INFORMATIK
BERICHTE

371 – 05/2016

DISTRIBUTED SECONDO:
An extensible highly available and scalable database management system

Jan Kristof Nidzwetzki, Ralf Hartmut Güting

Fakultät für Mathematik und Informatik
D-58084 Hagen

Distributed Secondo: An extensible highly available and scalable database management system

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

May 6, 2016

Abstract

This paper describes a novel method to couple a standalone database management system (DBMS) with a highly scalable key-value store. The system employs Apache Cassandra as data storage and the extensible DBMS Secondo as a query processing engine. The resulting system is highly scalable and can handle failing systems. All the data models and functions implemented in Secondo can be used in a scalable way without changing the implementation. Many aspects of the distribution are hidden from the user. Existing sequential queries can be easily converted into parallel ones.

1 Introduction

Database management systems (DBMS) have to deal with rapidly growing volumes of data [13]. Despite all the advances in hardware, it is often impossible to store and process large volumes of data on a single computer. In order to handle such amounts of data, *distributed database systems* (DDBMS) were developed [35] [34] [4]. DDBMS spread the data and the workload across multiple hardware nodes which are connected by a computer network. Therefore, the DDBMS has to deal with the following topics: (*i*) *distributed data storage* [5] [15]; and (*ii*) *distributed query processing* [30, p. 17].

- Distributed data storage systems use various hardware nodes to store large amounts of data. Techniques like replication are implemented to deal with failing nodes.

  These systems often provide only a very simple data model, e.g., a key-value interface. Application developers have to map complex data structures on this data model.

- Distributed query processing systems spread the calculation of a particular query across multiple systems. In most cases, a central coordinator exists. This coordinator splits the calculation into smaller tasks and assigns these tasks to the query processing systems. In many implementations, distributed data storage systems constitute the backbone for distributed query processing systems. The distributed data storage supplies the query processing systems with the necessary data.

Today, the family of modern distributed databases is referred as NoSQL-Systems (Not only SQL-Systems). In contrast to traditional database systems, these systems scale well, however, they provide only a small set of functions to process the stored data. Developers need to implement that kind of functionality directly into their application code. Algorithms like joins have to be

repeatedly implemented in the application. This is error prone and makes the handling of these systems complicated and time consuming.

In contrast, traditional database systems scale badly across multiple hardware nodes, however, they provide a wide set of functions to analyze data; e.g. functions like aggregation or joins. Some of the DBMS can handle complex data types such as spatial data and furthermore provide operations on them like spatial joins.

Handling node failures is a challenging task in distributed systems; they can contain more than 1 000 hardware nodes and the possibility of a hardware related failure is high. Google observed that in a cluster with 10 000 nodes, around 1 000 hardware related failures occurred in the first year [16]. In this regard, a distributed database must be able to deal with such failures.

This paper presents a novel method to build a highly available database system. The system employs the single computer DBMS Secondo [18] for query processing and the software *Apache Cassandra* [23] as distributed data storage. The resulting system is called Distributed Secondo and focuses on the following topics:

- High availability and high scalability: The architecture of the system is highly scalable and allows a user to add new systems easily. Additional systems can be added to process or to store larger amounts of data. No central coordinator exists and therefore, the architecture of the whole system is highly available.
- Support for high update rates: The system is capable of handling data streams. Data streams can be captured, stored and processed.
- Support for complex queries: Stored data can be analyzed with complex operations like joins. One focus of Secondo is the support for moving objects, i.e., a moving car or train. All the functions[1] and data models of Secondo are available in a distributed manner. The functions can be used in a scalable way without changing the implementation. Therefore, it's possible to process relational data, spatial or spatio-temporal data, to name just a few data models.
- Adaptive use of resources for query processing: The input data of queries is split up into small *units of work* (UOWs). UOWs are assigned dynamically to the query processing nodes. This allows the nodes to balance out speed differences, e.g., a different hardware configuration of the nodes like CPUs, RAM or Disks.
- A decentralized algorithm for job scheduling: The nodes of the system create, schedule and execute UOWs based only on local information. This avoids complicated distributed scheduling algorithms or the usage of a central coordinator.

In addition, the following topics are addressed in this paper:

- Transformation of existing queries: Most of the details about the data distribution are hidden from the user. This makes it easy to convert existing Secondo queries into parallel ones or to write new queries.
- A practical interface for coupling a DBMS with a key-value store: The kernel of Secondo is extended by an algebra module to realize the communication with Cassandra. In addition, operations are defined to manage distributed query processing. The design of the interface is universal; almost the same set of operations need to be implemented in other DBMS to create a comparable system.
- Formalization of the logical ring: This paper proposes a formalization and a notation of the logical ring of Cassandra. This notation is used to describe the algorithms for this system.

[1] Some special functions, like the interaction with other distributed systems, are excluded.

- BerlinModPlayer: To simulate a stream of GPS coordinate updates, we have developed a tool called BerlinModPlayer. This tool simulates a fleet of moving vehicles driving though the streets of Berlin. Each vehicle sends its position every few seconds to a central system. In this way, a stream of GPS coordinate updates is generated.

The major contribution of this paper is to demonstrate how a key-value store can be combined with a traditional DBMS to process complex queries in a distributed, scalable and highly available manner. Compared to many existing systems, Distributed Secondo is able to handle data streams. Furthermore, a novel algorithm is used to parallelize and schedule the queries, based on the structure of the distributed key-value store.

The rest of the paper is structured as follows: Section 2 introduces all the related work. Section 3 covers the required knowledge about Secondo and Cassandra. Section 4 presents the architecture of Distributed Secondo. Section 5 describes the formulation of queries. Section 6 describes the execution of queries; details about the generation of UOWs are also discussed. Section 7 presents some example use cases. Section 8 contains an experimental evaluation of the system. Section 9 concludes the paper.

2 Related Work

Numerous systems exist to work with large amounts of data. This section names the best known systems and describes the closest relatives of Distributed Secondo. *Chord* [37] is a *distributed hash table* (DHT). It focuses on storing data across many systems. The system is a pure data store, it does not provide functionality to analyze the stored data. *Amazon Dynamo* [10] is one of the originators of Cassandra. Dynamo uses some of the concepts of Chord to provide a highly available key-value store. These concepts are also used in Cassandra, but Cassandra provides a more complex data model. Several researchers have proposed algorithms to compute complex operations, e.g. joins, over DHTs. *RJoin* [20] splits up multi-way joins into joins with fewer attributes and computes them independently on multiple nodes. Only tuples that are inserted after the query was submitted, are taken into account for the result. The attributes of each inserted tuple are indexed and distributed in the DHT. *DHTJoin* [31] is an improved algorithm for computing a join over a DHT. Compared to RJoin, DHTJoin does not index tuples that could not contribute to the join result.

In 2004, Google presented a framework for processing large amounts of data. The framework is called *MapReduce* [9] and it is named after the two functions *map* and *reduce* which are used heavily in the framework. An application developer is only required to implement these two functions; all aspects about the distribution and fault-tolerance are hidden. With *Hadoop* [42] an open source implementation of this framework exists. MapReduce uses the *Google file system* (GFS) [15] to provide a highly available and distributed data storage. New data can be added to GFS but it is impossible to modify existing data. Hadoop implements its own distributed data storage called *Hadoop file system* (HDFS) [42, p. 41ff].

Google *BigTable* [5] is a data storage system built on top of GFS. The system provides a multi-dimensional key-value store designed to scale into the petabyte range. Each stored row in BigTable is identified by a unique key. A single row can store one or more column families. A column family describes the structure of the stored data. BigTable is considered as one of the originators of Cassandra; Cassandra uses almost the same data model. *Apache HBase* [14] is an open source implementation of BigTable.

Implementing Map-/Reduce-Jobs is a time consuming and challenging task. Traditional DBMS provide a wide range of predefined operators. Calculations can be easily described in SQL. To simplify the work with MapReduce, *Hive* [39] introduces a SQL-like interface for creating Map-/Reduce-Jobs; the jobs can be described in *HiveQL*. In addition, several approaches exist to couple Hadoop with a full featured database to analyze data. Most of them store the data in HDFS and load the data on demand into the DB.

Apache Drill [2] is a query engine for Hadoop that can understand SQL and reads self-describing

and semi-structured input data. Drill focuses on data analysis and does not provide techniques to modify data.

HadoopDB [1] is a hybrid of a DBMS and MapReduce technologies. The software emphasis is on analytical workloads. Hive is used as a SQL interpreter in this system, Hadoop is used as the task coordinator. The calculation of the result is executed by the DBMS. Like Drill, the focus is on data analysis, updates of the data are not supported.

In recent years, another Secondo based database hybrid called Parallel Secondo [26] was developed. It combines Secondo with Hadoop for parallel query execution and data distribution. In contrast to Distributed Secondo, Parallel Secondo does not focus on data updates and its architecture contains a master node, which is a single point of failure. Data is stored in HDFS or in a lightweight file system called *Parallel Secondo File System* (PSFS) which uses *secure copy* (scp) to interchange data between nodes.

A different approach without MapReduce is taken by *Google F1* [36]. F1 is a distributed relational DBMS that understands SQL and supports indexing and transactions. Google F1 was built on top of the distributed database *Spanner* [6] which was also developed by Google. To ensure global consistency, Spanner makes heavy use of special hardware for time synchronization like GPS and atomic clocks [6, p. 2]. In Spanner, the data is replicated across multiple data centers and the *Paxos protocol* [24] is used to achieve consensus.

Processing data streams is another important topic for modern applications. *IBM InfoSphere Streams* [33] is an engine for data stream processing. How the data streams are processed can be described in the *Streams Processing Language* (SPL) [19]. SPL is a language to process, transform and analyze data streams. *DEDUCE* [22] is an approach to integrate MapReduce-Jobs into the IBM InfoSphere Streams software.

3 Building Blocks of the System

Distributed Secondo uses Cassandra as data store and Secondo as a query processing engine. This section covers the required basics about these two software components to understand the architecture of Distributed Secondo.

3.1 Secondo

Secondo is an extensible database system developed at the university of Hagen. The DBMS consists of three main components: (*i*) a kernel; (*ii*) a graphical user interface; and (*iii*) an optimizer. The kernel is not bound to a particular data model. Algebra modules are used to extend the kernel and to define the supported data types and their operations. Data types can be numbers (e.g. integers), spatial objects (e.g. a line) or index structures to name just a few. On these data types, operations are defined like arithmetical functions, joins or string operations. Secondo is capable of handling two different types of queries: (i) SQL; and (ii) execution plans. The first type allows the user to describe a query in SQL. The SQL query is handled by the optimizer of Secondo and translated into an execution plan. The latter type allows the user to describe execution plans directly. For example, the SQL query `select * from Customer where Id = 11;` can also be expressed with the execution plan query `query Customer feed filter[.Id = 11] consume;`

The SQL query only describes the result of the computation whereas the execution plan query describes how the result is computed. The relation Customer is read by the **feed** operator and a stream of tuples is created. This tuple stream is read by the **filter** operator; the operator lets only the tuples pass, where the attribute Id is 11. Finally, the resulting tuple stream is collected by the **consume** operator which creates a relation from the stream, which is the result of the query.

3.2 Cassandra

Cassandra is a highly available key-value store which is freely available and licensed under the *Apache License 2.0* [3]. In Cassandra, data is stored in *keyspaces* comparable to databases in a DBMS. The

primary goal of Cassandra is the availability of the stored data. Therefore, the data is replicated on several systems. On the creation of a keyspace a replication factor is provided. For example, a replication factor of 3 means that all data of this keyspace is stored on 3 Cassandra nodes.

3.2.1 The Logical Ring

The key element of Cassandra is a DHT, organized as a *logical ring*. The logical ring consists of numbers called tokens which are ordered sequentially around the ring. The token with the highest number is connected back to the token with the lowest number (see Figure 1). Upon initialization, every Cassandra node obtains one or more tokens. The system is placed at the position according to its token(s). The range between two adjacent Cassandra nodes is called *token range*. A node is responsible for the token range between its token(s) and the token(s) of its predecessor node(s). A formal definition of the logical ring is given in Section 6.1. A consistent hash function [21] is used to determine the placement of data.

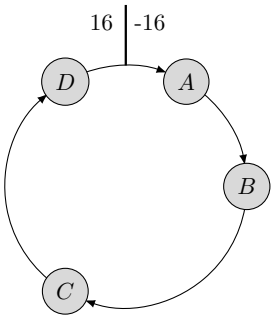


Figure 1: A logical ring that covers the token range from -16 to 16. The ring consists of four hardware nodes (A, B, C, D). Node B is responsible for the range between nodes A and B.

3.2.2 Consistency

Cassandra uses the concept of *tunable consistency*. For every operation, a *consistency level* has to be specified. The consistency level determines how many systems must be available to fulfill a operation successfully (see Table 1). Using a consistency level that includes more systems leads to better consistency. Written data is propagated to more systems and data is read from more systems which improves the probability that more recent data is read. The drawback is, that less systems can fail to complete the request successfully.

Consistency	Description
`ONE`	The data has to be read from at least one node to fulfill this operation.
`QUORUM`	It is required that at least $(\frac{replication\ factor}{2} + 1)$ nodes have to answer.
`ALL`	All nodes that store the data have to answer.

Table 1: Consistency levels supported by Cassandra.

In addition, Cassandra implements the concept of *eventual consistency* [40]. This consistency model guarantees that all accesses to a particular data item will see eventually the last updated value. How long it takes to see the last updated value is not specified. Cassandra maintains a time stamp for each stored value. If different versions of a value are read, only the value with the highest time stamp is returned.

4 Distributed Secondo

In order to resolve the issues presented in this paper, a novel method for coupling an existing DBMS with a key-value store is proposed. This system employs the extensible DBMS Secondo as a query processing engine coupled with the highly scalable key-value store Apache Cassandra as data storage. As the system is a distributed version of Secondo, the system is named Distributed Secondo [28] [27]. Distributed Secondo and the single computer Secondo offer an identical interface, i.e., the operators and data types can be used in a parallel manner.

Secondo and Cassandra are coupled loosely. This simplifies software upgrades and the use of newly developed data models or operators in a parallel manner way without modification. Furthermore, all the details about the distribution are encapsulated within one software component, which is referred to as *transparency* in the literature [30, p. 7-13] [38, p. 4-5].

4.1 System Architecture

Distributed Secondo is a distributed system; it consists of three node types: (*i*) *Management nodes* (MNs); (*ii*) *Storage nodes* (SNs); and (*iii*) *Query processing nodes* (QPNs). An overview of the data flow and the components of Distributed Secondo is shown in Figure 2.

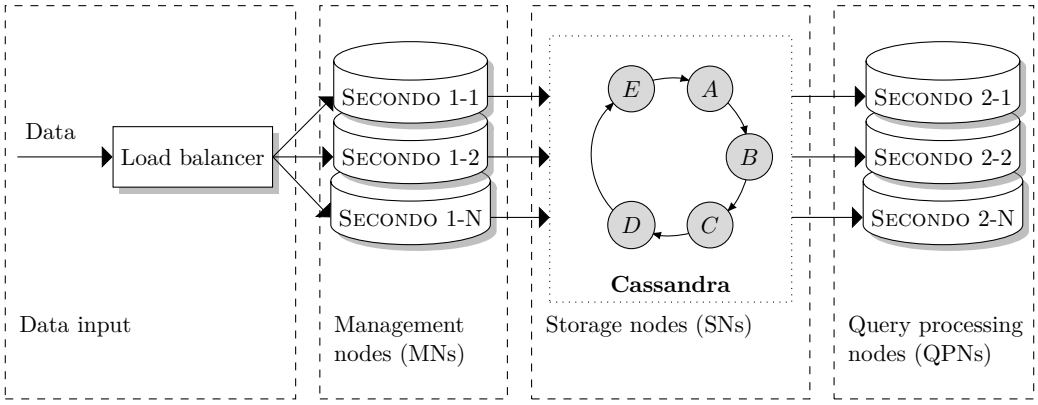


Figure 2: An overview of the data flow in Distributed Secondo. New data is processed by a load balancer. The load balancer spreads the data over multiple MNs; these MNs process the data and store them on the SNs. Afterwards, the data can be analyzed on the QPNs.

4.2 Management Nodes

Management nodes are running instances of Secondo. They are used for importing and exporting data into and from Distributed Secondo. The data can be read from a file or from a network socket. During the import process, the data can be converted into appropriate data types. Also other operations can be applied, for example the input data can be filtered and only certain data of the input can be processed.

The Management nodes are also used to specify the queries that should be executed by the QPNs. To achieve that, the Cassandra algebra (see Section 5) provides a set of operators. By means of the operators **cqueryreset** and **cqueryexecute** a *query execution plan* (QEP) is created. The QEP is an ordered list of the queries that should be executed. The operator **cquerywait** allows to wait for the completion of a certain query of the QEP. The operator blocks until the specified query has completed. In addition, the operator provides a progress estimation [17] for the current query. This feature is currently only supported for *complex queries* (see Section 4.4.2). The progress of a complex query is simply calculated by comparing the amount of completed UOWs compared to the amount of total UOWs ($progress = \frac{completed\ units\ of\ work}{total\ units\ of work}$).

4.3 Storage Nodes

Storage nodes run Cassandra in order to store data. The MNs and the QPNs read and write data onto these nodes. To achieve this, the kernel of SECONDO is extended by an algebra module. The algebra module contains all operators that realize the communication between SECONDO and Cassandra. The algebra is called CassandraAlgebra and its details will be discussed more precisely in Section 5.

4.3.1 Storing SECONDO Relations

The operator **cspread** is part of the CassandraAlgebra. **cspread** is used to export SECONDO relations onto the storage nodes. For each exported SECONDO relation, the operator creates a corresponding relation in Cassandra with four columns. Table 2 shows a SECONDO relation that is stored in Cassandra.

partition	node	key	value
65611	node1	17	`@()!3ds4f/§=!fgf())@jhsnmsirJZnbfgd[..]`
45454	node3	91	`d6d4/@3s3463Hbjfdgjn32[..]`
43543	node4	513	`ds73?6C32NUjhfnmri=*fkfjnNdier[..]`
_TPT	_TPT	_TPT	`(stream (tuple ((Id int) (Name string))))`

Table 2: A SECONDO relation stored in Cassandra. Each value field of the first tuples stores a SECONDO tuple encoded as a sequence of bytes. The fourth tuple stores the type of the SECONDO relation.

For each tuple in the SECONDO relation the operator inserts a tuple into the Cassandra relation. The SECONDO tuples are converted into a byte stream and stored in the *value* field of the relation. In addition, the operator determines the type of the relation, e.g. the columns and types, and stores this information in a special tuple in a nested list format[2]. This special tuple is indicated by a value of *_TUPLETYPE* in the first three columns (_TUPLETYPE is shortened as _TPT in the example). The relation type is needed to restore the relation in SECONDO; this can be done using the operators **ccollect**, **ccollectrange** and **ccollectquery**. Except the tuple for the relation type, the values of the columns *partition*, *node* and *key* are calculated as follows:

partition: This field is the primary key of the Cassandra relation. The data model of Cassandra specifies that the primary key determines the position of the tuples in the logical ring. The operator **cspread** requires to specify a partitioning attribute. For each tuple, the hash value of the partitioning attribute determines the value of this field. The concept of data partitioning will be discussed in the next section.

node: This field contains a freely selectable identifier. The identifier is one of the parameters of the **cpread** operator. Often this value is used to identify the system which has created this tuple.

key: This is a consecutive number for each stored tuple. The fields *node* and *key* are declared as clustering columns in Cassandra. This requires, that the combination of these two values and the primary key are unique.

The value of the partition column is determined by a hash function. The values of this column are not unique. If the relation contains tuples with an identical value of the partitioning attribute, the hash value of both tuples is identical too. To get an unique value, the key column is taken into consideration. The field contains a consecutive number and ensures the required uniqueness.

[2]Nested lists are used in SECONDO to represent structured data. For example: ((`value1 value2`) (`value3`))

4.3.2 Partitioning Data

Query processing nodes request data from the SNs at two different quantities: (*i*) a whole relation; or (*ii*) one or more token ranges. Which quantity is requested depends on the query. The first size is usually used, when a (small) relation needs to be made available on each QPN. The second size is used, when a big relation is split up into smaller chunks and needs to be processed in parallel. In this mode, the QPN fetches a data chunk from the SNs, executes the specified query and writes the result back to the SNs. The data chunks will be further called units of work. The structure of the logical ring determines the size and the number of available UOWs.

Some functions, e.g. joins, can be executed more effectively if the UOWs are partitioned considering the structure of the data. For example, an equi-join of the relations A and B should be executed. The join operator reads a UOW of relation A. After that, the operator needs to find all corresponding tuples in relation B. If the UOWs are created without considering the structure of the data, all UOWs of the relation B have to be read. If the units are partitioned considering the join attribute, only the UOW containing the join attribute has to be read.

For partitioning spatial data in Secondo, a grid can be used. The number of the cell determines the position where the data is stored within the logical ring of Cassandra. Such kind of partitioned data can be processed with join algorithms like the *Partition Based Spatial–Merge Join* (PBSM-Join) [32]. An example how spatial data is stored in Distributed Secondo is shown in Figure 3.

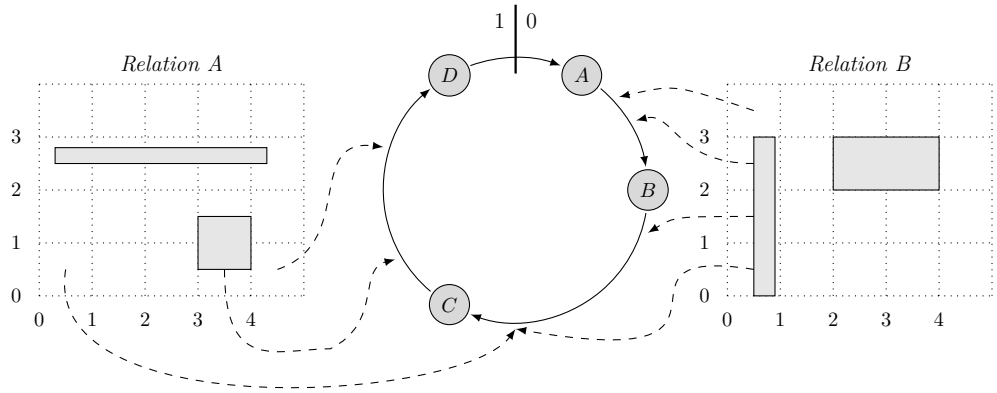


Figure 3: Two relations with spatial data stored in Cassandra. The spatial data is overlapped with a grid. The content of corresponding cells in both relations are stored at the same position in Cassandra.

4.4 Query Processing Nodes

The query processing nodes (QPNs) are capable of the computation of the queries. To carry out their work, they run one or more threads of the DBMS Secondo. Because Secondo is developed as a single CPU DBMS, it can only utilize one CPU core. With the use of multiple Secondo threads, all CPU cores of the hardware can be used.

It's not required to execute the software for SNs and QPNs on different hardware nodes; the hardware for SNs and QPNs can be equivalent. The work load for the SNs is mostly IO bound; the work load for QPNs is mostly CPU bound. By running both node types on the same hardware, all components can be utilized.

4.4.1 The QueryExecutor

The QueryExecutor is the link between Cassandra and Secondo (as depicted in Figure 4). The software encapsulates all the details about distribution, fault tolerance and distributed query execution. One instance of this program is executed on each QPN. The QueryExecutor fetches the

queries that have to be executed, distributes the queries to the SECONDO threads and does some additional tasks, e.g. sending heartbeat messages or printing information about the status to the console. The software is written in C++ and uses the DataStax *cpp-driver* [8] for the communication with Cassandra.

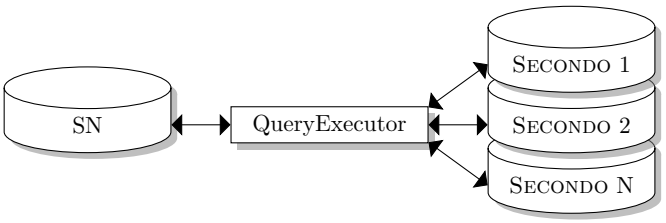


Figure 4: The QueryExecutor is the link between a SN and a QPN node.

As described before, the software for the SNs and for the QPNs can run on the same hardware. Therefore, it is also possible to use dedicated hardware for both node types. The software creates a TCP connection to Cassandra and a TCP connection to each SECONDO-Instance. This enables a high flexibility for the structure of the whole system.

4.4.2 Query Types

Some queries of the QEP can be executed equally on all QPNs. Other queries should split the input data into UOWs and process the data in parallel. For that reason, DISTRIBUTED SECONDO divides the queries of the QEP into two different kinds:

Simple queries: These queries are executed equally on all QPNs. This type of queries is used to create or open databases or to import (small) relations on the QPNs.

Complex queries: This kind of queries is executed on different parts of the input data in parallel. DISTRIBUTED SECONDO ensures that each QPN executes the query on a different UOW. In addition, the system guarantees that each UOW is processed completely, even in a case of failure of a QPN. To achieve that goal, DISTRIBUTED SECONDO can assign more than one UOW to a QPN.

Basically the computation of a complex query consists of three parts: (i) the import of the required data from the storage nodes; (ii) the computation of the query; and finally (iii) the export of the result of the computation to the storage nodes.

The distinction between these two types of queries is done automatically. Queries that will process data in parallel have to contain the operator **ccollectrange**. Each query that contains this operator is a complex query, all other queries are classified as simple queries.

4.5 System Tables

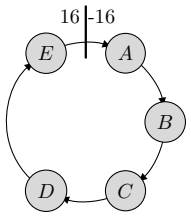
DISTRIBUTED SECONDO creates a few system tables in Cassandra. These tables are used to track the progress of the query processing, to store the QEP and to save the structure of the logical ring.

For instance, it is required to store the information about the logical ring in one table. That ensures, that each QPN has the same view on the logical ring. This is necessary, because the logical ring can change its structure (e.g. due to failing or newly started storage nodes). If two QPNs would have different views on the structure of the logical ring, they would make different decisions, i.e. the creation of mismatched UOWs, which leads to incorrect query processing.

4.5.1 The Example Environment

To reduce complexity in the examples that follow, the logical ring has been restricted to four nodes within a token range of $[-16, 16]$. However, in a real world environment there can be thousands of

nodes and tokens are a 128 bit signed integer, covering the range $[-2^{63}, 2^{63}-1]$. In the example environment, five SNs (A, B, C, D, E) are used. Each SN is also used as a QPN. The specific token ranges of the SN and the IP addresses can be found in Figure 5.



Node	IP	Token	Token ranges
A	192.168.0.1	-13	$[-16, -13]$, $(13, 16]$
B	192.168.0.2	-6	$(-13, -6]$
C	192.168.0.3	-2	$(-6, -2]$
D	192.168.0.4	5	$(-2, 5]$
E	192.168.0.5	13	$(5, 13]$

Figure 5: The used example environment in this paper.

4.5.2 The Structure of the Logical Ring

It is required that all QPNs have the same information about the structure of the logical ring. To ensure this, the structure of the logical ring is stored in one system table. The operator **cqueryinsert** writes this information into the system table `system_tokenranges`, as soon as the first query is inserted into the QEP. Table 3 shows an example of the system table `system_tokenranges`.

begintoken	endtoken	ip
13	16	192.168.0.1
-16	-13	192.168.0.1
-13	-6	192.168.0.2
-6	-2	192.168.0.3
-2	5	192.168.0.4
5	13	192.168.0.5

Table 3: The table system_tokenranges.

4.5.3 The Query Execution Plan

The *Query Execution Plan* determines which queries need to be executed by Distributed Secondo. The plans are created on the MNs and stored on the SNs. The data replication of Cassandra ensures that no failing SN can prevent the access to it. The QEP is stored in the system table `system_queries`.

The QEP contains a set of normal query plans for Secondo. Every query is associated with a number, to ensure that the queries are executed in the correct order. In addition, each query is associated with a version. The version information is used, to determine changes in the QEP. Currently the version of a query is simply the time stamp when the query was inserted into the QEP. Table 4 shows the example content of the table `system_queries`.

id	version	query
1	1433770969	`create database db1;`
2	1433770969	`open database db1;`
3	1433770971	`query [readparallel plz] count feed namedtransformstream[cnt] [write countresult cnt];`

Table 4: The content of the table system_queries.

4.5.4 Details About the QPNs

The system table `system_state` contains information about the QPNs, the query processing state and the used hardware. The column *ip* contains the IP of a specific QPN. The column *heartbeat* contains a heartbeat value for this QPN. Distributed Secondo uses this value to determine if a QPN is dead or alive. The column *query* contains the ID of the last successfully completed query on this QPN. This id corespondents with the query id in the QEP. The column *node* contains an unique value for this QPN. This value is generated by the QueryExecutor upon initialization.

The last three columns are used for information purposes. The column *memory* contains the amount of memory of this QPN, the column *cputype* contains the available CPU type. The field *threads* indicates how many Secondo threads are running on this node. Table 5 shows an example of the system table.

ip	heartbeat memory	query threads	node cputype
192.168.0.1	1462273445 7733	2 4	0565fb2b-06a6-46cb-91b9-cf8ddf85ac2d AMD Phenom II X6 1055T
192.168.0.2	1462273447 7733	2 4	7537f4c5-a954-4297-a8e6-6213f030f979 AMD Phenom II X6 1055T
192.168.0.3	1462273441 7733	2 4	32ddfcde-2635-4bee-8527-bc0eb97e18db AMD Phenom II X6 1055T
192.168.0.4	1462273443 7733	2 4	676cb8e1-a118-4d44-8922-f45824b975fb AMD Phenom II X6 1055T
192.168.0.5	1462273445 7733	2 4	ed3f2dc2-fd42-4fef-8218-80561a33d309 AMD Phenom II X6 1055T

Table 5: The table system_state.

4.5.5 Details About the Execution of Queries

Distributed Secondo places information about the execution of complex queries in the system table `system_progress`. Each QPN that processes an UOW, inserts a tuple after the completion of the UOW into this table. An example of the table is shown in Table 6.

The QueryExecutor generates a *Universally Unique Identifier* (UUID) [25] for each executed query. When a complex query has to compute multiple UOWs on a QPN, each UOW is processed with another UUID. As mentioned in Section 4.3.1, each stored tuple on the SN is equipped with three fields (partition, node and key) to identify the tuple. When the result of a complex query is stored on the SN, the value of the UUID is used as the value for the node field, to ensure, that the result is equipped with a unique identifier. The operator **ccollectquery** uses the data of this table, to read a result of a complex query.

id	begin	end	ip	queryuuid
2	13	16	192.168.0.1	163eced6-0c36-4685-8fd5-4e34876241e2
2	-16	-13	192.168.0.1	737212b7-0190-4432-b68f-8fbefb8cce9c
2	-13	-6	192.168.0.2	f7832156-1c20-49b9-b0cb-cd4c1c99f890
2	-6	-2	192.168.0.3	19e9558f-0cec-4963-9b4a-32c227c5d989
2	-2	5	192.168.0.4	fc664f90-5a56-4712-ac7a-a9195bb60f87
2	5	13	192.168.0.5	985326c9-6724-4ddb-8fc9-1e88be5a6759

Table 6: The table system_progress.

In addition, a second system table with the name `system_pending` is used to track which QPN processes which UOW at the moment. QPNs can use this table to find out which UOWs are processed right now.

id	begin	end	ip	queryuuid
2	13	16	192.168.0.1	163eced6-0c36-4685-8fd5-4e34876241e2
2	-13	-6	192.168.0.2	f7832156-1c20-49b9-b0cb-cd4c1c99f890
2	5	13	192.168.0.5	985326c9-6724-4ddb-8fc9-1e88be5a6759

Table 7: The table system_pending.

4.6 A Graphical User Interface

The system tables contain the information about the state of the whole system. To get information about the progress of a query and the state of the QPNs, it is cumbersome for a user to analyze these tables manually. In order to provide some comfort for the user, Distributed Secondo ships with a *Graphical User Interface* (GUI) which visualizes most of the data of the system tables. A screenshot of the GUI is shown in Figure 6.

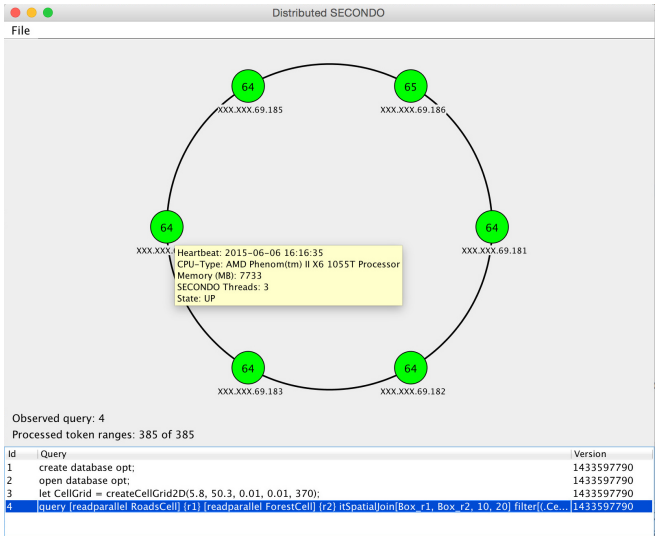


Figure 6: The Graphical User Interface of Distributed Secondo to view the state of the QPNs.

In the lower part of the GUI, the QEP is shown. The upper part shows the logical ring of Cassandra and the state of the nodes. Because we use the same hardware for SNs and QPNs, the nodes are drawn *green* when Cassandra and the QueryExecutor run on the system without any errors. This is indicated by up to date heartbeat messages in the system table. Nodes are drawn *red* when the node started the query processing and the heartbeat messages are outdated. This situation can be caused by a system crash during query execution. Nodes are drawn *yellow* when they are only used as SNs and no query processing is carried out on them. It is possible to select the queries of the QEP. Once a query has been selected, the amount of processed UOWs is shown inside the node. The data is updated every few seconds, this makes it possible to watch the progress of a certain query in real time. A tooltip will appear when the mouse is paused over a node; details about the node will be shown.

5 Query Formulation

In this section, the CassandraAlgebra is described. This algebra extends the kernel of Secondo with the operators which are needed to interact with Cassandra. At this point, it is assumed that

Distributed Secondo is installed properly, the QPNs and the SNs are running and the Secondo system is build with the CassandraAlgebra; the required steps are described on the website of the project [41].

Besides of the parameter of the operators, the section `[CassandraAlgebra]` of the Secondo configuration influences the behavior of the algebra. This section is used to define the address of the SN to communicate to and to set some default values (i.e. the consistency level).

5.1 The Operators of the CassandraAlgebra

The CassandraAlgebra provides all the operators shown in Table 8.

Operator	Signature	Syntax
Tuple Management:		
`cspread`	stream(tuple(...)) × text × attr [× text [× text]] → int	`_ # [_,_,_,_]`
`ccollect`	text [× text] → stream(tuple(...))	`# (_,_)`
`ccollectrange`	text × text × text [× text] → stream(tuple(...))	`# (_,_,_,_)`
`ccollectquery`	text × int [× text] → stream(tuple(...))	`# (_,_,_)`
`clist`	→ stream(text)	`# ()`
`cdelete`	text → bool	`# (_)`
Query Processing:		
`cqueryexecute`	int × text → bool	`# (_,_)`
`cquerylist`	→ stream(tuple(...))	`# ()`
`cqueryreset`	→ bool	`# ()`
`cquerywait`	int → bool	`# (_)`

Table 8: Operators and the syntax of the CassandraAlgebra.

The operators can be separated into two groups: (*i*) The first group is responsible for the communication with the SNs. Relations can be exported to the SNs and imported back. In addition, the stored relations can be listed and deleted. (*ii*) The second group is used to manage the query processing. With these operators, the QEP can be created, listed and deleted. The table also contains the syntax of the operators. The symbol _ denotes the places of the arguments of the operator, the symbol `#` denotes the place of the operator.

5.1.1 cspread

The Operator **cspread** is used to export tuples from Secondo onto the SNs. The operator requires at least two parameters; the third and the fourth parameter are optional. (*i*) The name of the table which is used to access the data later. (*ii*) The name of the partitioning attribute. (*iii*) The name of the current node and (*iv*) the desired consistency level to write the data. The default values for the third and the fourth parameter are set in the configuration of the algebra.

The name of the node is a freely selectable identifier which is required to handle parallel write requests on one table in Cassandra. The meaning of the partitioning attribute was already discussed in Section 4.3.2.

Example: Spread the content of the Secondo relation plz onto the storage nodes. The data is partitioned using the PLZ attribute. In Secondo the postfix syntax is used, hence the operator **cspread** is applied to the output of the operator **feed**.

```
query plz feed cspread['plz', PLZ];
```
(1)

5.1.2 ccollect, ccollectrange and ccollectquery

These three operators are used to import data from the storage nodes back into SECONDO. The operator **ccollect** reads a whole relation, the operator **ccollectrange** reads only the tuples of a relation, which are stored in a specified range of the logical ring. The operator **ccollectquery** reads the result tuples of a complex query. The number of parameters are different. The first parameter is always the name of the relation to be read. The last parameter is the consistency level used for this operation. This parameter is optional.

The operator **ccollect** requires only these two parameters. The operator **ccollectrange** uses two further parameters to determine the range of the logical ring to read to. The operator **ccollectquery** uses one additional parameter, the id of the query of the QEP which has produced the relation.

Example: Import the whole relation plz back into SECONDO:

```
let plz = ccollect['plz'];
```
(2)

Example: Import only the part of the relation plz into SECONDO, which is stored in the token range [−10000,
10000]:

```
let plz = ccollectrange['plz', '-10000', '10000'];
```
(3)

Example: Import the relation joinresult into SECONDO. The relation was produced by the query 8 of the QEP:

```
let joinresult = ccollectquery['joinresult', 8];
```
(4)

5.1.3 clist and cdelete

Both operators are used to manage the stored relations. The operator **clist** lists the relations that are currently stored on the SNs. This operator has no parameter and produces a stream of the names of the stored relations. The operator **cdelete** deletes a stored relation. The operator requires one parameter; the name of the relation that should be deleted.

Example: Show all stored relations onto the SNs:

```
query clist() transformstream consume;
```
(5)

Example: Delete the relation plz:

```
query cdelete('plz');
```
(6)

5.1.4 cqueryexecute, cquerylist and cqueryreset

These three operators are used to manage the QEP. The operator **cqueryexecute** inserts a query into the QEP. The operator expects two parameters: (*i*) A number of the position where the query should be inserted and (*ii*) the query. The number 1 identifies the first query. All the queries are stored in the system table `system_queries` (see section 4.5.3).

Example: Specify the first query of the QEP:

$$\underbrace{\texttt{query cqueryexecute(2,}\overbrace{\texttt{'open database db1;'}}^{\text{the QEP query}}\texttt{);}}_{\text{the insert query}}$$

The query above contains the query that should by executed. In the sequel, the full query will be called *the insert query*. The query that should be inserted into the QEP will be called *the QEP query*.

The operator **cquerylist** shows all queries of the QEP. This operator does not need any parameters.

Example: Show all the queries contained in the QEP:

```
query cquerylist() consume;
```
(7)

The operator **cqueryreset** deletes the QEP, in addition the system tables are recreated. Like the last operator, this operator does not expect any parameters.

Example: Create a new QEP:

```
query cqueryreset();
```
(8)

5.1.5 cquerywait

This operator can be used, to wait for the complete execution of a particular query of the QEP. The operator **cquerywait** expects one parameter; the id of the query to wait for. A query which includes this operator (*the wait query*) will run until Distributed Secondo has completed the specified query of the QEP (*the QEP query*). After the wait query finishes, the result of the QEP query can be fetched from the SNs.

Example: Wait until the query 3 is completed:

```
query cquerywait(3);
```
(9)

5.2 Placeholders

The QueryExecutor needs to change the queries in the QEP to ensure that each query processing node fetches the proper data from the storage nodes. To achieve that, placeholders are used. A placeholder starts and ends with two underscores. The placeholder is replaced by the QueryExecutor with a concrete value before the query is sent to Secondo. Currently the placeholders described in table 9 are supported.

Placeholder	Description
__CASSANDRAIP__	The IP address of the storage node with which the QuerxEyecutor is connected.
__KEYSPACE__	The name of the keyspace with which the QueryExecutor is connected.
__NODEID__	A unique identifier for this query node.
__TOKENRANGE__	The token ranges for which this query processing node is responsible for.
__QUERYUUID__	A unique identifier for every statement executed in Secondo.

Table 9: Supported placeholders and their meaning in queries.

5.3 Shortcuts

Shortcuts are used to simplify the formulation of QEPs. QEPs using the CassandraAlgebra can become large and complex. The shortcuts allow the formulation of queries that are shorter and easier to understand. The shortcuts are replaced by the QueryExecutor with concrete operators (see Table 10). Each shortcut exists in a second version, which accepts a consistency level as additional parameter.

Shortcut	**Replacement**
`[read tablename]`	`ccollect('tablename')`
`[readparallel tablename]`	`ccollectrange('tablename', __TOKENRANGE__)`
`[write tablename attr]`	`cspread('tablename', attr, '__QUERYUUID__')`

Table 10: Supported shortcuts and their replacements.

Listing 1 shows a query in two versions; one version with and one version without shortcuts. The version with shortcuts is shorter and easier to understand. In the query without shortcuts, the QEP query needs to be quoted with the elements `<text>` and `</text--->`. They are used to indicate the start and the end of the QEP query. This special form of quoting is needed, because the query contains quotation marks. By using shortcuts, the quotation marks are eliminated inside of the QEP query and the begin and the end can be indicated by quotation marks.

Listing 1 Shortening a query

```
1 # Version without shortcuts
2 query cqueryexecute(2,<text>ccollectrange ('table1', __TOKENRANGE__) count feed
3    namedtransformstream[cnt] [write countresult cnt];</text--->);
4
5 # Version with shortcuts
6 query cqueryexecute(2,'[readparallel table1] count feed
7    namedtransformstream[cnt] [write countresult cnt];');
```

6 Query Execution

This section describes all the techniques that are implemented in Distributed Secondo to execute queries and to handle faults during the query execution. The first subsection defines the logical ring formally. The second subsection describes the way how Distributed Secondo creates UOWs. In the subsequent sections, the algorithm for UOWs distribution is presented and some examples are given.

6.1 A Formal View on the Logical Ring

In Section 4.3, a rough overview of the logical ring has been given. In this section, this view is deepened and the characteristics are formally described. A *logical ring* L is a 4-tuple $(begin, end, SN, \tau)$, consisting of:

- b$egin, end$ are two integers which determine the range of the logical ring ($begin, end \in \mathbb{Z}$ and $begin < end$). The integers between *begin* and *end* are called *tokens* T.
- SN is a set of nodes called the *storage nodes* (SNs).
- τ is a function that assigns each $s \in SN$ one or more tokens ($\tau\colon SN \rightarrow 2^T$). These are the positions where the SNs are located in the logical ring.

The tokens of a logical ring are simply all integers between *begin* and *end*.

$$T := \{t \mid t \in \mathbb{Z} \wedge begin \leq t \leq end\} \tag{10}$$

A token is assigned to one particular SN exclusively:

$$s_1 \neq s_2 \Rightarrow \tau(s_1) \cap \tau(s_2) = \emptyset \text{ for } s_1, s_2 \in SN \tag{11}$$

The set P (*storage node positions*) is an ordered set of all tokens, that are assigned to the SNs:

$$P := \bigcup_{s \in SN} token(s) \tag{12}$$

τ^{-1} is the inverse function of τ. This function maps each $p \in P$ to a SN. Figure 7 depicts a logical ring of the range [-16,16] with four SNs $SN = \{s_0, s_1, s_2, s_3\}$.

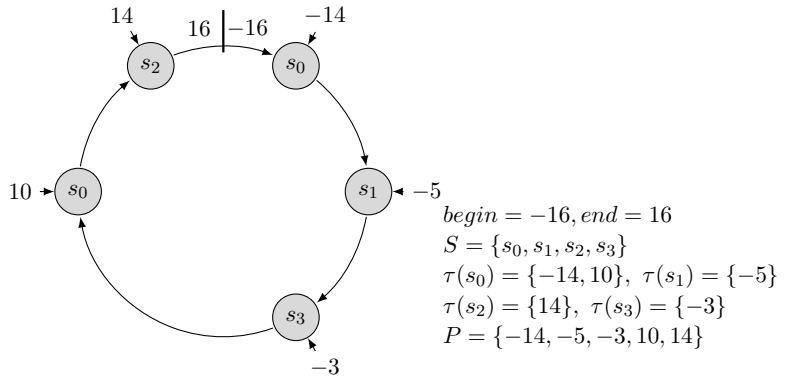


Figure 7: An example of a logical ring.

We call $r_{[a,b]}$ a *token range*, which contains all the tokens t from the interval $[a, b]$.

$$r_{[a,b]} := \{t \mid t \in \mathbb{Z} \wedge a \leq t \leq b\} \tag{13}$$

The positions of the SNs can be used to partition the logical ring into token ranges. The token ranges of this partitioning will be called the *canonical token ranges*. A logical ring with P = $(p_0, p_1, p_2, \ldots, p_n)$ divides into the token ranges:

$$\begin{aligned} CTR := \{&r_{[begin,p_0]}, r_{[p_0+1,p_1]}, r_{[p_1+1,p_2]}, \ldots, \\ &r_{[p_{n-1}+1,p_n]}, r_{[p_n+1,end]}\} \end{aligned} \tag{14}$$

A SN is responsible for all canonical token ranges which end at a position, where the SN is located in the logical ring. An exception is the last token range ($r_{[p_n+1,end]}$); the node with the lowest position ($\tau^{-1}(p_0)$) is responsible for this range. Each element of CTR is also called a *unit of work* in this paper[3]. The main use of CTR is the distributed generation of units of work. Each SN can calculate the set CTR only with information about the size of the logical ring and the position of the SNs. This information is already available on the SNs. In addition, each SN can determine for which token ranges it is responsible. In a logical ring, the SN s is responsible for all the token ranges that are included in R_s.

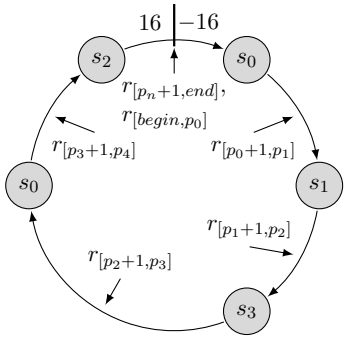
$$R_s := \{r_{[a,b]} \mid r_{[a,b]} \in CTR \wedge (\overbrace{(b \in \tau(s))}^{\text{Token range ends at node } s} \vee \underbrace{(a = p_n + 1 \wedge p_0 \in \tau(s))}_{\text{The first node is responsible for the last token-range too}})\} \tag{15}$$

Figure 8 depicts an example for a logical ring partitioned into its CTR and for the sets R_s.

To refer to the successor of a token range r, the function $succ(r)$ is used.

$$\begin{aligned} succ&: CTR \to CTR \\ succ(r_i) &:= r_{i+1 \ mod \ |r|} \end{aligned} \tag{16}$$

[3]The two cases $begin = p_0$ and $end = p_n$ are ignored in the description to keep the examples clear.



$$R_0 = \{r_{[p_n+1,end]}, r_{[begin,p_0]}, r_{[p_2+1,p_3]}\}$$
$$R_1 = \{r_{[p_0+1,p_1]}\}$$
$$R_2 = \{r_{[p_3+1,p_4]}\}$$
$$R_3 = \{r_{[p_1+1,p_2]}\}$$

Figure 8: A logical ring, partitioned into its canonical token ranges. The content of the set R_n for the four nodes are shown.

6.2 Assigning Units of Work to the QPNs

To process complex queries, UOWs need to be assigned to the QPNs. Every QPN has to fetch one or more UOWs and execute the query on the UOWs. The algorithm for the UOW assignment should meet the following requirements: (*i*) Every UOW should be assigned to a QPN. (*ii*) Crashes of QPNs have to be handled properly. (*iii*) No central coordinator should exist.

The definition of R_n allows each SN to calculate the set of local token ranges. This calculation can be performed only with local information. Therefore, the last condition (*iii*) is fulfilled. In this section, an algorithm is presented that uses this set as a foundation to meet the other two requirements. As mentioned earlier, the same hardware is used for QPNs and SNs. To indicate, that a SN s is running and executing queries (the node acts as a QPN), the function $alive(s)$ will be used.

$$alive\colon SN \to \{true, false\} \tag{17}$$

With the function above, it is possible to define the set QPN. This set contains all QPNs that are running faultless.

$$QPN := \{s \mid s \in SN \wedge alive(s)\} \tag{18}$$

For each $q \in QPN$, the set FR_q (foreign ranges) contains all the token ranges that belong to the other faultlessly running QPNs.

$$FR_q := \bigcup_{s \in SN \setminus \{q\}} R_s \tag{19}$$

The last definition in this section defines all the token ranges, which the QPN q is responsible for.

$$RR_q := \{r_{[a,b]} \mid responsible_q(r_{[a,b]})\} \tag{20}$$

The function $responsible_q(r)$ is defined as follows.

$$responsible_q\colon CTR \to \{true, false\}$$
$$responsible_q(r) := \begin{cases} true, & \text{if } r \in R_q \\ true, & \text{if } succ(r) \in RR_q \\ & \wedge\ r \notin FR_q \\ false, & otherwise. \end{cases} \tag{21}$$

The basic idea of the function $responsible_q(r)$ is to enlarge the set R_s. Starting at each token range, for which the node is responsible for, the adjacent token ranges are examined counterclockwise. The token range r is added to the set RR_q if no other QPN is responsible for it ($r \notin FR_q$). This process is continued until another QPN is responsible for the next token range. According to the definition of FR_q, this set contains only token ranges of nodes that are alive. FR_q is recalculated every few seconds, to ensure that node failures are handled properly.

Example: We have two QPNs q and m that are alive. The QPN q calculates the set RR_q. The local token ranges of QPN m are counterclockwise adjacent to q. As soon as the node m dies, the function $alive(m)$ evaluates to $false$. As a consequence, the local token ranges of m are removed from FR_q. This causes the adjacent token ranges of m to be added to RR_q.

Assigning and executing UOWs: It is now possible to formulate an algorithm that fulfills the three requirements. The basic idea of the algorithm is, to split up the problem of UOW assignments into three phases:

- Phase 1 (Processing of the local token ranges): In the first phase, each QPN q processes the UOWs that are contained in the set R_q.
- Phase 2 (Processing of adjacent token ranges): The adjacent UOWs will be processed. These UOWs are the units of the set RR_q.
- Phase 3 (Processing of random token ranges): A random number generator is used to determine which unprocessed UOWs should be processed.

  Before the UOW is processed, the system table `system_pending` is read to ensure that no other alive QPN is processing the same UOW already. If an other QPN is processing this QPN at the moment, an other unprocessed UOW is selected[4].

6.3 Examples

In the following sections, the function of the algorithm is shown in three examples. In the first two examples, no node failures are occurring. The last example shows, how the failure of one QPN is handled. All the examples refer to the demonstration environment, which was introduced in Section 4.5.1. To simplify the examples, it is assumed that all nodes are working at the same speed and QPNs are updating their heart beat values (HB) exactly every 10 seconds.

6.3.1 Example 1 – No Failing QPNs

In the first example, each SN also executes the software components for query processing; the set of SNs and QPNs are equal. During the query execution, no faults occur. In the first step (Figure 9) each QPN processes the local token ranges. The second step is also the last step (Figure 10), all UOWs are processed. In this example, only the first phase of the algorithm is needed.

6.3.2 Example 2 – Two QPNs

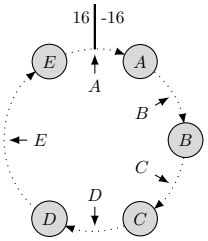
In the second example, the set of QPNs and SNs are different. The nodes A and D are QPNs and SNs, the nodes B, C and E are storage nodes only.

Step 1: The nodes A and D process their local token ranges (Figure 11).

Step 2: The adjacent token ranges will be processed. Because the nodes B and E do not have inserted heart beat entries into the corresponding system table, the function $alive(n)$ does evaluate to $false$ for these nodes. As a consequence, the token ranges of these nodes are processed by the nodes A and D (see Figure 12).

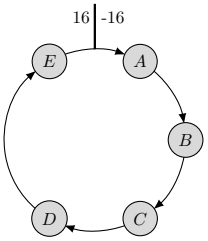
Step 3: In this step, the node A analyzes the token range $r_{[-1,5]}$. For this token range, the node D is responsible. This is indicated by up to date heart beat messages from the node D in the system table. The node A stops the processing at this point. The node D processes the token range $r_{[-5,-2]}$ (see Figure 13).

[4] Phase 3 is influenced by the *speculative task execution* of Hadoop [11, p. 3]. The table `system_pending` prevents, that all idle QPNs are processing the same UOW at the same time. This would lead to *hot spots* (parts of the logical ring that are read or written by many nodes simultaneously) and to longer query processing times.



Node	HB	Processed Token Ranges
A	10	–
B	10	–
C	10	–
D	10	–
E	10	–

Figure 9: Step 1: Each QPN processed the units of work of the local token ranges.



Node	HB	Processed Token Ranges
A	20	$r_{[14,16]}$, $r_{[-16,-13]}$
B	20	$r_{[-12,-6]}$
C	20	$r_{[-5,-2]}$
D	20	$r_{[-1,5]}$
E	20	$r_{[6,13]}$

Figure 10: Step 2: All units of work have been processed.

Step 4: In the last step, the node D is processing the token range $r_{[-12,-6]}$. After this token range has been processed, all the token ranges of the logical ring are processed (see Figure 14).

6.3.3 Example 3 – A Failing QPN During Query Execution

In this example, one QPN fails during query processing. This is noticed by the lack of new heart beat messages. This example continues the previous example, beginning after step 2. Deviating from the last example, in this example, the node D fails (see Figure 13).

Step 3a: In contrast to step 3 in the last example, the node D has failed. This is indicated by the absence of new heart beat messages. Because the QPN A is the last alive QPN, it becomes responsible for the unprocessed token ranges of node D (see Figure 15).

Step 4a: The QPN A is analyzing $r_{[-5,-2]}$. This token range has already been processed by QPN D. So no further action is needed (see Figure 16).

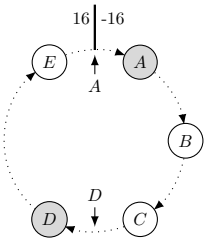
Step 5a: The QPN A is examining $r_{[-12,-6]}$. This token range is unprocessed and will be processed by QPN A (see Figure 17).

Step 6a: All token ranges of the logical ring have been processed completely (see Figure 18).

6.4 Reading Duplicate Free Results

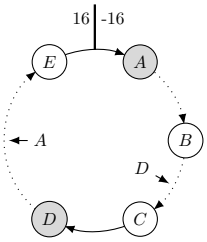
After all UOWs have been processed, the result of the query can be read. The result is usually stored into one relation. Reading the complete result relation can lead to problems, because it may contain duplicates. These duplicates arise in two situations: (*i*) a QPN is failing and the QPN has already written a part of the result into the result relation or (*ii*) in phase 3 of the algorithm, i.e., UOWs have been processed multiple times.

To get a duplicate free result, the operator **ccollectquery** can be used. As described already, the operator is using the system table `system_process` to determine which QPN has processed which UOWs completely. After a UOW has been completely processed, a corresponding entry is inserted into this system table by the QueryExecutor.



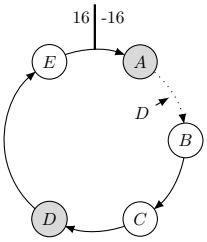
Node	HB	Processed Token Ranges
A	10	–
D	10	–

Figure 11: Step 1: The QPNs A and D are processing the units of work of the local token ranges.



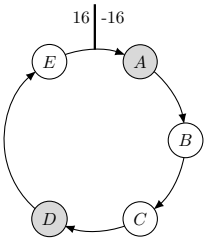
Node	HB	Processed Token Ranges
A	20	$r_{[14,16]}$, $r_{[-16,-13]}$
D	20	$r_{[-1,5]}$

Figure 12: Step 2: The QPNs A and D are processing the units of work of their neighbors' local token ranges.



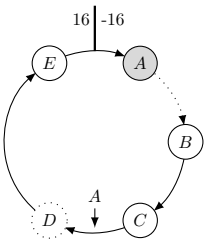
Node	HB	Processed Token Ranges
A	30	$r_{[6,13]}, r_{[14,16]}$, $r_{[-16,-13]}$
D	30	$r_{[-5,-2]}$, $r_{[-1,5]}$

Figure 13: Step 3: The QPN D is processing the units of work of the token range between node A and B.



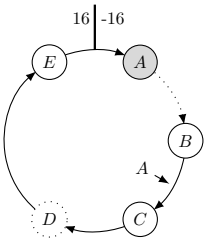
Node	HB	Processed Token Ranges
A	40	$r_{[6,13]}, r_{[14,16]}$, $r_{[-16,-13]}$
D	40	$r_{[-12,-6]}$, $r_{[-5,-2]}$, $r_{[-1,5]}$

Figure 14: Step 4: All units of work are processed.



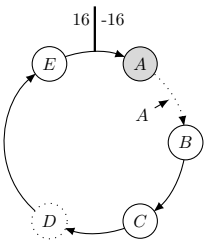
Node	HB	Processed Token Ranges
A	30	$r_{[6,13]}, r_{[14,16]}, r_{[-16,-13]}$
D	**20**	$r_{[-5,-2]}, r_{[-1,5]}$

Figure 15: Step 3a: The QPN D has failed. QPN A notices the failure by the missing heart beat updates. As a consequence, QPN A analyzes the token range between C and D.



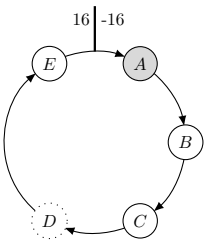
Node	HB	Processed Token Ranges
A	40	$r_{[6,13]}, r_{[14,16]}, r_{[-16,-13]}$
D	**20**	$r_{[-5,-2]}, r_{[-1,5]}$

Figure 16: Step 4a: The QPN A is analyzing the token ranges, until it finds an unprocessed one.



Node	HB	Processed Token Ranges
A	50	$r_{[6,13]}, r_{[14,16]}, r_{[-16,-13]}$
D	**20**	$r_{[-5,-2]}, r_{[-1,5]}$

Figure 17: Step 5a: The QPN A has found a non processed part of the logical ring.



Node	HB	Processed Token Ranges
A	60	$r_{[6,13]}, r_{[14,16]}, r_{[-16,-13]}, r_{[-12,-6]}$
D	**20**	$r_{[-5,-2]}, r_{[-1,5]}$

Figure 18: Step 6a: All units of work have been processed.

The fields *queryid*, *begin* and *end* are the primary key of the table. If two nodes have processed the same UOW, only one entry for this UOW is possible in that table. For example, the nodes A and B do insert an entry for the same UOW simultaneously, the last inserted entry overwrites the first inserted entry.

Reading the result: After all UOWs have been processed, the result of the query can be read. The operator **ccollectquery** reads from the table `system_process` all the UUIDs of the query executions that have produced a valid part of the query result. The operator requests all the tuples of the result relation which have a nodeid that is contained in the set of UUIDs. All other entries of the result relation are ignored. In Figure 19 the interaction of the components are depicted.

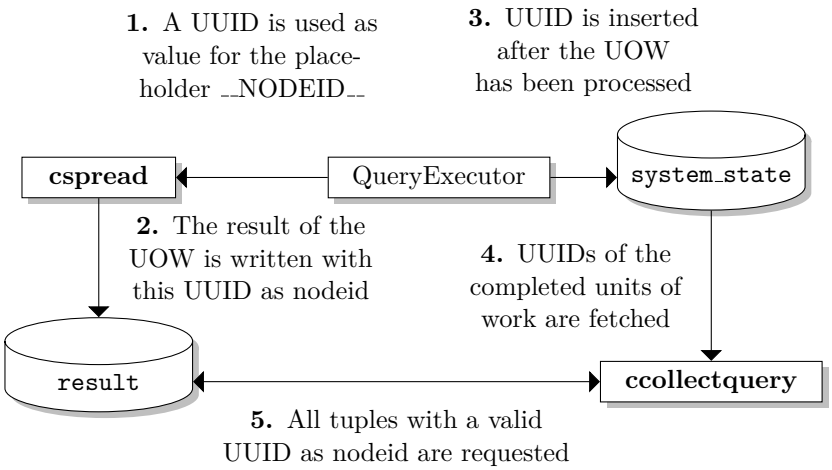


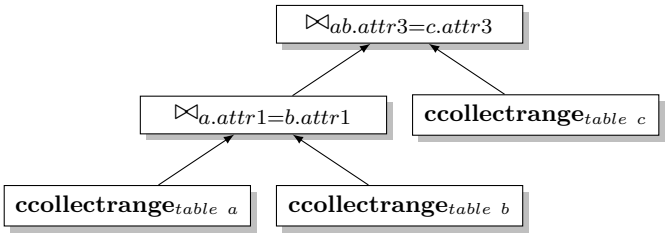
Figure 19: The interaction between the QueryExecutor and the operators cspread and ccollectquery.

6.5 Handling Write Errors During Query Execution

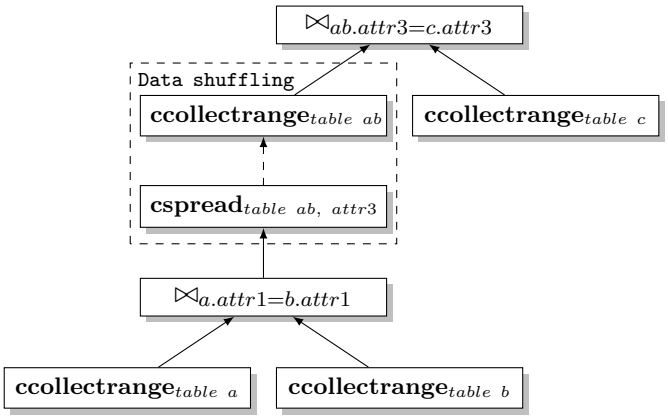
Even though Cassandra is a highly available system, it is possible that a write operation fails. This problem can be handled in two ways: (*i*) the data for the write operation is stored on the QPN until the write request is successfully confirmed. Failing write requests are retried. (*ii*) The complete calculation of the UOWs is repeated. The advantage of the first variant is, that no recalculation is needed. The drawback is, that a large amount of data needs to be cached. Write requests are performed asynchronously. This means, that about a few hundred simultaneous requests are performed concurrently. All the data for these requests need to be cached until they are finished successfully. Because write errors occur infrequently, Distributed Secondo implements the second variant. If a write error occurs, the whole UOW is recalculated. The detection of these errors is carried out by the QueryExecutor and the operator **cspread**. The operator produces an integer result with the amount of written tuples. If an error occurs, the operator produces a result of -1. The QueryExecutor analyzes each query before its execution. If the query uses the operator **cspread**, the QueryExecutor reexecutes this query if the result of the query is -1.

6.6 Data Shuffling Between Operators

When a distributed join operation is performed, only corresponding parts of the logical ring are analyzed. This is caused by the **ccollectrange** operator which requests these parts as data source for the join operator. For that reason, it is required that the data is partitioned regarding the join attribute. It is possible to cascade two or more join operators in one operator tree. But in this case, it is required, that all join operators use the same attribute for performing the join. When a different join attribute is used in the second join, the data needs to be reshuffled between the QPNs (as depicted in Figure 20).



(a) Invalid operator tree.



(b) Valid operator tree.

Figure 20: An invalid and a valid operator tree for two cascaded joins on different attributes. The first join operator (the lower one) joins the tables a and b on the attribute $attr1$. The second join operator joins the result of the first join with the table c on the attribute $attr3$.

Without the reshuffling step, the result of the join is invalid, because some result tuples may be missing. This is caused by the parallel calculation of the join. The second join operator reads a range of the logical ring and the result of the first join operator. Because the join attributes are different, the output of the first join operator is inappropriate for this computation.

The data shuffling is performed by the operators **cspread** and **ccollectrange**. The **cspread** operator spreads the result of the first join back onto the storage nodes. This step is executed by all QPNs. The computation of the second join is only started, when the first join is computed completely. It is important, that the spread data is partitioned using the join attribute of the second join operator. In Figure 20 (b) the second join operator uses two instances of the **ccollectrange** operators as data sources. The situation is exactly the same as for the first join operator, the **ccollectrange** operator reads corresponding parts of the logical ring and the join can be calculated.

7 Examples

In the following sections, some examples of Distributed Secondo are given. The first two sections describe a join on relational and spatial data respectively. The third section demonstrates how a data stream with high update rates can be handled with Distributed Secondo. In normal operation, it can be assumed, that the data of Distributed Secondo is still stored onto the SNs. However, to show complete examples and experiments the data is always spread in the first step and collected in the last step.

7.1 A Join on Relational Data

The first example shows a relational join on two relations. The first relation *customer* contains the customers of a company. The second relation *revenue* contains the revenue of the customers. A join on the attribute *customerid* should be used, to determine the revenue for the customers.

7.1.1 The Sequential Version

SECONDO contains a large amount of join operators. In this example, the operator **itHashJoin** is used. This operator performs the join based on a hashmap. The following query reads both input relations, generates a stream of tuples from the input and lets the operator **itHashJoin** process both tuple streams. Afterward, a stream of tuples with the join result is generated. The last operator **consume** collects the tuple stream and inserts the tuples into a new relation. The relation is stored with the name `join_result` on the hard disk of the local system.

```
let join_result = customer feed {r1} revenue feed {r2} itHashJoin
    [customerid_r1, customerid_r2] consume;
```
(22)

7.1.2 The Parallel Version

In this section, the same join is computed in a parallel way with DISTRIBUTED SECONDO. Because the join is performed on the attribute *customerid* the relations have to be partitioned according to this attribute. This task can be done by the queries of Listing 2.

Listing 2 Spread the data

```
1 query customer feed cspread ['customer', customerid];
2 query revenue feed cspread ['revenue', customerid];
```

Creating the QEP: Now, the QEP for the join has to be created. The QEP for the join contains three queries. The first query creates a database with the name `db1`. The second query opens this database. The two queries are needed to get SECONDO into a state where queries can be processed. The last query performs the join. Both relations are read in parallel by using the shortcut `[readparallel ...]`. The content of both relations is processed by the operator **itHashJoin** and stored into the relation `join_result` onto the SNs. The result relation is partitioned using the `customerid_r1` attribute.

Listing 3 The parallel join

```
1 query cqueryexecute(1,'create database db1;');
2 query cqueryexecute(2,'open database db1;');
3 query cqueryexecute(3,'[readparallel customer] {r1} [readparallel revenue] {r2}
4    itHashJoin [customerid_r1, customerid_r2]
5    [write join_result customerid_r1];');
```

Wait for the join to complete: The QPNs processes the QEP. After query 3 is completed, the result of the join can be read. The operator **cquerywait** is used, to await the completion. The operator blocks until the query is processed completely.

```
query cquerywait(3);
```
(23)

Reading the result of the join: In the last step, the result of the join is read from the SNs and stored on the local disk. To achieve this, the operator **ccolectquery** is used. As in the sequential version, the operator **consume** is used to convert the stream of tuples into a relation. The relation is stored with the name `join_result` on the local disk.

```
let join_result = ccollectquery('join_result', 3) consume;
```
(24)

7.1.3 Convert the Sequential Query

Query 3 of the QEP (line 3 in Listing 3) performs the join in a parallel way. This is simply a sequential join query (query number 22), modified for the parallel usage. The input relations *customer* and *revenue* are replaced by the shortcut `readparallel[...]` to read the data from the SNs and to split up the input into UOWs. In the sequential version, the result of the operation is stored on the local system in the relation *join_result* (`let join_result = ... consume`). These parts of the query are replaced by the `write[...]` shortcut. This ensures, that the result is stored onto the SNs, partitioned by the `customerid_r1` attribute. Figure 21 depicts the transformation of the query.

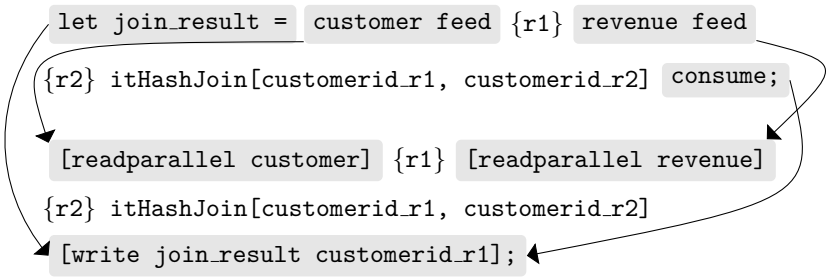


Figure 21: Transformation of a sequential query into a parallel one.

Generally, a sequential query (with a maximum of one join operator) can be easily converted into a parallel one in two simple steps: (i) The input relations must be replaced by the `readparallel[...]` shortcut. (ii) The output of the query must be consumed by the `write[...]` shortcut. In addition, the data must be stored onto the SNs and the parallel query must be placed into a QEP. If multiple joins occur in the sequential query, the data needs to be reshuffled between the join operations (see Section 6.6) and the sequential query needs to be split up into multiple ones.

7.2 A Join on Spatial Data

In this example, another join is performed. At this time, spatial data is used as input. Two relations with the geographic information of Germany are used. Both relations are created from data fetched from the *Open Street Map* [29] project. SECONDO ships with a script, that can download and import these data into a database. The relation *Roads* contains the roads of Germany, the relation *Buildings* all buildings. The spatial join is performed to find all the buildings that are crossed by a road. For example, a building on top of a road tunnel.

7.2.1 The Sequential Version

SECONDO contains a wide range of operators to perform a spatial join. The operator **itSpatialJoin** is the most efficient one. The operator builds up a *main memory R-tree* (*MMR-tree*) of the left relation and probes all tuples of the right relation against this tree. If the left relation does not fit into memory, only a partial MMR-tree is build in memory. The tuples of the right relation are probed against the MMR-tree and additionally stored onto disk. After the right relation is read completely, a new MMR-tree with the next part of the left relation is build. The stored version of the right relation is probed against the new MMR-tree. This is repeated until the left relation is processed completely. To work efficiently with the operator, the smaller relation should be used as the left relation and the operator should have enough memory to store the relation completely in memory.

The query for the join is printed in Listing 8 on page 37. The operator **itSpatialJoin** creates a stream of tuples with overlapping bounding boxes. The output of the operator is refined by the operator **intersects**. The **intersects** operator calculates whether both data elements really overlap. Because this is a CPU intensive operation, the **itSpatialJoin** operator ensures first that the bounding boxes do overlap.

7.2.2 The Parallel Version

The algorithm for the parallel version is an adapted version of the *SJMR* algorithm [43]. The spatial data is overlapped with a grid. Corresponding cells of both relations are stored at the same position in Cassandra. If an object is part of more cells, the object is stored multiple times. The concrete queries can be found in Listing 9 in the appendix.

After the data is spread, the QEP for the join can be created. The full QEP is shown in Listing 10. Compared to the sequential version, the join operation in the listing is more complex. Due to the fact that the same spatial object can occur in multiple cells, the join result may contain duplicates. These duplicates must be eliminated. This task is performed by the operators **filter** and **gridintersects**. The first operator ensures, that both objects are part of the same cell; this is simply an integer comparison. The operator **gridintersects** works as a filter too. It ensures, that only the overlapping in the common smallest cell is reported.

7.3 Handling a Stream of GPS Coordinate Updates

This example shows, how a data stream of GPS coordinates can be handled with Distributed Secondo. In a real world scenario, such data is generated by vehicles, which drive through the streets of a city and send GPS updates every few seconds to a central system.

BerlinMOD [12] is a benchmark for spatio-temporal database management systems. The benchmark generates trips of moving vehicles within Berlin. The data generated by BerlinMOD is used here, to simulate moving vehicles. Position updates are represented as lines. Every line consists of five fields, each separated by a comma (see Listing 4).

Listing 4 Lines with GPS coordinates

```
1 28-05-2007 10:00:14,10,1313,13.2967,52.4502
2 28-05-2007 10:00:15,22,3107,13.2178,52.5086
3 28-05-2007 10:00:15,112,16315,13.2063,52.5424
4 28-05-2007 10:00:15,6,751,13.322,52.4626
```

The first field contains the time when the GPS coordinate was recorded. The second field identifies the vehicle, the third field the trip. This field is ignored in this example. The fourth and the fifth field contain the coordinates. The generated coordinate stream consists of lines in the format described above, ordered by the time stamp of the lines.

To process the stream of data, one or more MNs run the operator **csvimport** which is reading CSV lines from a network socket and converts them into tuples. This can be done by the query printed in Listing 5. Distributed Secondo also includes a TCP load balancer. By using the load balancer, it is possible to distribute the stream of GPS coordinates on more than one MN. The operator **csvimport** generates a stream of tuples which consists of only basic data types like *int*, *real* and *string*. To work efficently with the spatial data, the imported data needs to be converted into special data types. This is the task of the operators in the middle of the query. The output of the operator **ipointstoupoint** is forwarded to the operator **cspread**. This operator stores all the tuples onto the storage nodes.

Listing 5 Import a stream of GPS data

```
1 query [ const rel(tuple([Time: string, Moid: string, Tripid: int, X: real,
2    Y: real])) value() ]
3    csvimport ['tcp://10025', 0, ""] extend [I: str2instant(.Time,
4    "D.M.Y h:m:s"), P: makepoint(.X, .Y)] projectextend[Moid; Position:
5    the_ivalue(.I, .P)] ipointstoupoint["Moid"] cspread['gpsdata', Moid];
```

8 Evaluation

The evaluation of Distributed Secondo is performed on a cluster of 6 nodes. Each of the nodes contains a Phenom(*tm*) II X6 1055T processor with 6 cores, 8 GB of memory and two 500 GB hard disks. All nodes are connected via an 1 Gbit/s switched Ethernet network.

On all nodes the Linux distribution Ubuntu 14.04 (amd64) is installed. Cassandra is used in version 2.2.5, running on an Oracle Java VM in version 1.7.0. The cpp-driver is used in version 2.3. Cassandra stores all the data on hard disk 1, the commitlog is written on hard disk 2. Unless stated otherwise, all experiments are executed in a keyspace with a replication factor of three and a consistency level set to `QUORUM`. Every SN uses 64 tokens in the logical ring. Therefore, 385 UOWs need to be processed. Each experiment was repeated five times and the average execution time is used in the results.

8.1 Used Data Sets

To measure the performance of the system, we used three data sets:

- OSM-Data: In Section 7.2, a data set with the spatial data of Germany was introduced. This data set is used again for the experimental evaluation. The relation *Roads* contains 9 655 101 tuples, the relation *Buildings* contains 24 058 912 tuples.
- TPCH-Data: To perform a join on relational data, we use the data generator of the TPC-H benchmark. The TPC-H benchmark offers a data generator to produce the corresponding data set. For the experiments, the data set was generated with a scale factor of 10. The relation *lineitem* contains 59 986 052 tuples, the relation *orders* contains 15 000 000 tuples.
- CSV-Data: We developed a data generator that generates synthetic CSV structured data. This tool is used to evaluate the performance of the MNs and the SNs. The tool can generate more data than the nodes can theoretically process. The data is generated in memory, this is much faster than reading data from a hard disk. For the experiments, the data generator creates 500 000 lines of CSV data with five fields and each field containing 1 000 characters[5]. Therefore, the nodes have to process 2386.47 MB of data.

8.2 Processing Data on the MNs

This experiment examines, how additional MNs can be used to process a stream of data. On node1, the data generator for the CSV-Data data set and a load balancer for TCP connections are executed. The load balancer distributes the network stream onto a varying amount of MNs. The MNs are running the operator **csvimport** to parse the input data. The operator generates a stream of tuples which is consumed by the **count** operator. The **count** operator can be used as a sink for tuples. The operator reads tuples and only increments an integer. Listing 6 contains the full query of this experiment.

Listing 6 Processing data on the MNs

```
1 query [const rel(tuple([C1: text, C2: text, C3: text, C4: text, C5: text))
2    value() ] csvimport['tcp://10025', 0, ""] count;
```

The data needs to be transferred though the network. Therefore, the theoretical minimal execution time can be calculated as follows: The data generator and the load balancer are executed on hardware node1. Consequently, the data for node1 does not need to be transferred through the network. In contrast, the data for the remaining nodes needs to be transferred. Generally, by using n nodes $\frac{n-1}{n}$ of the data needs to be transferred. Figure 22 shows, that the data stream can be processed faster with additional nodes.

[5]Each line contains 5000 characters + 4 field separators (e.g. »,«) + 1 new line character (e.g. »\n«) = 5005 bytes per line. By creating 500 000 lines with 5005 bytes each, 2386.47 MB data in total is generated.

	Management nodes					
	1	2	3	4	5	6
Time (Sec)	45.92	24.86	17.40	17.18	17.30	17.31
Speedup	-	1.85	2.64	2.67	2.65	2.65

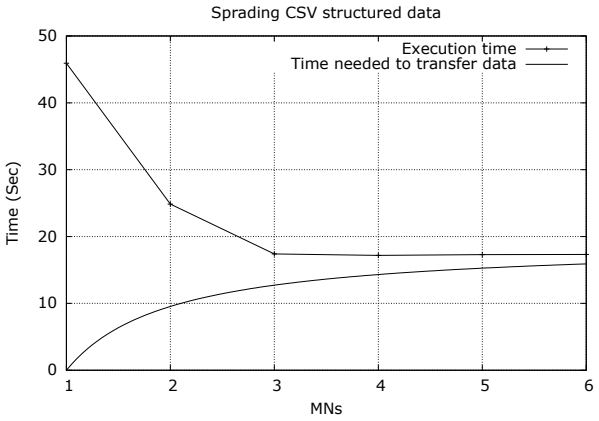


Figure 22: Time to spread 500 000 lines onto the MNs.

The measured execution time and the theoretical execution time for the data transfer approach each other. By using six nodes, the measured execution time is 17.31 seconds; transferring the data requires 16.68 seconds. This means, that we have only a difference of 0.63 seconds from the calculated theoretical value[6]. The measured execution times are useful to evaluate the results of the next experiment.

8.3 Writing Data onto the SNs

This experiment examines, how the SNs of Distributed Secondo can be upscaled. To measure this, the last experiment is repeated with a slightly modified query (see Listing 7).

Listing 7 Writing data onto the SNs

```
1 query [const rel(tuple([C1: text, C2: text, C3: text, C4: text, C5: text))
2    value() ] csvimport['tcp://10025', 0, ""]
3    cspread['data', C1, 'system', 'ONE'];
```

Again, the CSV-Data data set is used and spread with a load balancer onto 6 MNs. The MNs parse the data with the **csvimport** operator. This time, the output of the **csvimport** is read by the **cspread** operator which writes the received data onto the storage nodes. Figure 23 shows the result of the experiment. It can be seen, that the execution time decreases when the number of used SNs increases. Using a higher replication level leads to higher execution times.

In contrast to the last experiment, it is not possible to calculate a lower bound for the execution time. At first glance, it could be argued, that the lower bound must be three times of the lower bound of the last experiment. After the data is spread to the MNs, the data has to be transferred to the coordinating node. From the coordinating node, the data has to be transferred to the SN. Cassandra uses compression when data is transferred. This leads to a lower amount of data that has to be transferred and to lower execution times. In the last experiment, we calculated 16.86 seconds as a lower bound when six nodes were used. In this experiment, we measured an execution time of 26.64 seconds which is less than the doubled lower bound (33.36 seconds).

[6]The data generator creates 2386.57 MB of data, 1988.81 MB needs to be transferred. With an 1 GBit/s network link, the transfer takes 16.68 seconds.

	Storage nodes					
	1	2	3	4	5	6
Replication factor = 1						
Time (Sec)	44.7	30.9	29.68	28.89	26.48	26.64
Speedup	-	1.45	1.51	1.55	1.69	1.68
Replication factor = 2						
Time (Sec)	-	36.9	34.94	33.83	29.71	29.71
Speedup	-	-	1.06	1.09	1.24	1.24
Replication factor = 3						
Time (Sec)	-	-	39.35	37.19	33.32	33.16
Speedup	-	-	-	1.06	1.18	1.19

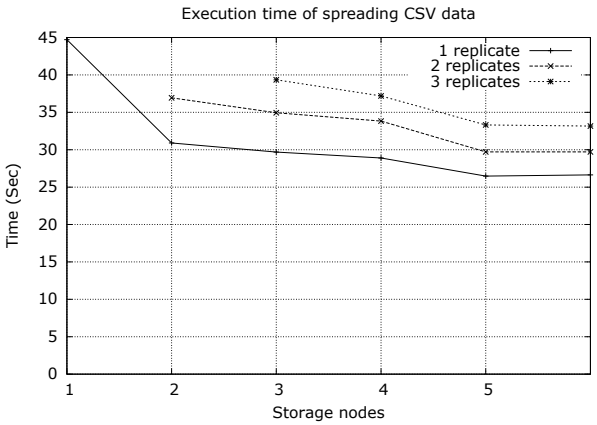


Figure 23: Time to spread 500 000 lines onto the SNs. The execution time is measured in seconds.

8.4 A Join on Relational Data

In this experiment a join on relational data is performed. The 12^{th} query of the TPC-H benchmark [7] is executed on the TPC-H data set. The sequential version of the query can be found in Listing 11; the parallel one in Listing 12 of the appendix.

The experiment is split up in two parts: (*i*) In the first part of the experiment, one QPN is used and the amount of Secondo-threads is increased from 1 to 3 threads. (*ii*) In the second part of the experiment, the number of QPNs is increased from 1 to 6, each node executes 3 Secondo-threads, i.e., by using 2 nodes, 6 Secondo-threads are active. The execution time of the experiment and the speedup is shown in Figure 24. It can be seen, that the execution time decreases when the number of Secondo-threads increases.

The sequential version on one node takes $\approx$ 1568 seconds, which is three times slower than the version with 6 nodes. Spreading the relations orders and lineitem onto the SNs takes 1107 and 4930 seconds. Importing the result back takes only 1 second. Each UOW produces only one result tuple, which contains the number of joined tuples.

8.5 A Join on Spatial Data

In this experiment, another join is performed. In contrast to the last experiment, spatial data is used. In Section 7.2, the details of the spatial join has been already discussed. The join is executed on the OSM-Data data set and the query processing time is observed.

8.5.1 Varying the Amount of QPNs

The sequential version uses 6 GB of memory for the MMR-tree of the **itSpatialJoin** operator. The operator needs to build two MMR-trees. Therefore, the second relation needs to be stored on disk and it has to be analyzed two times (one time for each MMR-tree). The parallel version uses 1.5

	Threads							
	1	2	3	6	9	12	15	18
Time	7233	3967	2547	1418	975	680	627	574
Speedup	-	1.82	2.84	5.10	7.42	10.64	11.54	12.60

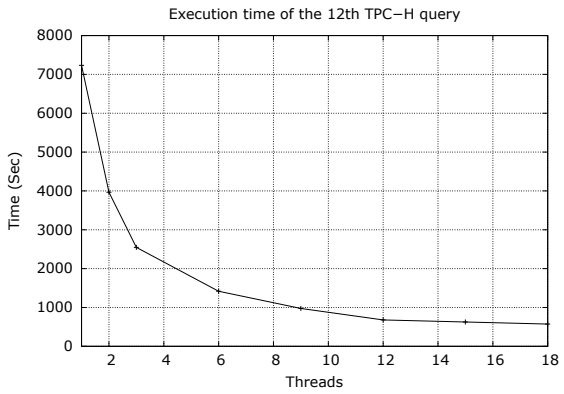


Figure 24: The 12th TPC-H query, executed on a varying number of QPNs and SECONDO-threads. Each hardware node performs a maximum of three SECONDO-threads. The execution time is measured in seconds.

GB of memory[7]. Figure 25 depicts the execution time of the join. By using multiple threads, the execution time can be reduced. The same is true for the usage of multiple hardware nodes. The sequential version has an execution time of 820 minutes, which is slower than the parallel version with one thread. The parallel version uses smaller parts (the UOWs) as input. Therefore, the operator **itSpatialJoin** can work more efficiently. The whole MMR-tree of the left relation can be created in memory and the right relation needs to be read only one time.

To execute the parallel version, the input relations need to be partitioned by a grid and stored onto the SNs. This task takes 26 minutes for the relation roads and 49 minutes for the relation buildings. In addition, after the execution of the join the result is stored onto the SNs. The 423 269 result tuples need to be imported back onto one MN to work with (this takes 2 minutes). It takes 77 minutes for the additional tasks in total.

The relational join has a slightly better speedup behavior than the join on spatial data. The reason for this is the handling of the result. The spatial version writes all result tuples back onto the SNs. The 12th query of the TPC-H benchmark writes only the amount of result tuples back onto the SNs.

8.5.2 Varying the Replication Level

In this experiment the spatial join is executed repeatedly. Between the executions, the replication level is increased from 1 replicate to 6 replicates. The experiment uses 6 QPNs, each running 3 SECONDO-threads. Two different consistency levels `ONE` and `ALL` are used. By using the first consistency level, only one SN has to acknowledge write operations and one SN has to answer on read operations. By using the second consistency level, the amount of answering SNs is determined by the replication factor.

It can be seen in Figure 26, that the join becomes slower when the replication level increases. The reason for this behavior is the additional amount of data that needs to be written. A higher consistency level leads to higher execution times too. The reason for this behavior is, that read and write operators need to wait for more systems until they are complete. By using a replication factor

[7]The parallel version executes multiple SECONDO-threads on one hardware node. This is the reason, why the parallel version can't use 6 GB memory for each thread. However, UOW are small and with 1.5 GB memory only one MMR-tree needs to be created. As a consequence, the second relation needs to be analyzed only one time.

	Threads							
	1	2	3	6	9	12	15	18
Time	15329	8441	5956	3094	2113	1697	1424	1314
Speedup	-	1.82	2.57	4.95	7.25	9.03	10.76	11.67

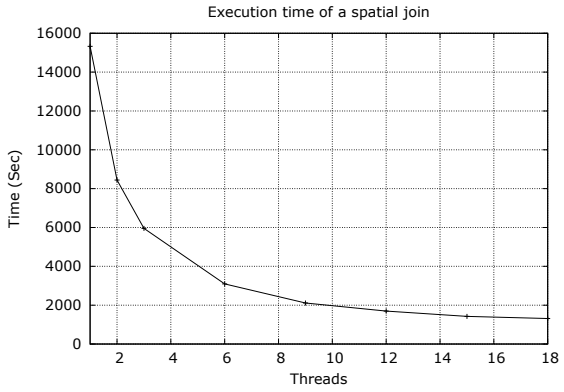


Figure 25: A spatial join, executed on a varying number of QPNs and SECONDO-threads.. Each hardware node performs a maximum of three SECONDO-threads. The execution time is measured in seconds.

of 6 and the consistency level `ONE` on a write operation, only one system needs to write the data to disk. The other 5 write operations can be handled asynchronously.

8.6 Unit of Work Reassignment

In a large distributed environment, it cannot be guaranteed, that the whole hardware is equal and provides the same resources for computation. Hard disks can become slow when read errors occur and sectors need to be read multiple times. Network links can be utilized and the data transfer to a certain node needs a longer time. To compensate the different computation speeds, the algorithm for the distribution of the UOWs distributes in phase 3 the units randomly across the idle QPNs. To demonstrate this feature, we changed the speed of the network connection of node 3 to 100 Mbit/s. The other nodes are still connected with 1 Gbit/s.

If each QPN only processes the directly assigned UOWs, one slow QPN could increase the whole processing time significantly. In our particular case, the QPNs 1,2,4,5,6 become idle after some time. At this point, only node 3 is computing UOWs. The query would not be complete until the last QPN finishes its work.

With UOW reassignment, a slow system could not increase the processing time much. Idle QPNs choose randomly unprocessed UOWs and process them. In our experiments, the 12^{th} TPC-H query runs 8 % faster (681 seconds without and 628 seconds with UOW reassignment). The spatial join runs 13 % faster (1928 seconds without and 1683 seconds with UOW reassignment). Figure 27 shows the amount of processed UOWs of a spatial join for each QPN. QPN 3 computes only $\approx 60\%$ of the UOWs of the other QPNs. This means, that the different processing speeds are balanced out; the slow QPN processes less work than the other QPNs.

9 Conclusion and Future Work

In this paper, we have presented a novel architecture of a scalable and highly available database management system. DISTRIBUTED SECONDO is capable of processing large amounts of data in parallel in a distributed environment.

In contrast to existing NoSQL-Databases, this system can process advanced data types and offers a wide range of operations to analyze data. The whole system is highly available, no central coordinator does exist. DISTRIBUTED SECONDO uses an algorithm based on the structure of the

Consistency Level	Replication factor 1	2	3	4	5	6
`ONE`	1190	1211	1253	1296	1317	1343
`ALL`	1210	1234	1267	1308	1341	1390

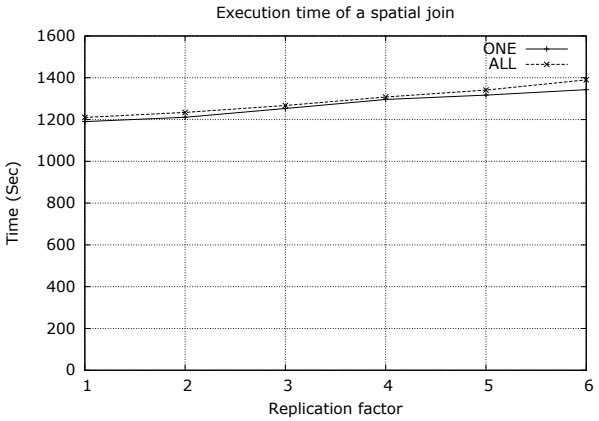


Figure 26: Execution time of a spatial join with different consistency levels and replication factors. All execution times are measured in seconds.

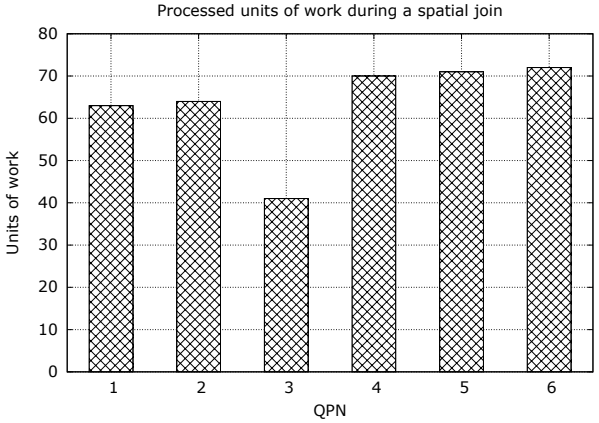


Figure 27: Average amount of processed UOWs per QPN during a spatial join without UOW reassignment. Node 3 is connected with a 100 Mbit/s network link, the remaining nodes with a 1 Gbit/s network link.

logical ring of Cassandra to create and distribute UOWs. For the future versions of Distributed Secondo, we are planning to replace Cassandra with a new storage manager. The storage manager should exploit the locality of the stored data better. When data is read from Cassandra, all copies of the data are read and transferred through the network. QPNs and SNs are sharing the same physical hardware, so most of the data can be read from the local hard disk without creating network traffic.

In addition, the fault tolerance and the resulting query processing time of the whole system needs to be analyzed more precisely. We plan to introduce functions in Secondo to crash the system with a certain probability. The evaluation of the software was performed in a small cluster with only six different hardware nodes. It is planned to evaluate the system in clusters with hundreds of hardware nodes. Distributed Secondo can handle data streams. A further area for research is the continuous update of data structures like indexes and the propagation of new data to the GUI of Secondo.

References

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, August 2009.

[2] Website of Apache Drill, 2015. `http://drill.apache.org` - [Online; accessed 20-Jul-2015].

[3] Apache license, version 2.0, 2004. `http://www.apache.org/licenses/` - [Online; accessed 30-Jul-2015].

[4] S. Ceri and G. Pelagatti. *Distributed Databases Principles and Systems*. McGraw-Hill, Inc., New York, NY, USA, 1984.

[5] 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. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[6] J.C. Corbett, J Dean, M. Epstein, A. Fikes, C. Frost, J.J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[7] Transaction Processing Performance Council. TPC BENCHMARK H (Decision Support) Standard Specification. `http://www.tpc.org/tpch/`. [Online; accessed 15-May-2015].

[8] Website of cpp-driver for Cassandra. `https://github.com/datastax/cpp-driver`, 2015. [Online; accessed 15-Sep-2015].

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[11] F. Dinun and T.S.E. Ng. Understanding the effects and implications of compute node related failures in hadoop. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 187–198, New York, NY. USA, 2012. ACM.

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

[13] J.F. Gantz and D. Reinsel. The digital universe in 2020: Big data, bigger digital shadow's, and biggest growth in the far east. *IDC*, 2012.

[14] L. George. *HBase: The Definitive Guide*. O'Reilly Media, Inc., 2011.

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

[16] Google spotlights data center inner workings - An interview with Google fellow Jeff Dean, 2008. `http://www.cnet.com/news/google-spotlights-data-center-inner-workings/` - [Online; accessed 25-Jul-2015].

[17] R.H. Güting. *Operator Based Query Progress Estimation.* Informatik-Berichte. FernUniversität in Hagen. Faculty of Mathematics and Computer Science, 2008.

[18] 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.

[19] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu. Ibm streams processing language: Analyzing big data in motion. *IBM J. Res. Dev.*, 57(3-4):1:7–1:7, May 2013.

[20] 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.

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

[22] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu. Deduce: at the intersection of mapreduce and stream processing. In Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan, editors, *EDBT*, volume 426 of *ACM International Conference Proceeding Series*, pages 657–662. ACM, 2010.

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

[24] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[25] P. Leach, M. Mealling, and R. Salz. RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace, 2005.

[26] J. Lu and R.H. Güting. Parallel secondo: Boosting database engines with hadoop. *2013 International Conference on Parallel and Distributed Systems*, 0:738–743, 2012.

[27] J.K. Nidzwetzki. *Entwicklung eines skalierbaren und verteilten Datenbanksystems.* Springer, 2016.

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

[29] Website of the Open Street Map Project, 2015. `http://www.openstreetmap.org` - [Online; accessed 09-Jul-2015].

[30] M.T Özsu and P. Valduriez, editors. *Principles of Distributed Database Systems.* Springer, New York, NY. USA, 3 edition, 2011.

[31] W. Palma, R. Akbarinia, E. Pacitti, and P. Valduriez. Distributed processing of continuous join queries using dht networks. In *Proceedings of the 2009 EDBT/ICDT Workshops*, EDBT/ICDT '09, pages 34–41, New York, NY, USA, 2009. ACM.

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

[33] R. Rea and K. Mamidipaka. Ibm infosphere streams - enabling complex analytics with ultra-low latencies on data in motion. *IBM Corporation*, 2009.

[34] J.B. Rothnie, P.A. Bernstein, S. Fox, N. Goodman, M. Hammer, T.A. Landers, C. Reeve, D.W. Shipman, and E. Wong. Introduction to a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 5(1):1–17, March 1980.

[35] J.B. Rothnie and N. Goodman. A survey of research and development in distributed database management. In *Proceedings of the Third International Conference on Very Large Data Bases - Volume 3*, VLDB '77, pages 48–62. VLDB Endowment, 1977.

[36] J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner, K. Littleield, and P. Tong. F1 - the fault-tolerant distributed rdbms supporting googles ad business. In *SIGMOD*, 2012. Talk given at SIGMOD 2012.

[37] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[38] A.S. Tanenbaum and M.v. Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[39] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

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

[41] Website of distributed secondo, 2015. `http://dna.fernuni-hagen.de/secondo/DSecondo/DSECONDO-Website/index.html` - [Online; accessed 15-Nov-2015].

[42] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

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

A Queries of the Experiments

Listing 8 Spatial Join - sequential version

```
let join_result = query Roads feed {r1}
   Buildings feed {r2}
   itSpatialJoin[GeoData_r1, GeoData_r2] {memory 6144}
   filter[(.GeoData_r1 intersects .GeoData_r2)] count:
```

Listing 9 Exporting the spatial data to the SNs

```
query Roads feed extend[Box: bbox(.GeoData)] projectextendstream
   [Name, Osm_id, Box , GeoData; Cell: cellnumber(.Box, CellGrid)]
   cspread['RoadsCell', Cell];

query Buildings feed extend[Box: bbox(.GeoData)] projectextendstream
   [Name, Osm_id, Box, GeoData; Cell: cellnumber(.Box, CellGrid)]
   cspread['BuildingsCell', Cell];
```

Listing 10 Spatial Join - parallel version

```
query cqueryexecute(1,'create database db1;');

query cqueryexecute(2,'open database db1;');

query cqueryexecute(3,'
   let CellGrid = createCellGrid2D(5.8, 47.1, 0.01, 0.01, 930);');

query cqueryexecute(4, 'query [readparallel RoadsCell] {r1}
   [readparallel BuildingsCell] {r2}
   itSpatialJoin[Box_r1, Box_r2] {memory 1536} filter[(.Cell_r1 = .Cell_r2) and
   gridintersects(CellGrid, .Box_r1, .Box_r2, .Cell_r1) and
   (.GeoData_r1 intersects .GeoData_r2)] [write pbsmresult Cell_r1];');
```

Listing 11 12th TPC-H query - sequential version

```
query LINEITEM feed
  filter[ ((.LSHIPMODE = "MAIL") or (.LSHIPMODE = "SHIP"))
    and (.LCOMMITDATE < .LRECEIPTDATE)
    and (.LSHIPDATE < .LCOMMITDATE) ]
ORDERS feed itHashJoin[LORDERKEY, OORDERKEY]
sortby[LSHIPMODE]
groupby[LSHIPMODE ; HLC: group feed
    filter[(.OORDERPRIORITY = "1-URGENT")
      or (.OORDERPRIORITY = "2-HIGH")] count,
  LLC: group feed
    filter[(.OORDERPRIORITY # "1-URGENT")
      and (.OORDERPRIORITY # "2-HIGH")] count]
consume;
```

Listing 12 12th TPC-H query - parallel version

```
query LINEITEM feed cspread['LINEITEM', LORDERKEY];

query ORDERS feed cspread['ORDERS', OORDERKEY];

query cqueryreset();

query cdelete('TPC12RESULT');

query cqueryexecute(1,'create database db1;');

query cqueryexecute(2,'open database db1;');

query cqueryexecute(3,'query [readparallel
    LINEITEM] filter[ ((.LSHIPMODE = "MAIL")
      or (.LSHIPMODE = "SHIP"))
      and (.LCOMMITDATE < .LRECEIPTDATE)
      and (.LSHIPDATE < .LCOMMITDATE)]
    [readparallel ORDERS]
    itHashJoin[LORDERKEY, OORDERKEY]
    sortby[LSHIPMODE]
    groupby[LSHIPMODE ; HLC: group feed
    filter[(.OORDERPRIORITY = "1-URGENT")
      or (.OORDERPRIORITY = "2-HIGH")] count,
    LLC: group feed
     filter[(.OORDERPRIORITY # "1-URGENT")
      and (.OORDERPRIORITY # "2-HIGH")] count]
    [write TPC12RESULT LSHIPMODE]');

query cquerywait(3);

query ccollectquery('TPC12RESULT', 3)
    sortby[LSHIPMODE] groupby[LSHIPMODE; HLC: group
    feed sum[HLC], LLC: group feed sum[LLC]] consume;
```

B Installation

In this section, the installation of Distributed Secondo is described. First of all, a management node will be installed. Starting out from this system, the QPNs and the SNs are installed. Generally, Distributed Secondo runs on every platform which is supported by Secondo. But using Debian 7/8 or Ubuntu 14.04 is recommended because these distributions are used during development and the system is tested on there.

Prerequisites: All nodes need to be installed with an proper OS. The network of the nodes needs to be configured and the nodes should be reachable via SSH. The software *rsync*, *ssh*, *screen*, *java*, *git*, *gcc* and *make* should be installed on these systems. On dpkg based Linux distributions, e.g. Debian or Ubuntu, this can easily be done by running the following command:

```
apt-get install cmake libtool automake git openssh-client screen
   build-essential libssl-dev rsync openjdk-7-jdk
```
(25)

B.1 Installing a Management Node

- Create the necessary directory structure:

```
export DSECONDO_DIR=~/dsecondo/
```
(26)

```
mkdir -p $DSECONDO_DIR
```
(27)

```
mkdir -p $DSECONDO_DIR/secondo
```
(28)

- Download and install Secondo into the directory `$DSECONDO_DIR/secondo`. An installation guide and the download of Secondo can be found under the URL http://dna.fernuni-hagen.de/secondo/content_downloads.html

- Add the following lines to your ~`/.secondorc` and adjust the names of the QPNs and SNs. The placeholder `%%YOUR_DSECONDO_DIR%%` needs to be replaced with the full path of the directory `$DSECONDO_DIR`. The placeholder `%%YOUR_QPN_INSTALL_DIR%%` needs to be replaced with the name of the directory were Secondo should be installed on the QPNs (e.g. `/opt/dsecondo`).

Listing 13 The additional lines for the .secondorc

```
###
# DSECONDO
###

export DSECONDO_DIR=%%YOUR_DSECONDO_DIR%%
export DSECONDO_QPN_DIR=%%YOUR_QPN_INSTALL_DIR%%

# Locate the libuv and cpp-driver installation dir
if [ -d $DSECONDO_DIR/driver/libuv ]; then
export LD_LIBRARY_PATH=$DSECONDO_DIR/driver/libuv/.libs:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$DSECONDO_DIR/driver/cpp-driver:$LD_LIBRARY_PATH
else
export LD_LIBRARY_PATH=$DSECONDO_QPN_DIR/driver/libuv/.libs:
    $LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$DSECONDO_QPN_DIR/driver/cpp-driver:$LD_LIBRARY_PATH
fi

# DSECONDO - Hostnames of the QPNs
export DSECONDO_QPN="node1 node2 node3 node4 node5 node6"

# DSECONDO - Hostnames of the SNs
export DSECONDO_SN="node1 node2 node3 node4 node5 node6"
```

- Reread the file ~/.secondorc by executing:

```
source ~/.secondorc                                                    (29)
```

- Download, build and install the cpp-driver and its dependencies:

```
cd $DSECONDO_DIR/Algebras/Cassandra/tools                              (30)
./manage_dsecondo.sh install_driver                                    (31)
```

- Add the following lines to the file `makefile.algebras` to enable the CassandraAlgebra:

Listing 14 Additional lines for the makefile.algebras

```
1 ALGEBRA_DIRS += Cassandra
2 ALGEBRAS += CassandraAlgebra
3 ALGEBRA_DEPS += uv cassandra
4
5 ALGEBRA_INCLUDE_DIRS += $(DSECONDO_DIR)/driver/cpp-driver/include
6 ALGEBRA_INCLUDE_DIRS += $(DSECONDO_DIR)/driver/libuv/include
7
8 ALGEBRA_DEP_DIRS += $(DSECONDO_DIR)/driver/libuv/.libs
9 ALGEBRA_DEP_DIRS += $(DSECONDO_DIR)/driver/cpp-driver
```

- Build Secondo with the CassandraAlgebra enabled:

```
cd $DSECONDO_DIR/secondo                                               (32)
make                                                                   (33)
```

- Add the following lines to the configuration file of Secondo (`SecondoConfig.ini`). The placeholder `%%SN_IP%%` has to be replaced with the IP of one of the storage nodes.

Listing 15 Additional lines for the SecondoConfig.ini

```
1 [CassandraAlgebra]
2 CassandraHost=%%SN_IP%%
3 CassandraKeyspace=keyspace_r3
4 CassandraConsistency=QUORUM
5 CassandraDefaultNodename=node1
```

- Now, the installation of the Management Node is finished.

B.2 Installing the Query Processing Nodes

The Secondo installation of the Management Node will be used to install the Query Processing Nodes. The whole installation is carried out by an installer. The installer will copy the Secondo installation (and the dependencies like the cpp-driver) into the directory `$DSECONDO_QPN_DIR` onto the QPNs `$DSECONDO_QPN`.

- Install the QPNs:

```
cd $DSECONDO_DIR/secondo/Algebras/
Cassandra/tools                                                        (34)
./manage_dsecondo.sh install                                           (35)
```

- If you have modified the Secondo-Installation on the MN, for example by switching to a more recent version or by activating additional algebras, you can use the script `manage_dsecondo.sh` to upgrade all QPNs. The software `rsync` is used to update only the changed files onto the QPNs. The upgrade of the QPNs is done by the following command:

```
./manage_dsecondo.sh upgrade                                           (36)
```

- All the QPNs can be started by running the following command

  ```
  ./manage_dsecondo.sh start
  ```
  (37)

  By changing the parameter `start` to `stop` all the nodes are stopped.

B.3 Installing the Storage Nodes

The installation of the storage nodes is performed by the script `manage_cassandra.sh`. This script downloads Cassandra, installs it and applies a basic configuration. In addition, this script can be used to start and stop Cassandra.

- Install Cassandra on the SN `$DSECONDO_SN`:

  ```
  ./manage_casandra.sh install
  ```
  (38)

- Start the SNs. (Similar to `manage_dsecondo.sh` changing the parameter `start` to `stop` will stop all SNs.)

  ```
  ./manage_casandra.sh start
  ```
  (39)

- Create the default keyspaces (keyspace_r1 - keyspace_r6) and system tables. The keyspace keyspace_r1 has a replication factor of 1, keyspace_r6 has a replication factor of 6. This step is only needed at the first start of Cassandra.

  ```
  ./manage_casandra.sh init
  ```
  (40)

After completing these steps, the installation of Distributed Secondo is finished.

Verzeichnis der zuletzt erschienenen Informatik-Berichte

[360] Xu, J., Güting, R.H.:
A Generic Data Model for Moving Objects, 8/2011

[361] Beierle, C., Kern-Isberner, G.:
Evolving Knowledge in Theory and Application: 3rd Workshop on Dynamics of Knowledge and Belief, DKB 2011

[362] Xu, J., Güting, R.H.:
GMOBench: A Benchmark for Generic Moving Objects, 3/2012

[363] Finthammer, M.:
A Generalized Iterative Scaling Algorithm for Maximum Entropy Reasoning in Relational Probabilistic Conditional Logic Under Aggregation Semantics, 4/2012

[364] Güting, R.H., Behr, T., Düntgen, C.:
Book Chapter: Trajectory Databases; 5/2012

[365] Paul, A., Rettinger, R., Weihrauch, K.:
CCA 2012 Ninth International Conference on Computability and Complexity in Analysis (extended abstracts), 6/2012

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

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

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

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

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