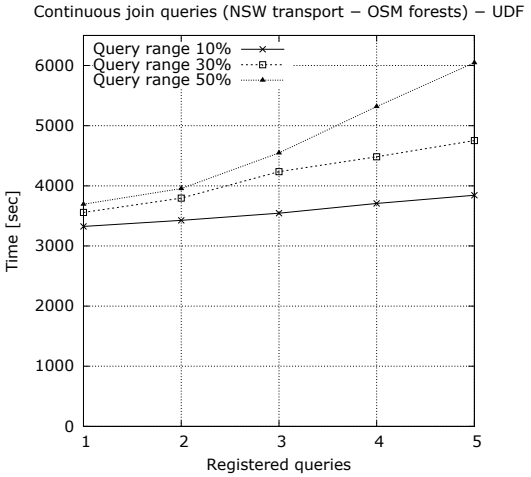
(a) Spatial join of BerlinMod and OSM roads.



(b) Spatial join of BerlinMod and OSM forests.



(c) Spatial join of NSW transport and OSM roads.



(d) Spatial join of NSW transport and OSM forests.

Figure 22: Performing UDF refined continuous spatial join queries with a varying query range and number of registered queries.

6.8 Scaling-up Continuous Join Queries

In this experiment, the continuous spatial join is performed on a varying number of nodes. It is evaluated how horizontal scaling can be used in BBoxDB Streams to speed-up a continuous join operation. The spatial join in the experiment uses a query range of 50% of the space that is covered by the stream dataset. Three parallel continuous queries are registered; the result is refined by a UDF that refines the spatial join (as described in the experiment of Section 6.7). In all experiments, the same query range is used to ensure the same amount of stream elements are processed by the queries, and the same amount of spatial join results are calculated. The time to process the data stream is taken during the experiment and shown in Figure 23.

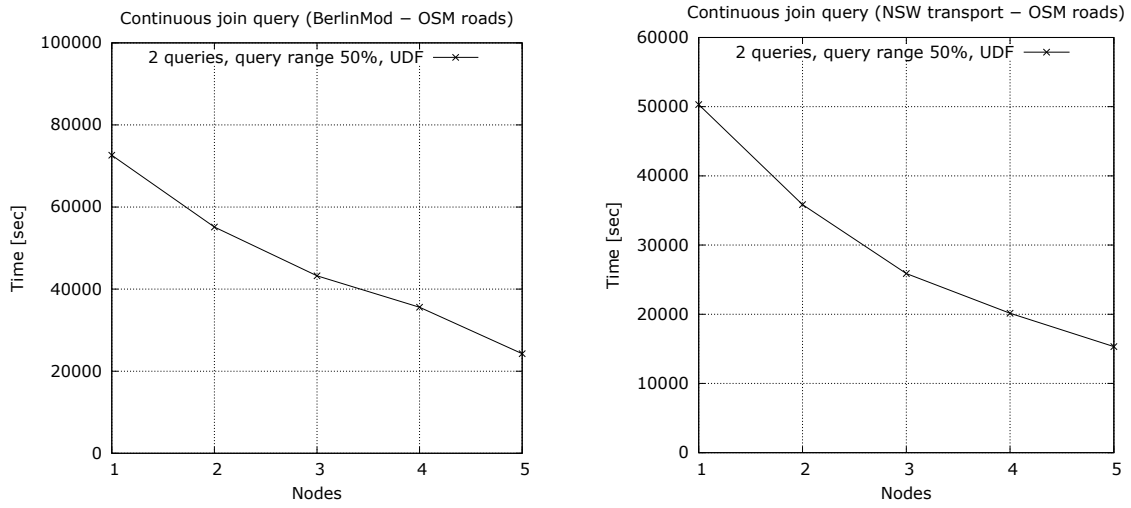
An increasing number of nodes speeds-up the spatial join. This is caused by the additional resources of the additional systems that can be used to calculate the spatial join. The experiment shows that a speed-up factor of ≈ 3 is reached by scaling-up the number of nodes from one node to five nodes.

6.9 Distributing Data

In Section 4.6, different data distribution modes of the stream capturing tool are described. In this section, these strategies are evaluated. The first subsection focuses on the *FETCH* strategy, whereas

Stream Dataset	Static Dataset	Resulting tuples without refinement	Resulting tuples with refinement
BerlinMod	Road	786 266 377	152 136 430
BerlinMod	Forest	5 880 667	1 501 017
NSW transport	Road	110 072 568	36 473 212
NSW transport	Forest	2 530 673	105 187

Table 9: The result size of the spatial join when the result (1) is performed only on the bounding boxes, and (2) refined by a UDF on the real geometries.



(a) Spatial join of BerlinMod and OSM roads.

(b) Spatial join of BerlinMod and OSM forests.

Figure 23: Scaling a UDF refined continuous spatial join queries with a varying number of nodes.

the second subsection focuses on the *STATIC* and *DYNAMIC* strategies.

6.9.1 The FETCH Strategy

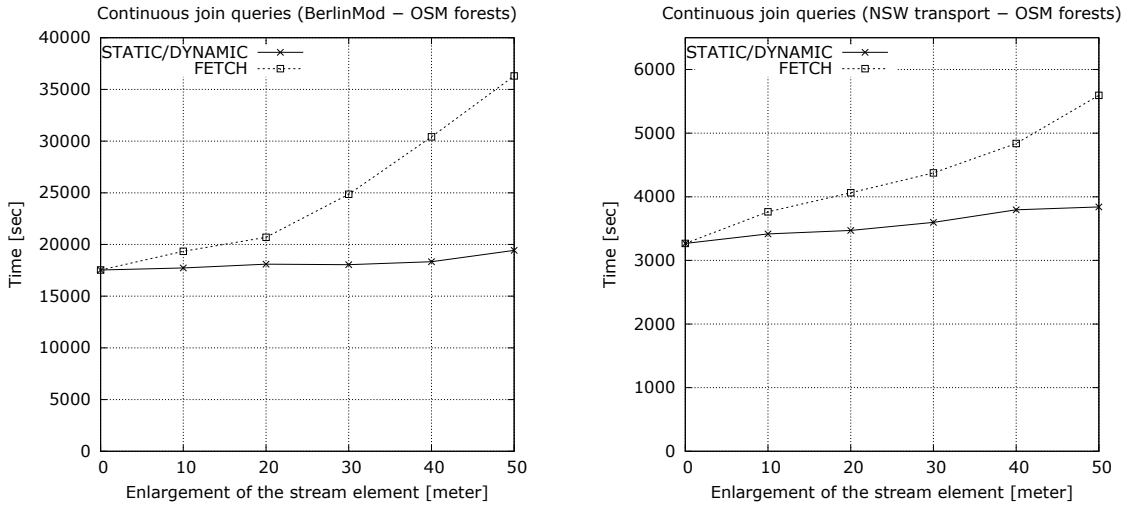
In this experiment, the behavior of the *FETCH* strategy is evaluated. When this strategy is used, the stream capturing tool distributes the tuples unchanged. During the calculation of a spatial join, missing join partners are requested via the network (see Section 4.6.1). In this experiment, the strategy is compared with the *STATIC* and the *DYNAMIC* strategies. These strategies distribute the elements to the required nodes and don't perform a network access during the spatial join.

This experiment compares: (1) the time to perform a spatial join without executing a network access (denoted a *STATIC/DYNAMIC* strategy) with (2) the time to perform a spatial join when join partners need to be fetched via the network (denoted as *FETCH* strategy). The overhead that is caused by the *STATIC/DYNAMIC* strategy in the stream capturing tool is discussed in the next section.

For the evaluation, the continuous spatial join is performed with a varying enlargement of the bounding box. Again, the time to import the stream dataset is measured. The continuous spatial join is performed between the stream datasets and the OSM forest dataset. The OSM forest dataset is used because it is smaller compared to the OSM road dataset. Therefore, increasing the size of the bounding box of the stream element increases the space where the spatial join searches for join partners. Due to the small cardinality of the static dataset, this change does affect the cardinality of the spatial join result much. However, with an increasing enlargement of the bounding box, the spatial join has to perform more and more network operations to fetch the missing join partners from other nodes.

In this experiment, the bounding box of the stream elements is performed by calling the `enlargeStreamTupleBoundBoxByWGS84Meter(..)` method. One continuous query is registered, which covers

the complete space. In addition, the spatial join is refined by the `UserDefinedGeoJsonSpatialStrictFilter` UDF. The results of the experiment are shown in Figure 24.



(a) Spatial join of BerlinMod and OSM forests.

(b) Spatial join of NSW transport and OSM forests.

Figure 24: Performing a continuous spatial join only on local data or by fetching non-local data via the network.

The figure shows that the *FETCH* strategy needs more time to process the stream dataset than the *STATIC* or *DYNAMIC* strategy. This is caused by the network access that is required by using the *FETCH* strategy. The network access causes: (1) some latency to process the query and (2) time to transfer the join partners. Performing a range query on another node took around 65.07 ms on average in the experiment. In the worst case, when all stream element tuples require a network access, this strategy leads to a throughput of $\frac{1\,000}{65.07} = 15.36$ elements per second. Without performing any network access (as done by the *STATIC* and *DYNAMIC* strategy), a throughput of $\approx 14\,000$ elements per second is reached when the operations are performed sequentially. However, this operation could be parallelized to increase the throughput. Nevertheless, the network access requires additional resources and increases the latency of the stream processing. Due to these reasons, the *FETCH* strategy is not recommended to process a data stream.

6.9.2 The STATIC and DYNAMIC Strategies

In this experiment, it is determined how the padding of the bounding boxes (as performed by the *STATIC* or *DYNAMIC* strategy) affects the throughput of the stream capturing tool.

To measure the throughput of the stream capturing tool, the data was only processed by this tool; no data are actually written to the BBoxDB nodes. Otherwise, the network or the disks of the BBoxDB nodes would decrease the throughput. For the evaluation, the datasets are imported using the different strategies, and the throughput of the stream elements is determined.

The *DYNAMIC* strategy was performed in three different configurations: (1) with one client and one registered query (*DYNAMIC 1C-1Q*), (2) with one client and 100 registered queries (*DYNAMIC 1C-100Q*), and (3) with 100 clients and each client is running 100 queries (*DYNAMIC 100C-100Q*). The *UNCHANGED* strategy does not modify the bounding boxes. The strategy is used to show the throughput when no modifications on the bounding boxes are made. The result of the experiment can be seen in Figure 25.

The experiment shows two important points: (1) the used dataset affects the throughput of the stream capturing tool significantly, and (2) modifying the bounding box slows down the throughput of elements only slightly. However, how the padding is determined (by the *STATIC* or the *DYNAMIC* strategy) does not affect the throughput; both strategies modify the bounding box in the same way. Because only the maximum enlargement of all registered continuous queries is used, the number of

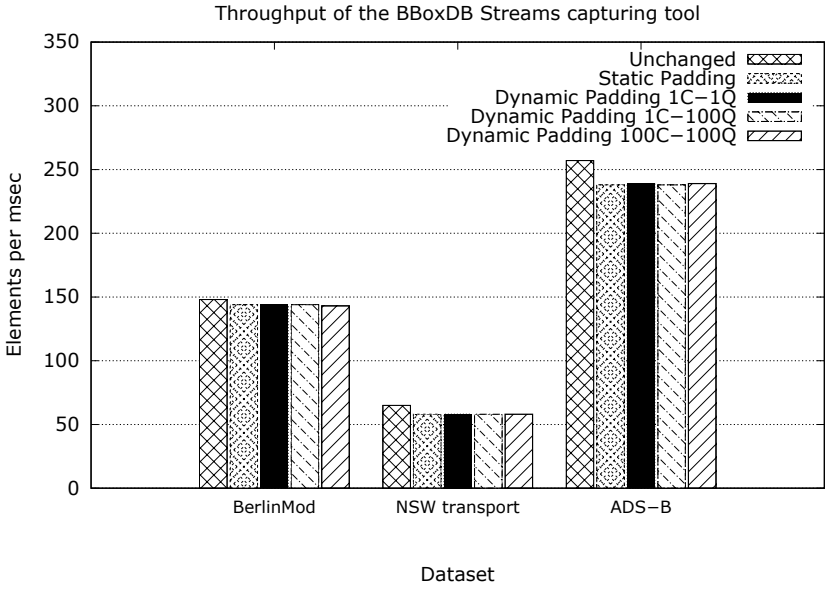


Figure 25: Throughput of the BBoxDB Streams capturing tool.

registered queries or clients does not affect the throughput of the stream capturing tool. The type of the dataset determines how fast the stream elements can be processed. The modification of the bounding box can be performed without decreasing the throughput significantly.

6.9.3 Recommended Distribution Strategy

Three data distribution strategies are implemented in BBoxDB Streams. The experiments of the last subsection have evaluated the behavior of these strategies. The *FETCH* strategy does not need additional logic for the distribution of the tuples. However, the continuous spatial join is slower using this strategy because the join has to perform a network operation. The *STATIC* and the *DYNAMIC* strategies do not introduce a significant overhead in the distribution of the tuples. These strategies work independently of the registered queries in the system, and spatial joins can be performed efficiently using locally stored data only. The drawback of the *STATIC* strategy is that the enlargement that is used in the continuous queries needs to be determined manually. The *DYNAMIC* strategy determines these values automatically and adapts changes when the registered continuous queries are changed. Therefore, we have chosen the *DYNAMIC* strategy as the default data distribution strategy in BBoxDB Streams.

6.10 Pre-partition the Space

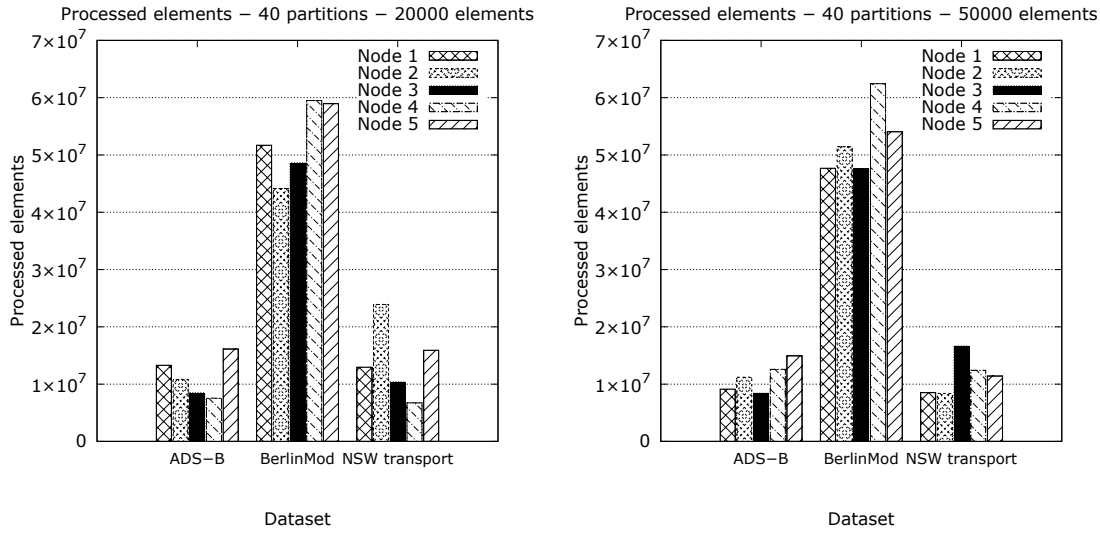
A key feature of BBoxDB Streams is the distribution of a data stream to the nodes of a cluster. To distribute the data stream, the space has to be partitioned. The created partitions determine how the stream elements are distributed. BBoxDB re-partitions unevenly distributed data automatically in the background by splitting and merging the space (see Section 3.4). However, re-partitioning is a slow process because the stored data need to be transferred between the nodes.

When a distribution group is created, only one partition exists, and only the nodes that store this partition are utilized when data are stored. To utilize all nodes directly after a new distribution group is created, the space can be pre-partitioned. This is usually done by taking a small number of random samples from the dataset and creating an initial partitioning. We call this *sample-based pre-partitioning*. However, when a stream is processed, no random samples can be taken to create the pre-partitioning. A stream can only be accessed sequentially and not randomly. Therefore, another technique is needed to pre-partition the space for a data stream.

To pre-partition the space using a data stream, we propose an *element-based pre-partitioning*

approach. By using the element-based pre-partitioning, the first n-elements are captured from the stream, and these elements are used to create the partitions. In contrast to the sampling-based method, all fetched elements are used. The quality of this method is evaluated in this section.

For this experiment, a varying amount of elements are taken from the stream, and a varying amount of partitions are created and assigned via round-robin to the nodes of the cluster. Afterward, the remaining stream is processed, and the distribution of the elements is determined. The distribution of the elements per node for 40 partitions is shown in Figure 26.



(a) 40 partitions, 20000 elements.

(b) 40 partitions, 50000 elements.

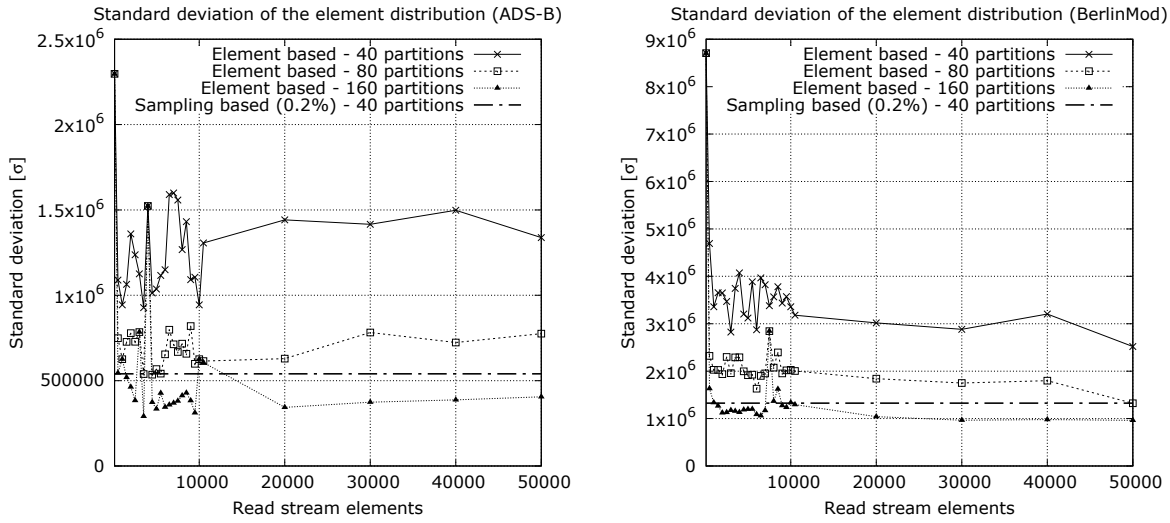
Figure 26: Pre-partitioning the space with the datasets using 40 partitions.

How balanced the data are distributed to the nodes can be quantified by calculating the *standard deviation* (σ). A high standard deviation means that high differences between the number of elements per node do exist. In contrast, a low standard deviation means that the nodes process an almost equal size of elements. Figure 27 shows the standard deviation when the amount of read stream elements is varied and the number of created partitions is varied. In the experiment, the created partitions are mapped via round-robin to the five nodes of the cluster, and the resulting standard deviation between the processed stream elements of the nodes is calculated.

It can be seen in the figures that the stream element distribution becomes more balanced when more partitions are created (indicated by a lower standard deviation). It can also be seen that the amount of read stream tuples has only a slight effect on balancing the stream elements. For example, reading 20 000 or 50 000 stream elements results in the almost identical standard deviation.

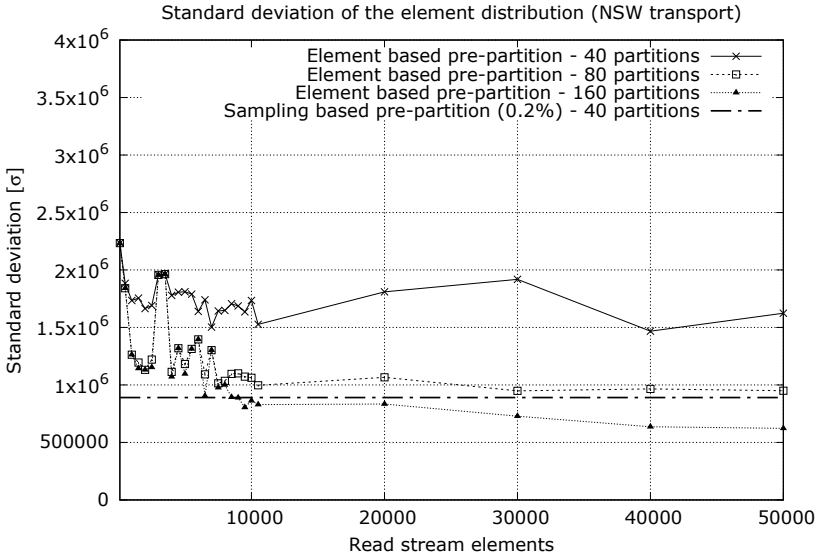
The data streams that are used in the experiments contain the position data of several entities. These entities are contained repeatedly in the data stream with different positions. The entities move, new entities (e.g., aircraft that have taken off) appear, and other entities disappear (e.g., aircraft that have landed) from the data stream. However, the entities are distributed across the area covered by the data stream at any time. Taking the first n elements from a data stream to pre-partition the space generates a proper partitioning of the space.

In the last three figures, also the standard deviation of the sampling-based pre-partitioning approach is depicted for a comparison. The sampling based pre-partitioning takes 0.2% of the data as samples and creates 40 partitions. It can be seen, that the element-based pre-partitioning approach generates a pre-partitioning of the same quality. Therefore we recommend this strategy when a data stream should be used to pre-partition the space. However, this only works when the stream has a stable data distribution which is equivalent to the data distribution of the first n elements of the stream. In the data streams that are handled in this paper, this is given for an accumulation of airplanes in Europe and the USA, cars in densely populated areas of Berlin or buses in the Sydney metropolitan area.



(a) ADS-B dataset.

(b) BerlinMod dataset.



(c) NSW transport dataset.

Figure 27: The standard deviation of the stream element distribution of datasets.

7 Conclusion

In this paper, we have presented an extension to the datastore BBoxDB called BBoxDB Streams. The extension allows the handling of n-dimensional data streams in an efficient and distributed manner. Data streams of any dimension can be handled. Point and non-point data are supported. Data streams are imported by a stream capturing tool. The stream capturing tool reads the data stream, converts it into BBoxDB tuples, and spreads them to a cluster of nodes. Converter for common data stream formats such as ADS-B or GTFS real-time encoded data are shipped with BBoxDB Streams. In contrast to existing stream processing systems, the stream is partitioned and distributed based on the location of the stream elements in space. This allows the efficient handling of n-dimensional data streams.

Two types of queries are implemented: (1) continuous range queries which allow comparing the stream with a query rectangle, and (2) continuous spatial joins which allow the comparison of the stream elements with previously stored big data.

Transformations of the queries can manipulate the bounding boxes of the stream elements. To the best of our knowledge, we provide with this paper the first stream processing system that is

optimized for the handling of n-dimensional data and capable of comparing stream elements with previously stored data. Additional topics, such as the efficient distribution of stream objects to the cluster nodes, are discussed. The evaluation of the software has shown that BBoxDB Streams is a scalable solution for processing multi-dimensional data streams.

BBoxDB Streams enhances the GUI of BBoxDB. Several real-world data streams can be observed, and queries can be executed on these streams. For instance, aircraft movements can be observed (ADS-B data) by continuous range queries, or buses in Sydney (NSW transport data) can be observed and continuously joined with a road network or spatial data like forests. At the moment, queries are expressed by (1) writing query plans in JSON, (2) by using the `QueryPlanBuilder`, and (3) by using the GUI. An enhancement for the next version of the software could be a query language to allow more interactive queries. Besides, the experiments were performed in a small cluster of five nodes. There are plans to evaluate the system in a cluster with more hardware nodes.

References

[1] T. Akidau, S. Chernyak, and R. Lax. *Streaming Systems: The What, Where, When and how of Large-scale Data Processing.* O'Reilly Media, Incorporated, 2018.

[2] L. Alarabi, M.F. Mokbel, and M. Musleh. ST-Hadoop: A MapReduce Framework for Spatio-Temporal Data. *Geoinformatica*, 22(4):785–813, October 2018.

[3] Apache Accumulo project - Website. `https://accumulo.apache.org/`, 2021. [Online; accessed 20-Jan-2021].

[4] Apache Hadoop project - Website, 2021. `https://hadoop.apache.org/` - [Online; accessed 20-Jan-2021].

[5] Apache HBase project - Website. `https://hbase.apache.org/`, 2021. [Online; accessed 12-Jan-2021].

[6] Apache Kafka - Datatypes. `https://kafka.apache.org/10/documentation/streams/developer-guide/datatypes`, 2021. [Online; accessed 03-Jan-2021].

[7] Apache Kafka - Joining data. `https://kafka.apache.org/20/documentation/streams/developer-guide/dsl-api.html#joining`, 2021. [Online; accessed 03-Jan-2021].

[8] Apache Kafka - KTables. `https://kafka.apache.org/20/documentation/streams/developer-guide/dsl-api.html#streams_concepts_ktable`, 2021. [Online; accessed 03-Jan-2021].

[9] Apache Storm project - Website, 2021. `https://storm.apache.org/` - [Online; accessed 20-Jan-2021].

[10] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, September 2001.

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

[12] P. Baumann, P. Furtado, R. Ritsch, and N. Widmann. The rasdaman approach to multi-dimensional database management. In *Proceedings of the 1997 ACM Symposium on Applied Computing*, SAC '97, pages 166–173, New York, NY, USA, 1997. ACM.

[13] BBoxDB project - OpenStreetMap data to GeoJSON converter. `https://jnidzwetzki.github.io/bboxdb/tools/dataset.html`, 2021. [Online; accessed 07-Mar-2021].

[14] BBoxDB project - Website. `http://bboxdb.org`, 2021. [Online; accessed 03-Jan-2021].

[15] R. Bellman. *Dynamic Programming*. Dover Publications, 1957.

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

[17] E. Brewer. CAP twelve years later: How the "rules" have changed. *IEEE Computer*, 45(2):23–29, 2012.

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

[19] A. Colmerauer and P. Roussel. The birth of prolog. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *History of Programming languages—II*, pages 331–367. ACM, New York, NY, USA, 1996.

[20] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[21] C. Düntgen, T. Behr, and R.H. Güting. Berlinmod: a benchmark for moving object databases. *VLDB Journal*, 18(6):1335–1368, 2009.

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

[23] R. Escriva, B. Wong, and E.G. Sirer. 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, pages 25–36, New York, NY, USA, 2012. ACM.

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

[25] C.L. Forgy. OPS5 user's manual. Technical report, Department of Computer Science Carnegie, Mellon University, Pittsburgh, Pennsylvania, 1981.

[26] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[27] A. Fox, C. Eichelberger, J. Hughes, and S. Lyon. Spatio-temporal indexing in non-relational distributed databases. In *2013 IEEE International Conference on Big Data*, pages 291–299, Oct 2013.

[28] E. Friedman and K. Tzoumas. *Introduction to Apache Flink: Stream Processing for Real Time and Beyond*. O'Reilly Media, Inc., 1st edition, 2016.

[29] E. Gamma, R. Helm, R. Johnson, and J.M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[30] The Wikipedia article about Geohashing. `https://en.wikipedia.org/wiki/Geohash`, 2021. [Online; accessed 14-Jan-2021].

[31] GeoMesa project - Website. `http://www.geomesa.org`, 2021. [Online; accessed 20-Jan-2021].

[32] R.H. Güting. Second-order signature: A tool for specifying data models, query processing, and optimization. *SIGMOD Rec.*, 22(2):277–286, June 1993.

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

[34] R.H. Güting and J. Lu. Parallel SECONDO: Scalable Query Processing in the Cloud for Non-standard Applications. *SIGSPATIAL Special*, 6(2):3–10, March 2015.

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

[36] D. Han and E. Stroulia. HGrid: A Data Model for Large Geospatial Data Sets in HBase. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 910–917, 06 2013.

[37] C. Hewitt. Planner: A language for proving theorems in robots. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, page 295–301, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[38] E.F. Hill. *Jess in Action: Java Rule-Based Systems.* Manning Publications Co., Greenwich, CT, USA, 2003.

[39] J. Hughes, A. Annex, C. Eichelberger, A. Fox, A. Hulbert, and M. Ronquest. Geomesa: a distributed architecture for spatio-temporal fusion. In *Geospatial Informatics, Fusion and Motion Video Analytics V, 94730F*, volume 9473 of *Proceedings SPIE*, pages 9473 – 9473 – 13, 2015.

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

[41] S. Idreos, E. Liarou, and M. Koubarakis. Continuous multi-way joins over distributed hash tables. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '08, pages 594–605, New York, NY, USA, 2008. ACM.

[42] J. Kreps. Questioning the lambda architecture. `https://www.oreilly.com/ideas/questioning-the-lambda-architecture`, 2014. [Online; accessed 03-Jan-2021].

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

[44] S. Li, S. Hu, R.K. Ganti, M. Srivatsa, and T.F. Abdelzaher. Pyro: A Spatial-Temporal Big-Data Storage System. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 97–109, 2015.

[45] E.K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys Tutorials*, 7(2):72–93, 2005.

[46] A.R. Mahmood, A.M. Aly, T. Qadah, E.K. Rezig, A. Daghistani, A. Madkour, A.S. Abdelhamid, M.S. Hassan, W.G. Aref, and S. Basalamah. Tornado: A Distributed Spatio-textual Stream Processing System. *Proc. VLDB Endow.*, 8(12):2020–2023, August 2015.

[47] N. Marz. How to beat the cap theorem. `http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html`, 2011. [Online; accessed 03-Jan-2021].

[48] M.F. Mokbel, X. Xiong, M.A. Hammad, and W.G. Aref. Continuous Query Processing of Spatio-Temporal Data Streams in PLACE. *GeoInformatica*, 9(4):343–365, Dec 2005.

[49] G.M. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing.* International Business Machines Company, 1966.

[50] MySQL Reference - Features of the User-Defined Function Interface, 2021. https://dev.mysql.com/doc/refman/8.0/en/udf-features.html [Online; accessed 17-Jan-2021].

[51] Z. Nabi. *Pro Spark Streaming: The Zen of Real-Time Analytics Using Apache Spark*. Apress, Berkely, CA, USA, 1st edition, 2016.

[52] N. Narkhede, G. Shapira, and T. Palino. *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale*. O'Reilly Media, Inc., 1st edition, 2017.

[53] National Imagery and Mapping Agency, Department of Defense. World Geodetic System 1984: its definition and relationships with local geodetic systems. Technical Report TR8350.2, St. Louis, MO, USA, January 1984.

[54] J.K. Nidzwetzki and R.H. Güting. Distributed Secondo: An Extensible and Scalable Database Management System. *Distributed and Parallel Databases*, 35(3-4):197–248, December 2017.

[55] J.K. Nidzwetzki and R.H. Güting. Demo Paper: Large Scale Spatial Data Processing With User Defined Filters In BBoxDB. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4125–4128, 2019.

[56] J.K. Nidzwetzki and R.H. Güting. BBoxDB: A Distributed and Highly Available Key-Bounding-Box-Value Store. *Distributed and Parallel Databases*, 38:439–493, June 2020.

[57] J.K. Nidzwetzki and R.H. Güting. BBoxDB Streams: Distributed Processing of Real-World Streams of Position Data (Demo-Paper). In Y. Velegrakis, D. Zeinalipour-Yazti, P.K. Chrysanthis, and F. Guerra, editors, *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, pages 662–665. OpenProceedings.org, 2021.

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

[59] R. Obe and L. Hsu. *PostgreSQL: Up and Running*. O'Reilly Media, Inc., 2012.

[60] Open Street Map Project - Website, 2021. http://www.openstreetmap.org - [Online; accessed 15-Jan-2021].

[61] Open Street Map Project - Object Properties, 2021. https://wiki.openstreetmap.org/wiki/Category:Properties - [Online; accessed 15-Jan-2021].

[62] Oracle. The Documentation of the spatial GeoRaster feature, 2021. https://docs.oracle.com/cd/B19306_01/appdev.102/b14254/geor_intro.htm - [Online; accessed 22-Jan-2021].

[63] J. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. *SIGMOD Rec.*, 19(2):343–352, May 1990.

[64] W. Palma, R. Akbarinia, E. Pacitti, and P. Valduriez. DHTJoin: processing continuous join queries using DHT networks. *Distributed and Parallel Databases*, 26(2):291, Aug 2009.

[65] PostGIS. The Documentation of the raster datatype, 2021. https://postgis.net/docs/RT_reference.html - [Online; accessed 22-Jan-2021].

[66] J. Qi, R. Zhang, C.S. Jensen, K. Ramamohanarao, and J. He. Continuous spatial query processing: A survey of safe region based techniques. *ACM Comput. Surv.*, 51(3), May 2018.

[67] M. Stonebraker, P. Brown, J. Becla, and D. Zhang. Scidb: A database management system for applications with complex analytics. *Computing in Science and Engg.*, 15(3):54–62, May 2013.

[68] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. *SIGMOD Rec.*, 21(2):321–330, June 1992.

[69] The Esri Project. Esri Geometry API for Java, 2021. `https://github.com/Esri` - [Online; accessed 03-Mar-2021].

[70] The Tiny MD-HBase project on Github, 2021. `https://github.com/shojinishimura/Tiny-MD-HBase` - [Online; accessed 26-Jan-2021].

[71] K. Vibhore, A. Henrique, G. Bugra, and W. Kun-Lung. DEDUCE: at the intersection of MapReduce and stream processing. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, Proceedings*, pages 657–662, 2010.

[72] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[73] The Website of the adsbhub.org project, 2021. `http://adsbhub.org` - [Online; accessed 12-Jan-2021].

[74] The Website of the Open Data Hub for New South Wales transport data, 2021. `https://opendata.transport.nsw.gov.au` - [Online; accessed 20-Jan-2021].

[75] M. Widenius and D. Axmark. *MySQL Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

[76] F. Zhang, Y. Zheng, D. Xu, Z. Du, Y. Wang, R. Liu, and X. Ye. Real-time spatial queries for moving objects using storm topology. *ISPRS International Journal of Geo-Information*, 5(10), 2016.

[77] R. Zhang, D. Lin, K. Ramamohanarao, and E. Bertino. Continuous intersection joins over moving objects. In *2008 IEEE 24th International Conference on Data Engineering*, pages 863–872, 2008.

[78] Y. Zhang, M. Kersten, and S. Manegold. SciQL: Array Data Processing Inside an RDBMS. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1049–1052, New York, NY, USA, 2013. ACM.

[79] X. Zhou, X. Zhang, Y. Wang, R. Li, and S. Wang. Efficient Distributed Multi-dimensional Index for Big Data Management. In *Proceedings of the 14th International Conference on Web-Age Information Management*, WAIM'13, pages 130–141, Berlin, Heidelberg, 2013. Springer-Verlag.

Verzeichnis der zuletzt erschienenen Informatik-Berichte

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

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

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

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

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

[377] Jan Kristof Nidzwetzki, Ralf Hartmut Güting:
BBoxDB - A Distributed and Highly Available Key-Bounding-Box-Value Store, *5/2018*

[378] Marija Kulaš:
On separation, conservation and unification, *06/2019*

[379] Fynn Terhar, Christian Icking:
A New Model for Hard Braking Vehicles and Collision Avoiding Trajectories, *06/2019*

[380] Fabio Valdés, Thomas Behr, Ralf Hartmut Güting:
Parallel Trajectory Management in Secondo, 0*1/2020*

[381] Ralf Hartmut Güting, Thomas Behr, Jan Kristof Nidzwetzki:
Distributed Arrays – An Algebra for Generic Distributed Query Processing, 05*/2020*

[382] Raphael Herding, Lars Mönch:
A SHORT-TERM DEMAND SUPPLY MATCHING APPROACH FOR SEMICONDUCTOR SUPPLY CHAINS, 06/2021