Report Vertiefung, Spring 2013
Constant Interval Extraction using Hadoop

Thomas Brenner, 08-928-434

1 Introduction and Task

Temporal databases are databases expanded with a time dimension in order to qualify the entry in a relation with a specific timestamp. Each temporal relation has in addition to its entries two additional columns, a starting point and an endpoint during which the entry is valid. An example for a temporal database can be the following Figure 1, which shows a relation, where a salary is valid only between the starting point and the endpoint:

Relation 1		
Salary	Starting point	Endpoint
10’000	91	98
20’000	93	99

Figure 1: Example Relation 1

To evaluate nontemporal functions, for the example in Figure 1 the average salary which is paid, over temporal data, it is necessary to determine constant intervals, during which the data is not altered. Figure 2 shows the constant intervals for the relation in Figure 1, over which the data is not altered:

Constant Intervals over relation 1				
Total Salary	Constant Intervals			
10’000	91	93		
30’000		93	98	
20’000			98	99

Figure 2: Constant Intervals over Relation 1

By increasing the amount of data in the database, the effort to compute these constant intervals can increase rapidly and a brute force approach can be very expensive by performing these computations on a single computer in a PostgreSQL-database. It can be a possibility, to extract these timestamps out of the PostgreSQL-database, build the constant intervals with a more fitted method than PostgreSQL and, after the successful generation of the necessary data, to reintegrate the data into the database.

Apache Hadoop is a framework of software, which supports distributed applications on large clusters of commodity hardware. One of the greatest strength of Hadoop is the fact, that it was build to handle a large amount of data without the necessity of using large-capacity computers.
The main goal of this Vertiefung is, to make performance tests with Hadoop and PostgreSQL, in order to evaluate, in which cases Hadoop performs the computation of these constant intervals faster than PostgreSQL. Other goals are to learn the functionality and the possibilities of Hadoop and to make a connection between the Hadoop framework and PostgreSQL-databases.

2 First steps with Hadoop

2.1 What is Hadoop

Apache Hadoop is an open-source framework to perform distributed tasks, with a high data-intensity. It's based on a computational strategy called MapReduce. This means, that the task is divided in small fragments of work, which can be executed on every node in the cluster. As a second strong feature Hadoop provides the user with a distributed file system to handle the huge amount of data, which can be computed with Hadoop. The Hadoop file system as well as the MapReduce is designed to handle the failure of single nodes very generously, because in reality, thousands of independent computers can be summarized in a cluster and the failure handling is crucial for a framework like Hadoop.

Main point of the Hadoop framework is the MapReduce algorithm. This algorithm takes the input data and generates out of it a set of data, a key and a value. An example could be objects in two-dimensional environment. The name of the object is the key, which is the identifier for the object, and the value could be its position in a two-dimensional plane, defined by an X- and a Y-coordinate. After the data is modified by the mapper, which means, categorized by a key, the data is shuffled between the nodes, in order to guarantee failure security. After the shuffling process, the values, now subscripted by the key, can be reassembled by the reducers.

Hadoop was derived from some papers, which Google published in 2004. Until now some of the greatest companies in the world, like Deutsche Bank, Yahoo and Facebook, are using huge Hadoop clusters to perform their necessary computations.

2.2 Installation

To use Hadoop during this Vertiefung, Hadoop 1.0.4 was installed on a Ubuntu-Desktop 12.10 distribution, which was running as a virtual machine. The latest Hadoop distribution can be found on the Apache Website (http://hadoop.apache.org/releases.html). After the installation, some configuration files have to been altered and adjusted to the specific system. For example in the conf/*-site.xml files, the configuration of Hadoop can be adjusted to the wishes of the user. Figure 3 shows some configuration detail, which are useful to add to the hadoop-site.xml file in order to prepare the Hadoop-environment.

```xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
  <name>hadoop.tmp.dir</name>
  <value>/tmp/hadoop-${user.name}</value>
</property>
<property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:54310</value>
</property>
<property>
  <name>mapred.job.tracker</name>
  <value>hdfs://localhost:54311</value>
</property>
<property>
  <name>dfs.replication</name>
  <value>8</value>
</property>
<property>
  <name>mapred.child.java.opts</name>
  <value>-Xmx512m</value>
</property>
</configuration>
```

Figure 3: Configuration of hadoop-site.xml

The last property of the hadoop-site.xml (Figure 3) defines the slot-size of the hadoop-node. This property is very important to define the right size of reducers in a later stage.

2.3 Writing a first MapReduce application

In order to understand the MapReduce algorithm better, a Yahoo Tutorial (http://developer.yahoo.com/hadoop/tutorial/index.html) was followed to write the first MapReduce applications. A classical example to begin with MapReduce and Hadoop is the application WordCount. The goal of this application is to count the number of Words in a set of input files. As in the previous chapter explained, Hadoop uses the MapReduce algorithm to perform the computation, in order to distribute different tasks over the cluster. The structure of the program is shown in Figure 4.

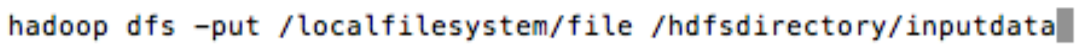
```
mapper (filename, file-contents):
  for each word in file-contents:
    emit (word, 1)

reducer (word, values):
  sum = 0
  for each value in values:
    sum = sum + value
  emit (word, sum)
```

Figure 4: MapReduce algorithm of WordCount

The mapper takes all the input files and emits for each word a set of value and key, where the key is the word itself (in order to identify it through all words) and the value is 1, because one instance of the word was found. After the map-procedure, the data is shuffled and rearranged by the reducer. As it is shown in Figure 4, for each key, the sum is increased and in the end emitted.

2.4 Running MapReduce on Hadoop

After the MapReduce program is compiled, the Hadoop-environment is set-up properly and started, the necessary input data has to be loaded into the Hadoop File system (HDFS), so that it can be used by Hadoop. This task can be achieved by the following command shown in Figure 5.



```
hadoop dfs -put /localfilesystem/file /hdfsdirectory/inputdata
```

Figure 5: Command to put data into HDFS

After the data is put into the HDFS, one can perform the Hadoop-Job. It's important that Hadoop was build to perform on a huge amount of data, so the overhead for computations over a small size of data can produce a performance-time a lot bigger than the expected amount of time.

2.5 Number of Mappers and Reducers

The number of mappers is automatically set by the system and depends on factors like the amount of input data. While the exact number of mappers can not be manipulated manually, the maximum number can be set, in order to optimize the execution of a task.
On the other hand, the performance of the task can be significantly improved by choosing the optimal number of reducers. There exists no exact rule, how many reducers have to be chosen. Nevertheless, there are some specifications in the system, which can be used to estimate a number of reducer. As a rule of thumb 1.75 * (nodes * tasks per node) can be a good choice for most problems, because with this amount of reducers, the faster nodes will finish their first round of reduces and launch a second round of reduces doing a much better job of load balancing. But not only the number of slots is decisive. Also the composition of the task can influence the performance of Hadoop.

3 Connect SQL and Hadoop

Temporal data is stored and computed in a PostgreSQL-database. A very important step to compute the constant intervals in Hadoop is the connection between these two frameworks. This means, how can the data be extracted from PostgreSQL, be computed in Hadoop and afterwards reimported into the database**.** In this Vertiefung two different ways to accomplish this goal were elaborated.

The first way, described in section 3.1. of this report, is to use a framework called Hive, which is also provided by Apache, and works on the top of Hadoop, giving the user the possibility to create a SQL-like database, in which queries can be run similar to PostgreSQL, but being processed in the background by Hadoop. The idea behind Hive is, to transform the whole database into a Hive-database and perform all the queries in this Hive-database.

The second way is to extract the data out of the PostgreSQL-database, sort the data in Java using Hadoop, transform it in the desired format and give it back to PostgreSQL. The second way is described in section 3.2. of this report.

3.1 Using Hive to compute Constant Intervals

3.1.1 First Steps with Hive

As above mentioned, a way to process a PostgreSQL-query in Hadoop is the Hive-framework. The necessary files can be downloaded on the Apache Website (http://hive.apache.org/releases.html). As a pre-condition to run Hive, PostgreSQL must be installed on the system, because Hive needs a SQL-database to work on it.

The desired release of Hive has to be unzipped and then the folder must be placed in the Hadoop-Home directory. As a next step, the Hive-matrix must be installed properly into SQL. Some of these schemas can be found in the installed Hive folder or on the Apache Website: (http://svn.apache.org/viewvc/hive/trunk/metastore/scripts/upgrade/).

After this schema is loaded into the specific SQL-database, the configuration-file, similar to the one for Hadoop, must be specified. The configuration of Hive can be found following the path: ./conf/hive-site.xml. In this configuration file, the authorization to the SQL-database must be specified, to allow Hive access to the matrix already installed in the database. The necessary XML-properties are shown in Figure 6.

After the installation of Hive, the Hive-console can be accessed via the "./hive/bin/hive"-command in the terminal. The functionality of the Hive-console is very similar to a corresponding console in MySQL or PostgreSQL. Hive supports a similar language called HiveQL.

Hive works a lot like PostgreSQL. Because it should handle a large amount of data, even the time to load the data into the HDFS can be enormous. A big advantage of Hive is that the data has not to be loaded explicitly into the HDFS or a database, but can be used as an external text file.

```
<property>
  <name>javax.jdo.option.ConnectionURL</name>
  <value>jdbc:mysql://MYHOST/metastore</value>
</property>

<property>
  <name>javax.jdo.option.ConnectionDriverName</name>
  <value>com.mysql.jdbc.Driver</value>
</property>

<property>
  <name>javax.jdo.option.ConnectionUserName</name>
  <value>hiveuser</value>
</property>

<property>
  <name>javax.jdo.option.ConnectionPassword</name>
  <value>password</value>
</property>

<property>
  <name>datanucleus.autoCreateSchema</name>
  <value>false</value>
</property>

<property>
  <name>datanucleus.fixedDatastore</name>
  <value>true</value>
</property>
```

Figure 6: Necessary Hive Properties

3.1.2 First tests with Hive

In order to test the performance of Hive and Hadoop in the context of PostgreSQL, a first series of performance tests were made to compare a simple and a double Equijoin-Function between two relation in a small Hadoop-cluster compared to a PostgreSQL-database running on a single instance. These series had nothing to do with the main problem of calculating the constant intervals of temporal data, but was done to find out more about the performance of Hive and Hadoop used on general SQL-queries. The results are shown in Figure 7.

The specifications of the computers during this test were the same, where the cluster consisted of 3 or 5 (can be seen in the column Cluster Nodes) of the same machines as the computer, where the results were calculated on PostgreSQL. The RAM had for each instance a size of 1.7 GB and one virtual core. The results can be seen in the table in Figure 7:

Cluster Nodes	Mappers	Reducer	Operator	Nr of Inputs	Calculation Time in Hive	Calculation Time in PostgreSQL
3	5	1	Join	10’000’000	723s	266 s
3	5	10	Join	10’000’000	674s	
3	5	40	Join	10’000’000	778 s	
5	10	1	Join	20’000’000	897 s	495 s
5	10	5	Join	20’000’000	609s	
5	10	20	Join	20’000’000	562s	
5	10	1	2x Join	20’000’000	1139s	650 s
5	10	5	2x Join	20’000’000	802s	
5	10	15	2x Join	20’000’000	738s	
5	10	20	2x Join	20’000’000	772s	
5	10	40	2x Join	20’000’000	823s	
5	10	100	2x Join	20’000’000	974s	
5	22	1	Join	40'000'000	1921 s	780 s
5	22	5	Join	40'000'000	1097 s	
5	22	10	Join	40'000'000	979 s	
5	22	15	Join	40'000'000	1055s	
5	22	40	Join	40'000'000	1070s	
5	22	1	2x Join	40'000'000	2380s	1112 s
5	22	5	2x Join	40'000'000	1478s	
5	22	15	2x Join	40'000'000	1446s	
5	22	40	2x Join	40'000'000	1539s	

Figure 7: Results of the test series

The times were taken by the integrated time of the Hive-framework.

The test was executed with varying numbers of reducers to find out, if the Hive-framework follows the same rules as an ordinary Hadoop-task, and by their variation better or worse results can be achieved.

In Figure 8, where the data of the table in Figure 7 is summarized, the comparison between Hive and PostgreSQL can be seen more easily. To compare the results, always the time of the Hadoop task with the lowest computation time was taken. This means, the optimal number of reducers was taken.

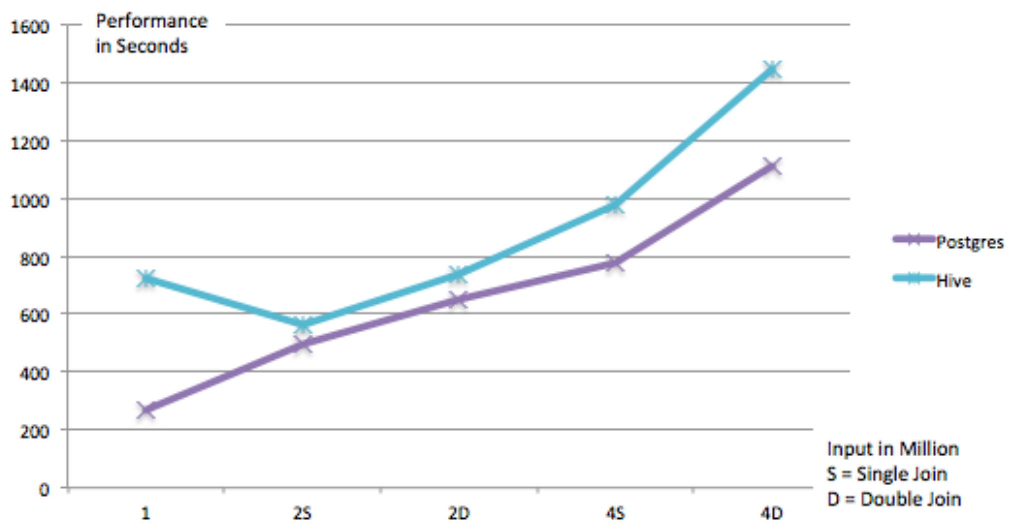


Figure 8: Diagram of the results for the comparison of the Equijoin

As it can be seen, for a simple equijoin-task, the performance of a PostgreSQL-database seems to be better than the corresponding Hive-task. Because both curves, Hive and PostgreSQL, grow in a similar way, the power of Hadoop and Hive does not come into account with this amount of data or with this function. In order to evaluate the performance of Hive in the context of temporal databases, in a next step the queries to generate constant intervals have been analysed.

3.1.3 Constant Intervals with Hive

During the transcription from PL/pgSql to HiveQL some serious, game-changing problems occurred. Because Hive doesn't support all the features PL/pgSql provides, some because they are not yet implemented, some because some problems cannot be processed efficiently by Hadoop and will therefore never be implemented. One of these features is the "aggregate Join", another one is, that Hive cannot perform subqueries in the "where"-clause. Because the "aggregate Join" and a subquery in a "where"-clause are the centre point of the constant-interval-query, Hive is not able to perform this specific query and is therefore not applicable for the computation of constant intervals. A computation of the constant interval problem using Hive is therefore not accomplishable.

3.2 Sorting the Data with Hadoop

3.2.1 The Idea

The expensive query in the context of temporal databases is the query, where the time-points are sorted in ascending order and formed into a relation, where the next time-point is linked with the preceding entry to build the constant interval. The idea behind the sorting is to extract the data out of the database, sort the timestamps in Hadoop, build the intervals with Java and reintegrate it in the database.

3.2.2 The Implementation

Hadoop has a feature, which allows an output-sorting for every reduce-task. The problem is, that if Hadoop itself distributes the data to the different reducers, every output-file is sorted locally, but not the overall data. To solve this problem it is necessary to add a Partitioner to the MapReduce algorithm, in order to tell Hadoop, which key has to be distributed to which reducer. After Hadoop is finished, the data is sorted overall and the different output files can be merged simply. In order to distribute the data to reduce task, it is to some point necessary to know the distribution of these time-points, because otherwise the reducers may not been fully used to capacity.

In the evaluation discussed below, the timestamps were equally distributed between year 1000 and year 1999. Therefore a simple algorithm to distribute the timestamps to each reducer was relatively easy to program. The code of the sorting-application can be seen in Appendix I.

3.2.3 Results of the Sorting-Application

With the MapReduce-Sorting algorithm descriped in chapter 3.2.2, a series of tests were run, in order to verify, that sorting the timepoints with Java and Hadoop gives the user an immense advantage. In the following Figure 7 the results of this test-serie is shown.

Cluster Nodes	Mappers	Reducer	Data sets	Preparation Time	Computation Time	Post-editing time
5	16	1	50'000'000		11 m 16 s	
5	16	5	50'000'000		11 m 14 s	
5	16	10	50'000'000		10 m 31 s	
5	16	20	50'000'000		10 m 51 s	
5	16	100	50'000'000	4 m 25 s	14 m 20 s	37 s
5	16	1	10'000'000		3 m 46 s	
5	16	5	10'000'000		3 m 14 s	
5	16	10	10'000'000		3 m 14 s	
5	16	20	10'000'000		3 m 28 s	
5	16	100	10'000'000	1 m 13	7 m 11 s	29 s
5	16	1	1'000'000		1 m 14 s	
5	16	5	1'000'000		1 m 17 s	
5	16	10	1'000'000		1 m 26 s	
5	16	20	1'000'000		1 m 39 s	
5	16	100	1'000'000	14 s	5 m 49 s	16 s
5	16	1	100'000		56 s	
5	16	5	100'000		1 m 06 s	
5	16	10	100'000		1 m 19 s	
5	16	20	100'000		1 m 42 s	
5	16	100	100'000	7 s	5 m 23 s	29 s
5	16	1	20'000	7 s	47 s	10 s
5	16	5	20'000		1 m 02 s	
5	16	1	1'500	7 s	46 s	8 s
5	16	5	1'500		59 s	

Figure 9: Table of the results

The time was taken by the system timing tool. The specifications of the Hadoop cluster were the same as in test in chapter 3.1.2. Every data set consisted of a pair of two timestamps, a starting timestamp and an ending timestamp. That means in reality, 100 data sets mean 200 timestamps, which have to be sorted and 199 constant intervals can be built, if all timestamps are unique.

To evaluate, when Java and Hadoop become more efficient than PostgreSQL, the query to generate constant intervals was run on a single instance with PostgreSQL, which had the same specifications than one node in the Hadoop cluster in the previous test (Figure 9).

Number of data sets	Number of timestamps	Computation time in s
100	200	0.41 s
500	1’000	4.1 s
1’500	3’000	41s
5’000	10’000	6 m 38 s
10’000	20’000	16 m 13 s
20’000	40’000	32 m 24 s

Figure 10: Table of the results in PostgreSQL

Comparing the two tables in Figure 9 and Figure 10, it can be seen, that around 1’500 data sets, which means about 3’000 individual timestamps, Hadoop becomes equally efficient than PostgreSQL. Hadoop has an advantage when the number of data sets increases above this amount, because the computation time increases by far not as fast as the time in PostgreSQL. The computation time in Hadoop remains at about one minute until approximately 1’000’000 data sets, because Hadoop has an overhead of this amount of time, and the effective calculation time does not influence the overall time significantly below this limitation.

3.3 Improving the query in PostgreSQL

The query in PostgreSQL, which was used in the previous chapters, was not optimized fully according to the specific problem of calculation constant intervals. It was a brute-force approach to the problem. In the following chapter the results, achieved in Hadoop, will be compared to a modified query in PostgreSQL. The idea behind this new query is, that the table in PostgreSQL is sorted by the “order by” step in the query, and afterwards each row in the table is connected with the next entry in the table to form a constant interval. The function, implemented in PL/pgSql, is shown in Figure 11:

```
create or replace function sortInterval(refcursor) returns setof refcursor as '
Declare
        a Date;
        b Date;
        ref cursor for select t from time order by t asc;
Begin
        open ref;
        fetch ref into a;
        close ref;
        for recordvar in ref loop

                fetch ref into b;
                insert into intervals values(a,b);
                a := b;
        end loop;
        return next $1;
end;
' Language plpgsql;
```

Figure 11: Function to build constant intervals with cursor

The results of a performance test with this function, and its comparison to Hadoop, can be seen in Figure 12.

Number of Timestamps	Time of Cursor Function in minutes	Corresponding Time in Hadoop in minutes including preparation time
10'000	1 s	1 m 04 s
100'000	4 s	1 m 32 s
1'000'000	55 s	1 m 44 s
10'000'000	13 m 33 s	4 m 56 s
50'000'000	53 m 15 s	15 m 33 s

Figure 12: Table of the results in PostgreSQL with the cursor function compared to Hadoop

As it can be seen, the new function performs a lot better than the brute-force query used in the previous chapters, due to the new architecture of the query/function and its optimization to the specific problem. The more interesting question is, how the performance is compared to Hadoop. As it can be seen in Figure 12, up to approximately 1'000'000 single timestamps, the function is more efficient. In Figure 13 the tipping point can be approximately estimated:

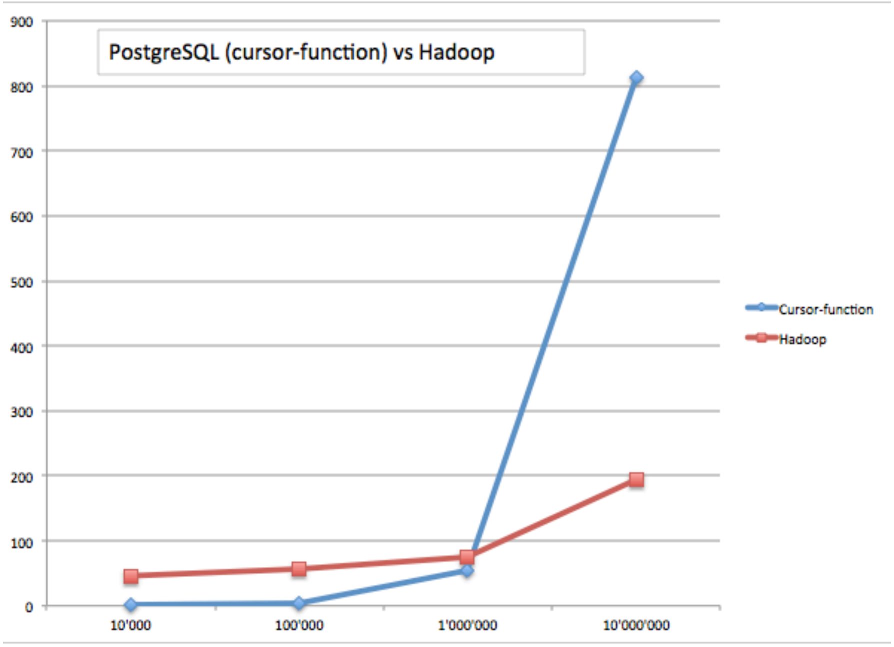


Figure 13: PostgreSQL (cursor-function) vs Hadoop

PostgreSQL with this modified approach to calculate constant intervals is more efficient up to a specific amount of timestamps. This border of 1'000'000 timestamps is much higher than with the brute-force approach used in the previous chapters. But with an increasing amount of timestamps beyond this point, it still can not achieve the advantages in efficiency that Hadoop provides.

3.4 The comparison of the different approaches

In the following Figure 14, the results of the brute-force approach, the cursor function and the approach with Hadoop are compared to each other. The fastest time to perform the computation for each row is bold:

Number of Data sets	Brute-force approach in PostgreSQL	Cursor function in PostgreSQL	Hadoop including Preparation time
1’500	41 s	**1 s**	1 m 01 s
20’000	16 m 13 s	**1 s**	1 m 04 s
100’000	not computed	**4 s**	1 m 32 s
1’000’000	not computed	**55 s**	1 m 44 s
10’000’000	not computed	13 m 33 s	**4 m 56 s**
50’000’000	not computed	53 m 15 s	**15 m 33 s**

Figure 14: Comparison of the results

3.5 The testing environments

Because a Hadoop-cluster with multiple instances could not be set-up due to lacking of resources, for the tests in chapter 3 a Hadoop-cluster from Amazon Web Services (http://aws.amazon.com/) was rented. The cluster was already set-up with Hadoop 1.0.3 and could be used via a SSH-Connection.

4 Conclusion

Hadoop is a very powerful framework, which can compute large amounts of data with relative simple technics. It is only needed to program 40 – 60 lines of code to write a powerful application. Even not so skilled programmers can use Hadoop to master the computation of data. Hadoop has some minor stumbling blocks during the installation process, but with a bit of experience in this environment, the framework is relatively simple and easy to use, where the benefit of Hadoop can be immense.

Relating to the specific problem of computing the constant intervals of temporal data, Hadoop can be very advantageous. While the using of Hive was a dead end, because of the necessity to aggregate-join some relations, the extraction of the data out of PostgreSQL can be viewed as a success. The use of Hadoop may decrease the time of calculating constant intervals, even if the time to extract, transform and import the data again is added to the sorting time.

The tests, conducted in chapter 3.2.3, showed, that a separate computation of constant intervals for temporal databases in Hadoop becomes more efficient at around 2’500 data sets (5’000 individual timestamps). The time to extract, alter and reintegrate the data in and out of a PostgreSQL-database is already included in this calculation.

As it is shown in this Vertiefung, the application of an external method to compute constant intervals can be very useful above a certain number of data sets. Hadoop is a very powerful tool, which is able to perform the necessary computation and has been developed to handle a large amount of data. The only negative point is, that in order to use Hadoop properly, the data has to be partitioned correctly, which can be difficult sometimes, if the distribution of the data is not known.

Appendix I

The following source code shows the Hadoop part of the application to sort the timestamps. Input is one ore more text files with one timestamps in each row, the output are a number of text files equal to the number of reducers chosen.

```
public class LineIndexer {

	//Define the number of reducers for your Hadoop-Job
	public static int numberOfReducer = 1;

	//The Mapper-class
	public static class LineIndexMapper extends MapReduceBase implements
			Mapper<LongWritable, Text, Text, Text> {

		private final static Text word = new Text();
		private final static Text location = new Text();

		public void map(LongWritable key, Text val,
				OutputCollector<Text, Text> output, Reporter reporter)
				throws IOException {

			FileSplit fileSplit = (FileSplit) reporter.getInputSplit();
			String fileName = fileSplit.getPath().getName();
			location.set(fileName);

			String line = val.toString();
			StringTokenizer itr = new StringTokenizer(line.toLowerCase());
			while (itr.hasMoreTokens()) {
				word.set(itr.nextToken());
				output.collect(word, new Text(location));
			}
		}
	}

	//The Reducer-class
	public static class LineIndexReducer extends MapReduceBase implements
			Reducer<Text, Text, Text, Text> {

		public void reduce(Text key, Iterator<Text> values,
				OutputCollector<Text, Text> output, Reporter reporter)
				throws IOException {
			output.collect(key, null);
		}
	}

	//The Partitioner-class
	// For testing reasons, 1,5,10,20,100 numbers of reducers can be chosen
	public static class MyPartitioner implements Partitioner<Text, Text> {

		public int getPartition(Text key, Text value, int numReduceTasks) {

			int reduc = 0;

			// For 10 reducers:
			if (numberOfReducer == 10) {
				char century = key.toString().charAt(1);
				reduc = Character.getNumericValue(century);
			}
			// For 5 reducers:
			else if (numberOfReducer == 5) {
				char century = key.toString().charAt(1);
				int century_int = Character.getNumericValue(century);
				char decade = key.toString().charAt(2);
				int decade_int = Character.getNumericValue(decade);
				int temp = 10 * century_int + decade_int;
				reduc = temp / 20;
			}

			// For 20 reducers:
			else if (numberOfReducer == 20) {
				char century = key.toString().charAt(1);
				int century_int = Character.getNumericValue(century);
				char decade = key.toString().charAt(2);
				int decade_int = Character.getNumericValue(decade);
				int temp = 10 * century_int + decade_int;
				reduc = temp / 5;
			}

			// For 100 reducers:
			else if (numberOfReducer == 100) {
				char century = key.toString().charAt(2);
				reduc = Character.getNumericValue(century);
			}

			else {
				reduc = 0;
			}

			return (reduc);
		}

		public void configure(JobConf arg0) {
		}
	}

	/**
	 * The actual main() method for our program; this is the "driver" for the
	 * MapReduce job.
	 */
	public static void main(String[] args) {

		JobClient client = new JobClient();
		JobConf conf = new JobConf(LineIndexer.class);

		conf.setJobName("LineIndexer");

		conf.setOutputKeyClass(Text.class);
		conf.setOutputValueClass(Text.class);

		FileInputFormat.addInputPath(conf, new Path("input"));
		FileOutputFormat.setOutputPath(conf, new Path("output"));

		conf.setMapperClass(LineIndexMapper.class);
		conf.setReducerClass(LineIndexReducer.class);
		conf.setPartitionerClass(MyPartitioner.class);

		conf.setNumReduceTasks(numberOfReducer);
		client.setConf(conf);

		try {
			JobClient.runJob(conf);
		} catch (Exception e) {
			e.printStackTrace();
		}

	}

}
```