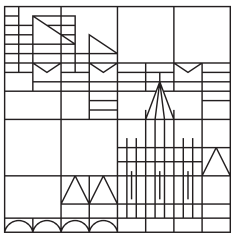
Using Map and Reduce for Querying Distributed XML Data

Lukas Lewandowski

Master Thesis in fulfillment of the requirements for the degree of
Master of Science (M.Sc.)

Submitted to the Department of Computer and Information Science
at the University of Konstanz

Universität Konstanz



Reviewer:
Prof. Dr. Marc H. Scholl
Prof. Dr. Marcel Waldvogel

Abstract

Semi-structured information is often represented in the XML format. Although, a vast amount of appropriate databases exist that are responsible for efficiently storing semi-structured data, the vastly growing data demands larger sized databases. Even when the secondary storage is able to store the large amount of data, the execution time of complex queries increases significantly, if no suitable indexes are applicable. This situation is dramatic when short response times are an essential requirement, like in the most real-life database systems. Moreover, when storage limits are reached, the data has to be distributed to ensure availability of the complete data set. To meet this challenge this thesis presents two approaches to improve query evaluation on semi-structured and large data through parallelization. First, we analyze Hadoop and its MapReduce framework as candidate for our distributed computations and second, then we present an alternative implementation to cope with this requirements. We introduce three distribution algorithms usable for XML collections, which serve as base for our distribution to a cluster. Furthermore, we present a prototype implementation using a current open source database, named BaseX, which serves as base for our comprehensive query results.

Acknowledgments

I would like to thank my advisors Professor Marc H. Scholl and Professor Marcel Waldvogel, who have supported me with advice and guidance throughout my master thesis. Thank you also for the great possibility to work in both the DBIS and the DISY department and for provisioning a comprehensive workplace and all necessary work materials.

Also I would like to thank Dr. Christian Grün and Sebastian Graf for the many helpful discussions regarding my research work, the technical support especially concerning BaseX, and the interesting insights into the challenging life of a researcher.

This work was supported by the Graduiertenkolleg *Explorative Analysis and Visualization of Large Information Spaces*, Computer and Information Science, University of Konstanz. I would like to thank the Graduiertenkolleg for its worthwhile support.

I also want to thank Patrick Lang and Oliver Egli for the possibility to study with them. Both have become very good friends during the last years and both were good teammates during my studies. Special thanks goes to Anna Dowden-Williams for numerous useful comments to this manuscript. Thanks also to Michael Seiferle, Alexander Holupirek and the rest of the BaseX team for the nice time at the DBIS group. Of course, I want to thank also Sebastian Belle, Thomas Zink, and Johannes Lichtenberger for the helpful ideas and discussions concerning my thesis.

Finally, I want to thank my parents for their great support and the financial help through my studies. Special thanks also goes to Susanne for her understanding and support.

Contents

1 Introduction

1.1 Motivation

Semi-structured data like XML is widely used for example as data exchange format, message format like SOAP [GHM+07], configuration files, logging, or even as storage format in companies and public institutions. Advanced XML databases like BaseX, Treetank or eXist-db are designed to store XML and evaluate queries performing complex data analysis on many XML files, but when data is growing fast, several issues must be considered: Firstly, reliability using replication techniques and data availability. Secondly, an essential point is information retrieval. When the size of available information is growing, the time of performing complex queries on the data is increasing due to many I/O accesses. This is still true even, if several indexes are applied due to the size of indexes that do not fit in commodity main memory and have to be swapped out. Furthermore, many updates require to rebuild or update the existing index structure consuming a lot of time for large data collections. Thirdly, concurrent read and write operations slow down query performance when many clients try to access a single database due to locking of database instances. In a distributed storage environment load balancing achieves more concurrent reads and writes. Moreover, the whole distributed database is not locked. Fourthly, when database sizes exceed the computer storage limit, either additional storage must be added or data has to be distributed to several computers. In most relational databases there are several approaches for distribution based on row or column partitioning, but XML has a hierarchical structure making distribution more difficult. XML databases can be divided into two groups: very large XML instances, having to be fragmented based on several splitting algorithms considering the structure of an XML document and large collections of small XML files where XML files have no complex structure. Distributed querying is a challenge in both cases because of the dynamic nature of XQuery expressions. This master thesis focuses on distribution and querying of large XML collections consisting of many XML files due to the fact that most XML files have sizes of only some MB. Furthermore, large XML instances are often constructed of many records on child

level, which can be easily transformed to XML collections.

1.2 Contribution

Since the requirements of a complete distributed XML database system are comprehensive, this thesis is not able to cover all components. Hence, the focus is on distribution and querying of XML collections using an XML database as backend. The main contributions of this elaboration are:

- Analysis of the usability of Hadoop's MapReduce framework for XML query evaluation.
- Introduction of an own implemented distribution and querying approach with performance evaluation.
- Integration into an XML database using the example of BaseX.

1.3 Outline

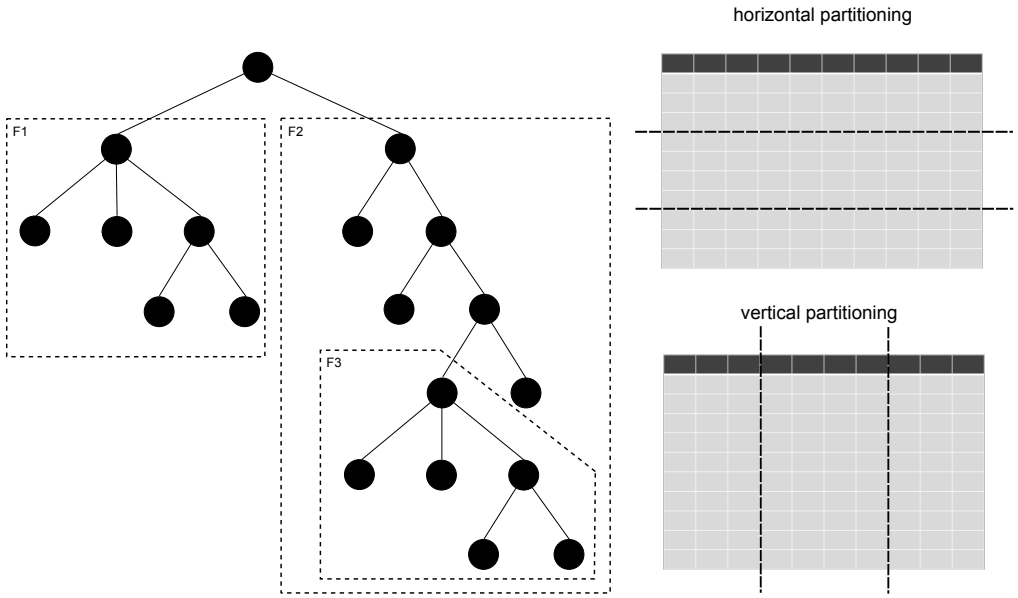
This master thesis is organized as follows: Chapter 2 introduces some basic knowledge on XML collections, XQuery and XML databases. Chapter 3 distances this work from other related research approaches. In Chapter 4, we describe the author's first approach, the Hadoop MapReduce approach. Chapter 5 depicts the alternative approach considering distribution and querying of XML collections in detail, and it introduces some optimizations. The results and the performance evaluations of the implemented prototype are also presented in Chapter 5. Chapter 6 describes the integration into BaseX and depicts an example query workflow. In Chapter 7, we give a preview of the planned future work. Finally, a conclusion completes this master thesis in Chapter 7.

2 Preliminaries

2.1 Semi-Structured Data

Semi-structured information is not as powerful as structured information - or isn't it? Information that is strongly structured has a defined schema, which allows assigning data types to all information. In a relational world, we first define a schema for the data we want to store. Afterwards, the data is mapped to the defined schema and stored in the database. The eXtensible Markup Language (XML), introduced in [BPSM+08], is a hierarchic semi-structured format for exchanging information. One important benefit of XML is that it does not necessarily need a defined schema to store information in a database. The exchange format can contain meta information about the structure, e.g., data types, to describe all carried information. Furthermore, XML allows to type data as mixed content, e.g., to mix up further meta information within a text paragraph. It is also possible to omit data typing in the exchange format itself for example to allow a database implementation to type the information dynamically. It is therefore not possible to state that semi-structured information is weaker than structured information, but it allows a more complex data definition than in classical relational structured models. Another important XML property is the ordering of elements in the tree representation, which has to be satisfied also when an XML document is fragmented and distributed. Relational databases are partitioned in most cases horizontally or vertically as stated in [CNP82] and [NCWD84] when the database has to be distributed to several server instances due to the defined data table schema. XML is much more difficult to partition [Gra08]. An XML structure is not regular by default. Depending on the carried information, an XML tree can have a text oriented structure, a document-centric file or a data oriented structure, a data-centric XML file. Thus, XML fragmentation must use the most suitable partitioning algorithm depending on the structure of the tree. A sample comparison of both, relational database partitioning and XML partitioning is depicted in Figure 2.1.

In the context of XML databases the storage representation of an XML document obviously also has an impact on the fragmentation techniques. One of the most often chosen



(a) XML partitioning. (b) Relational table partitioning.

Figure 2.1: Fragmentation of documents based on tree structure and relational table.

storage representations is in fact a table representation of the initial document, i.e., BaseX [Grü10] uses such an approach. Consequently, some could suggest to not partition the XML representation, but to partition the storage table. This approach works well for a particular database implementation, but is not a universal approach for others, especially for not-table storage representations. Furthermore, even if all implementations would use a table representation, we would obtain a high coupling between the tables and the distribution approach. Each change of the storage representation needs an adjustment of the table distribution algorithm as well. Hence, this thesis does not focus on relational table partitioning.

2.2 Collections of XML Documents

A collection of XML documents is a group of assigned documents identified by a URI. By default the order of retrieving documents from a collection is completely implementation dependent. In [MMWK10] the XQuery and XPath 2.0 fn:collection function is introduced, which allows accessing collections and documents through the query languages. Furthermore, the fn:collection function is stable by definition. Repeated calls to this function will return the same result. In BaseX both, a collection or a single docu-

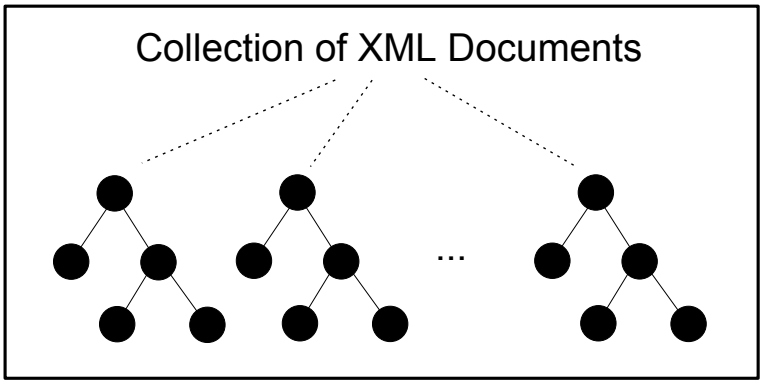


Figure 2.2: XML Collection.

ment is mapped to a database instance. Thus, it is possible to store an XML document as database and later add additional documents. As stated in [Grü10] large single documents are rare. Most XML files have the size of only few MB. However, collections of similar XML documents reach sizes of several GB or even TB. Furthermore, if several different collections on a database server exist, distribution must be considered. As this situation is more common, this master thesis focuses on querying and distribution of XML collections. Therefore, partitioning techniques for single documents, as introduced in section 2.1, are not elaborated in detail, but this master thesis' approach serves as a basis and can also be extended to single document partitioning, as well. An example of a collection of XML documents is depicted in Figure 2.2. In real-life, a collection of RSS feeds could serve as base for, e.g., text analysis.

2.3 XQuery

The XQuery language, as specified in [BCF+07], is the de-facto standard for advanced querying in XML context. One special fact of XQuery is that it allows to address several XML documents or collections from one XQuery expression residing in the case of distribution on several database servers. Therefore, identifying databases is first feasible during compile time. Furthermore, XQuery expressions are able to choose document or collection names dynamically. When using such complex dynamic features in a query expression, identifying responsible databases and therefore responsible database servers, is a prerequisite to enable parallel evaluation of sub queries on the different database servers. Since it is possible to address several databases and collections in one query ex-

pression, a completely parallel evaluation of all sub queries is not guaranteed, because sub queries could require results from its predecessor query to evaluate their own query expression.

2.4 MapReduce

MapReduce was first introduced by Dean and Ghemawat in [DG04] and is currently adopted in many distributed systems. It is a model for parallel and distributed computation. The model mainly consists of two functions: *map* and *reduce*, which have to be implemented by the user. Figure 2.3 depicts an overview of the used architecture. The map function receives data from a local machine as two input parameters, a key as identifier and a value as record (*Input* and *Map Phase*). Afterwards, the map function performs a user-defined computation on the record and outputs the result as new key-value-pair. All results from the map functions, corresponding to the same key, will be appended to a list of values. This is the transition between the *Map Phase* and the *Reduce Phase*. This new list of key-value-pairs serves during the reduce phase as input parameter for the reduce functions. The user defined reduce function analyses the list of values and computes the complete result that is returned to an output file.

All map and reduce functions will be executed in parallel on worker nodes containing the necessary data. The data has to be partitioned into key-value-pairs (records) to serve as input for the MapReduce job. Furthermore, a *combiner* can be defined by the user to support a local reduce-like function to avoid unnecessary network communication between each map and reduce function. A central job tracker is coordinating all MapReduce jobs and takes additional bookkeeping to ensure correct and finite job execution. Additionally, the MapReduce framework ensures a failure-free job execution even when a working node dies, e.g., as consequence of a hard disk failure. A detailed description of the MapReduce idea is given in Chapter 4.

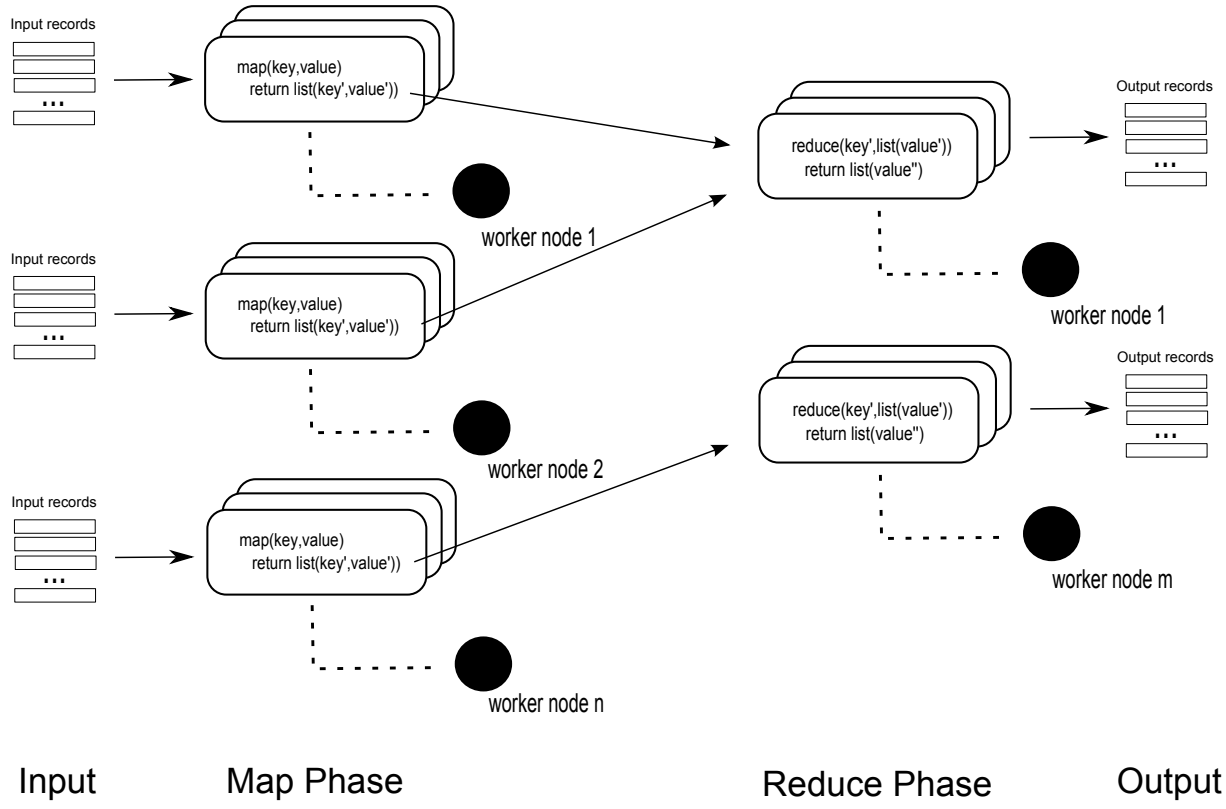


Figure 2.3: MapReduce architecture.

3 Related Work

Distribution and querying of distributed data is an often discussed topic in the research area of relational databases. Extensive work has been done on horizontal and vertical fragmentation of relational data as in [CNP82] and [NCWD84]. Kossmann [Kos00] introduced several approaches how distributed data processing is managed in a relational world.
Single XML documents have another structure than relational data. Their tree structure must not be balanced and thus, fragmentation is not always possible as data organized in tables. Several researchers considered the problem of XML fragmentation: Bonifati and Cuzzocrea [BC07] introduced an approach based on structural constraints of the XML tree. One example constraint, the width, is defined in advance to perform fragmentation. Depending on the defined values the fragmentation quality varies. Bremer and Gertz [BG03] presented a schema-based fragmentation approach considering the element tag occurrences in the tree. They also describe the possiblity to query the distributed XML data using only a subset of XPath. Abiteboul et al. [ABC$^+$03] introduced a different approach, in which static XML documents are distributed and dynamic content is injected using web services. In [GKW08] the authors presented various split algorithms for XML documents. The *PostorderSemSplit* achieved good results, but no algorithm performed optimally for all XML document types.
There are several approaches focussing on querying the distributed XML data [BF05, PM02, BCFK06], but none of them uses the complete XQuery standard as querying language.
When considering the NoSQL ideas, Hadoop and its MapReduce framework are identified as most used distributed computing framework within research. Several approaches are based on Hadoop and MapReduce to solve, e.g., indexing or information retrieval problems [KATK10, ZZYH10, VCL10]. Their focus is solving problems on large data sets, but they do not consider query response times for database query performance, which are in general in milliseconds. Furthermore, only few approaches can be applied for XML querying. Only Khatchadourian et al. [KCS11b] support MapReduce execution using XQuery (ChuQL). They extended the XQuery language to define MapReduce

processes, which will be executed on top of Hadoop. In [KCS11a] they presented an interesting approach and their evaluation results for a large XML data set of several hundred GB. The performance was quite good, but they do not focus on minimal query execution time of distributed XML data.
XQuery is the de-facto standard for complex querying within XML data. There are also approaches implemented directly on top of an XQuery processor. The *Distributed XQuery* (DXQ) idea is one of the most interesting and introduced in [FJM+07a] and [FJM+07b]. The authors extend the XQuery specifcation by DXQ grammer extensions to support distributed querying based on XQuery expressions. One advantage is that one does not have to use a distributed file system like the approaches based on Hadoop. The focus of the DXQ approach is to offer web services implemented in XQuery to support applications like their introduced Domain Name System (DNS) resolution example. They do not focus on querying large XML data sets and they do not present any experimental results. Since there is an intersection of ChuQL, DXQ and our proposed approach, the following table provides an overview about similarities and differences. Values in brackets are not known.

Characteristics	ChuQL	DXQ	Our Approach
Distribution out of XQuery	Yes.	Yes.	Yes.
Extension of XQuery language	No.	Yes.	No.
EXPath support	No.	No.	Yes.
XML database support	No.	No.	Yes.
Implementation architecture similarities	No.	(No.)	(No.)
Experimental results	Yes.	No.	Yes.
Delegation of reduce step	Yes.	No.	Yes.
Querying of large distributed data	Yes.	No.	Yes.

Table 3.1: Differences between ChuQL, DXQ and our approach.

4 Querying and Distribution of XML Data using Hadoop and MapReduce

4.1 Hadoop and XML

There are three main principles in distributed environments to fulfill the distribution of data: a centralized approach with a master node that is responsible for the coordination of the network, a decentralized approach where all network nodes are equal and responsible to forward requests, and a hybrid approach of both where, e.g., sub networks are coordinated by one master node and the master nodes are connected through a peer-to-peer network. Each of the introduced approaches has strengths and weaknesses concerning several topics like single point of failure, performance or robustness. Currently, no free available open source XML databases like BaseX, Treetank or exist-db, support distribution of large XML collections or documents to a cluster. One possibility is to implement a classic distributed principle as mentioned before from scratch. On the other hand, XML databases belong to the NoSQL section as depicted in [Edl11], and currently there are several distributed computation models and distributed databases. The most common one is Hadoop [Fou11], which consists of a distributed file system (HDFS) and the computation model MapReduce. Hadoop's HDFS and MapReduce framework APIs are written in Java and are based on Google's Google File System (GFS) and MapReduce [GGL03, DG04]. HDFS is responsible to distribute data through a master node to data nodes. The master node is responsible to coordinate requests and manages the free resources on each data node. Furthermore, the master node is not responsible for transferring data to the data nodes from a client. Its assignment is to locate responsible data nodes and to provide data node locations to the HDFS client program. The client program then distributes the incoming data to the data nodes. Thus, the master node is only responsible for coordinating requests, managing of storage availability, and for storing snapshots in case of a master failure. In such a failure a new master node will be initialized with the data from the last snapshots. The distributed data is by default organized in block sizes of 64 MB. MapReduce is responsible to perform parallel

computation on the several data nodes containing the distributed data. The *map* function receives records from the data node as key/value pairs and the computation results are written as intermediate results to the output and distributed and replicated through HDFS. The *reduce* function receives all intermediate results associated with a key and performs result aggregation, which is written to the final output and distributed and replicated as well.
The original idea was to use Hadoop's MapReduce framework to extend native XML databases with distributed querying techniques. Therefore, a prototype with Hadoop was developed and evaluated against an alternative implemented centralized REST [Fie00] approach, based on JAX-RX [GLG10], to compare distribution of XML files and querying afterwards. In the next section both prototypes are described in detail.

4.2 Prototype

Since this thesis focuses on large collections of XML documents, HDFS was configured to retrieve a folder containing all XML files as input used for the distribution. HDFS distributes these files to the data nodes, one file per block as long as the file is smaller or equal than the defined block size. Furthermore, HDFS ensures replication with a factor of three and load balancing in the distributed file system. After distribution, MapReduce is performed to evaluate queries on the distributed XML files. To ensure the XML well-formedness, each XML document is mapped as one single map input record. The XML document path specifies the input key. The XML record is imported in either Treetank, BaseX or Saxon and the query is evaluated on each document within this map function. The results are written as intermediate output. The reduce function collects all query results and prints the complete result to the output. Figure 4.1 depicts this initial situation.

In phase 1, the XML input files will be distributed to the data nodes. The *NameNode* is responsible for allocation of free blocks on the data nodes and sends the block locations to the client node, labeled with the arrows 1 and 2. The client node then writes its input to the corresponding data node, depicted as arrow 3. Afterwards the written block will be replicated through the data node, see arrows 4. Thus, single hard disk failures do not affect the Hadoop execution.

In phase 2, a MapReduce job is initialized for each input file. The client sends its MapReduce request to the *Job Tracker*, see arrow 1. The Job Tracker allocates the data position

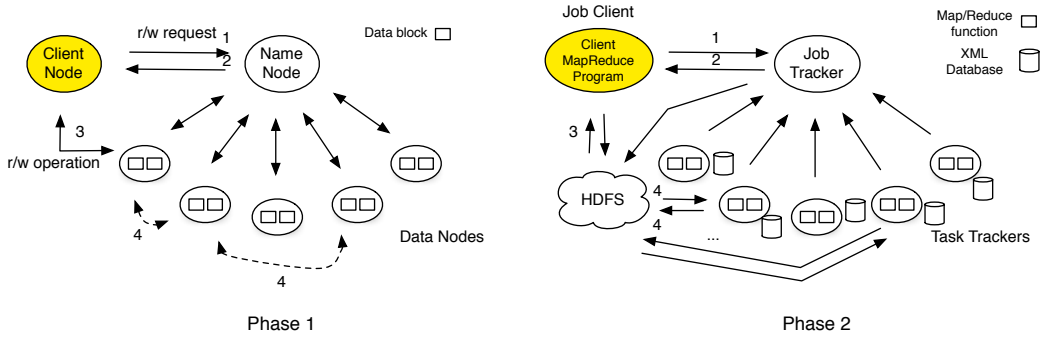


Figure 4.1: Phase 1: Distribution of XML input files via Hadoop's HDFS. Phase 2: Importing of distributed XML files to XML databases.

in HDFS where the written map and reduce functions needed to be stored, arrows 3 and 4. Afterwards, *Task Trackers* are assigned to perform the map and reduce computation on the local machines. The map function receives as input one single XML file and imports it to the local XML database. The reduce function is responsible to write a list of all stored XML files in the XML databases. When a query has to be performed, another MapReduce job must be initiated as depicted in phase 2 of Figure 4.1. The map function evaluates the query on the local XML database and writes the query result as intermediate result to the output. The reduce function receives the query results and combines all to the complete results.

The second idea was to use a REST based implementation for distribution and querying XML files. This prototype is also a centralized approach with master/slave dependencies like Hadoop. A designated master node consists of a REST interface for incoming client queries and XML documents for distribution. The master node is using a hash map for addressing the data nodes. Distribution is performed in round-robin manner to each registered data node. Each XML file is sent by the master node to one or more data nodes through an HTTP PUT or HTTP POST request. On the data nodes, a web server listens for incoming requests and performs importing of new XML files to Treetank or BaseX. Queries are sent in parallel from the master node to all data nodes through an HTTP GET or HTTP POST. The results, evaluated by the data nodes are sent back to the master node. The collected results are combined and forwarded to the client. This situation is illustrated in Figure 4.2.

The results of the comparison are described in the next section.

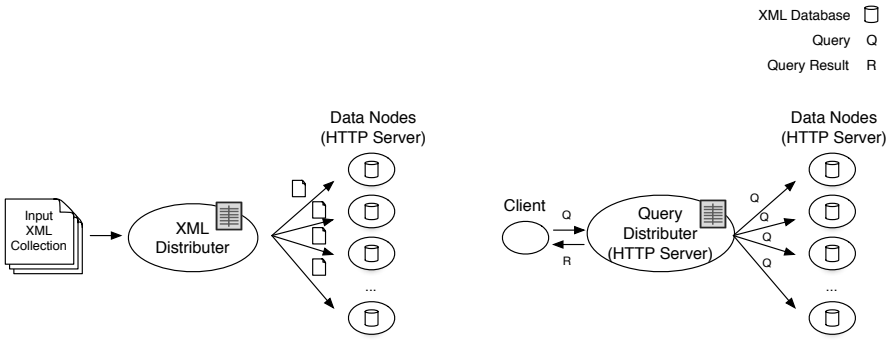


Figure 4.2: a) distribution of XML collections through a master node. b) querying the distributed XML sub collections.

4.3 Results

The first evaluation was performed with Saxon in main memory and without any XML database to omit I/O accesses during map and reduce execution. For test purposes, a collection of 10,000 small XML files (200 MB), each about 20 KB size, was distributed with Hadoop. Afterwards, Saxon's query processor evaluated a test query, searching for a special text content, on each XML file (map function) and wrote the query result as intermediate result to the output. Then, the reducer collected all sub results and combined them to the complete result. The test was executed on four virtual Linux Debian computers with 1 GB RAM each. This 200 MB collection was evaluated in approx. *six to seven hours* on a four data nodes cluster instead of only few seconds or milliseconds.
To analyze the long execution times of the MapReduce functions several amounts of *empty* map and reduce functions are evaluated on different data node cluster sizes. Figure 4.3 describes the results of the empty MapReduce execution. Each test was performed ten times and the averages were used in the diagram. Only one map and reduce function (one distributed input file) needed at least 25 seconds. The execution does not grow significantly between 1 and 40 input files, but afterwards, it increases linearly with the number of input files. 100 map jobs on three data nodes needed about 120 seconds and 1000 map jobs on a cluster of ten nodes needed about 590 seconds. These results show that MapReduce is not able to cope with many input files and the execution time is not comparable with database execution times, where query results have to be evaluated in milliseconds or few seconds. This large overhead for executing MapReduce functions is due to following issues:

- For each input file at least one MapReduce job is needed.

- Each MapReduce job is maintained by the Job Tracker node.
- MapReduce job sends progress to Job Tracker node.
- The user defined map/reduce function will be sent as jar file to at least 10 data nodes (jar size is important for network traffic).
- Task Trackers send each 2-3 seconds a heartbeat to the Job Tracker and afterwards they get a new job assigned.
- Task Trackers read the user jar from HDFS and copy it to the working directory of the Task Tracker.
- For each map and reduce function a new JVM will be initialized.
- Reducer does not start before all map functions have been executed successfully.
- Intermediate results from map functions will be written to HDFS to ensure persistence when some Task Tracker dies.
- Final results will be written to HDFS, too.

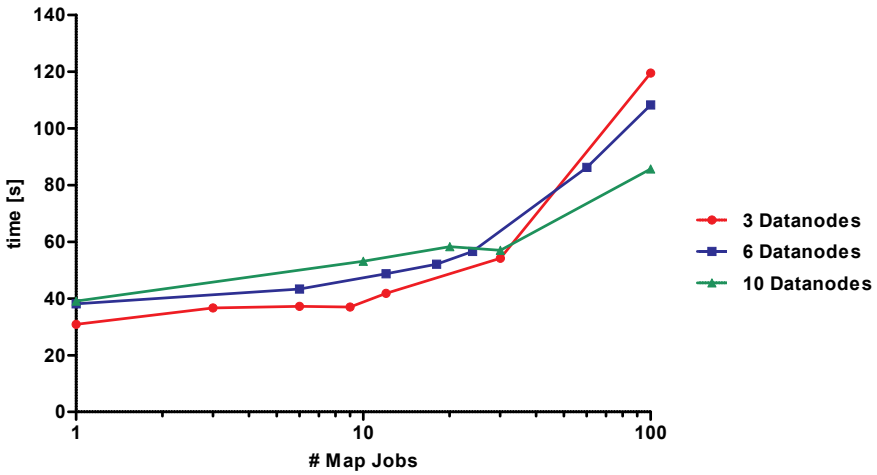


Figure 4.3: MapReduce evaluation on different data node cluster sizes.

To cope with the *many small files Hadoop problem*, sub collections with sizes smaller than or equal the Hadoop chunk size (64 MB by default) were created in a pre-processing step. Afterwards, the sub collections were distributed to the cluster. The MapReduce jobs then must consider $\frac{collectionsize}{chunksize}$ input files. In the map functions the distributed sub collections are imported into Treetank and BaseX. The reduce functions collect meta information about stored XML files and write this meta information to the output. To compare distribution times with an XML database cluster, three collection sizes (1 GB, 10 GB

and 25 GB) were generated with XMark [SWK+02] and analyzed with four distributed approaches and one local BaseX instance for reference. The distributed approaches are designed as follows:

- Hadoop 64 MB Chunks: The standard chunk size as described above.
- Hadoop 256 MB Chunks: Increased chunk size to 256 MB.
- REST Single File: The REST distribution approach as described in Figure 4.2 a, where no chunk size is defined and each small input file will be distributed to a data node.
- REST 64 MB Chunks: The REST distribution approach, where sub collections of sizes of about 64 MB are distributed to data nodes.

This situation is depicted in Figure 4.4. As expected, a local import into an XML database as BaseX (Local BX) has the shortest import execution time. Hadoop 64 MB and 256 MB chunk size approaches perform distribution and XML import in almost the same time. The REST approaches differ from each other. The REST Single File approach loses a lot of time for opening and closing an HTTP connection. The REST 64 MB Chunks approach performs similar to Hadoop distribution.

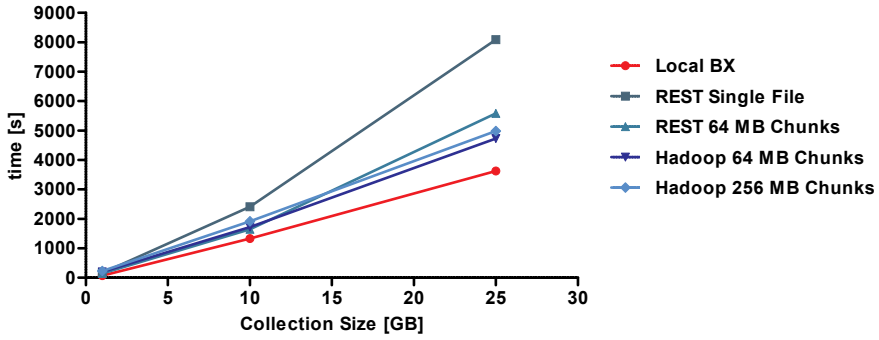
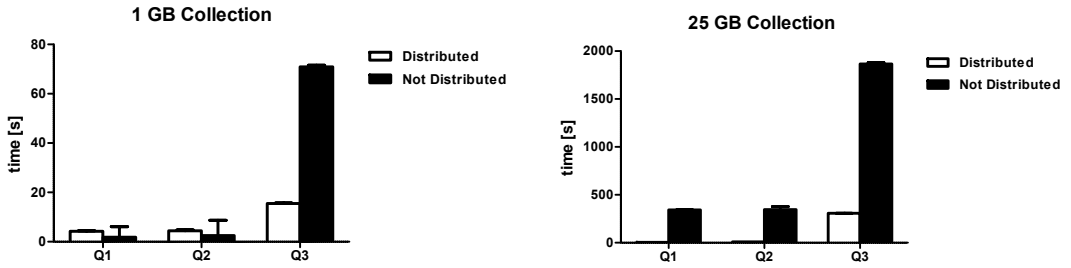


Figure 4.4: Comparison of importing XML collections to BaseX in a distributed and non-distributed environment.

As mentioned above, query evaluation within MapReduce is too slow and therefore useless for database query needs. As a consequence, the REST Query Distributer performs querying, which is presented in Figure 4.2b. For test purposes three queries have been designed to measure distributed query evaluation (distributed on eight heterogeneous physical data nodes) compared to local query evaluation.

```
Q1: count(/site/regions/africa/item[location/text()='United States'])
```



(a) 1 GB XMark collection.

(b) 25 GB XMark collection.

Figure 4.5: Query execution on 1 GB and 25 GB XMark collection in a distributed and non-distributed environment.

```
Q2: count(/site/regions/descendant::item[quantity/text()>1])

Q3: count(/site/descendant::text[. contains text 'prevent'
    using stemming])
```

Q1 describes a simple XPath query matching for a particular text node (applying a text index). Q2 describes another simple XPath that searches for nodes with integer values, which are above a certain constant. Q3 is a full text query using stemming, which matches nodes containing a particular string using stemming functions. As can be seen in Figure 4.5, with regard to query execution times, small collection sizes do not benefit from distribution except for Q3. This is due to the used text index, which has a constant execution time. Q3 does benefit from distribution, because of a missing full text index. In (b), all queries benefit from distribution due to the larger collection size, where the text index is too large to fit into main memory, when using commodity computers for the cluster. The queries are surrounded with the *count* XPath function to reduce network traffic to guarantee that distributed querying time is mainly to query evaluation and creating network connections between the client and the servers.

4.4 Conclusion

The above section analyzed the usability of Hadoop's MapReduce framework for XML database distribution and querying features. The first results showed that Hadoop cannot cope directly with many small files as input for its MapReduce execution. This is due to the fact that Hadoops *JobTracker* initializes a new job for each input file and is responsible for all managing concerns of these jobs. If there are millions of input documents,

there are also millions of states to manage and numerous *map* steps will be executed. To solve this problem, the second approach created sub collections with sizes smaller than the chunk size of Hadoop. This idea solved the problems with many small files. Thus, distribution to XML databases became possible with good results. Using Hadoop's MapReduce framework to query the distributed sub collections is not beneficial, because of the overhead of MapReduce jobs execution. Therefore, querying was performed with an implemented REST prototype with good results.

Hadoop's MapReduce framework performed well for distribution of XML databases. The REST distribution approach performed not as fast as Hadoop, but with a similar performance. Hadoop is delivered as a comprehensive package and is not easy to extend and in order to tune to receive faster execution times. Furthermore, users are dependent on bugfixes and improvements from the software provider. In contrast to our alternative implementation like the REST distribution approach, it is almost always feasible to improve performance, e.g., to switch the network protocol whenever beneficial. Thus, all upcoming approaches will neither consider Hadoop's MapReduce framework for distribution nor the querying of XML data and base on new implemented prototypes.

5 Querying and Distribution of XML Data in an XML Database

5.1 MapReduce Framework vs Map and Reduce in an XML Database

There are several interesting ideas in the MapReduce framework introduced in [DG04] that serve as basic ideas for the approaches introduced next. These are mainly parallel evaluations of queries on different sites in *map* processes and combining their results in *combiners* and *reducers* as well as its speculative execution approach. MapReduce features like storing intermediate results on the secondary storages or shuffling between map and reduce are omitted or redefined. This is due to the fact that BaseX evaluates most queries in milliseconds or only few seconds. Furthermore, *large* XML data is distributed on manageable cluster sizes and not on XML clusters beyond 100 servers. Thus, failures will occur not as often as on really large clusters to justify persistence of each intermediate map results on the local hard disks.

Our BaseX' map and reduce approach is defined as follows: The *map* process is evaluated in parallel on all responsible data nodes in the cluster. The map functions are written by the user as XQuery file, which is distributed to the cluster. Within the XQuery file the user can access all documents and collections located on the data server. Therefore, we do not constrain the user to work on one particular database or collection. The *reduce* process collects all intermediate results from the map XQuery files. These results serve as input for the user defined reduce functions provided as XQuery files. Reduce functions can be executed as local combiner on each data server as a preprocessing step and globally to combine all intermediate results. The following table outlines the key differences between Hadoop MapReduce and our approach.

The next section introduces the distribution of XML collections within BaseX. It focuses

Characteristics	Hadoop MapReduce	Our Approach
Map and reduce function distribution via	Java Jar file.	XQuery file.
Parallel map execution	Yes.	Yes.
Replication	Yes.	No.
Storing intermediate results to hard disk	Yes.	No.
Shuffeling	Yes.	No.
Failure tolerant	Yes.	No.
Queries on	Records.	On database server.
Reduce function	Yes.	Yes.

Table 5.1: Intersections and differences of our map and reduce approach in comparison to the Hadoop MapReduce framework.

on identifying convenient data servers and building chunks of documents for distribution. BaseX' map and reduce architecture is introduced after the distribution section.

5.2 Distribution

This master thesis focuses on querying distributed semi-structured data, in particular distributed XML collections. We introduce one possible architecture for the distribution of an XML collection. Partitioning of large XML collections consisting of small XML documents is similar to horizontal fragmentation in relational world and thus, most relational techniques are applicable as well.

5.2.1 Architecture

As introduced in Chapter 1, a distribution architecture can be a centralized, decentralized or a hybrid approach. We decided to choose a centralized approach for XML collection distribution. The reasons for such a decision are that centralized approaches have less overhead for meta-data organization related to data server and database states. Furthermore, it is less complex to coordinate query routing to responsible data servers and collection of query results. Load balancing is managed with little expense as well. On the other hand, such a decision leads to a potential single point of failure or a possible bottleneck if many clients have to interact with the coordinator. To cope with this challenge our approach moves from a central architecture to a hybrid approach during this chapter, where all data nodes act as coordinator nodes. This architecture is depicted in Figure 5.1

As depicted in the Figure 5.1 a user *client* application contains an XML collection that must be distributed. The collection is transmitted to the *Coordinator*. The *Coordina-*

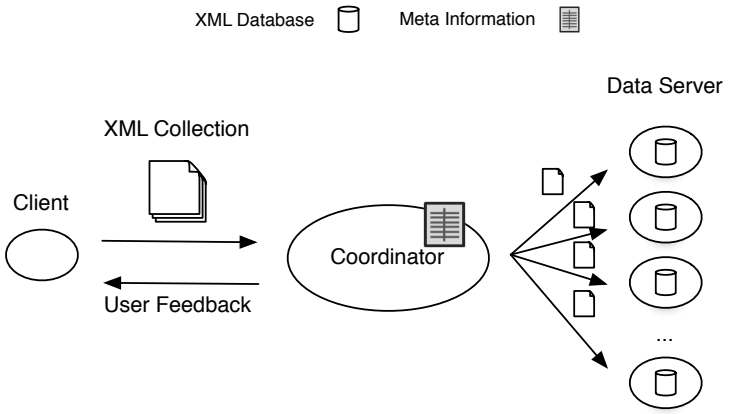


Figure 5.1: Distribution architecture.

tor knows all available *XML data servers* in the XML cluster and their states. The XML documents that are contained in the collection, are traversed and distributed by the coordinator server to the XML data servers defined by an algorithm. The different distribution algorithms are introduced below. During distribution, the coordinator sends the state of the distribution process to the user client application.

5.2.1.1 Distribution Algorithms

The first approach focuses on distribution in a round-robin manner. Each input document is directly distributed to an XML data server. The advantage of this algorithm is that the XML collection is distributed in a uniform way. Each data server contains the same amount of documents. Large XML collections are queried in parallel because all data servers contain one part of the collection. On the other hand, this algorithm also has disadvantages. First, if the collection is small, i.e., 1 MB, the collection is distributed like $\frac{collection}{\#documents}$, which is a huge overhead for querying a small XML collection at a first glance. Second, if all collections are uniformly distributed to all data servers, all collections become useless if one server is no loner available.

Algorithm 1 RoundRobinDistribution($collection$: Collection)

```
1  for doc in collection do
2     srv ←next(SERVERS)
3     distribute(doc, srv)
4  end for
```

The second approach solves the above situation by using additional meta information about collection size and the amount of contained documents and information about all data servers. It first checks the size and the amount of contained documents to detect the amount of potential data servers. Afterwards, it inspects the available free space amount of the XML data servers to obtain an approximate uniform storage distribution. The available free space amount of each data server is calculated periodically. One disadvantage of this algorithm is, i.e., that 95% of a collection is distributed to one server and the remaining 5% to the next server if the collection size is only few MBs larger than the defined threshold based on the server meta information.

Algorithm 2 AdvancedDistribution($collection$: Collection, $meta$: ServerMetaData)

```
1  if collection.SIZE < meta.RAMSIZE then
2    distribute(collection, getFreeServer())
3  else
4    servers ←getSortedServers()
5    subcolsize ← 0
6    srv ←next(servers)
7    for doc in collection do
8      if (subcolsize + doc.SIZE) < meta.RAMSIZE then
9        subcolsize ← subcolsize + doc.SIZE
10         distribute(doc, srv)
11       else
12         srv ←next(servers)
13         subcolsize ← doc.SIZE
14         distribute(doc, srv)
15       end if
16     end for
17   end if
18   meta.Update()
```

5.2.1.2 Partitioning and Packaging

It is beneficial to distribute a collection with additional meta information about server configurations and states. Furthermore, it should be considered to execute the *distribute()* function only once for one target data server to avoid unnecessary network communication in particular for opening and closing network connections. Therefore, we open for each server the network connection and send all selected documents in one run, by checking the size of each document and adding it to the defined server. Afterwards, the connection is closed. This improvement depends on the chosen network

protocol and yields small to large speed upgrades.

5.2.2 Prototype

The prototype we present uses BaseX[Grü10], a native XML database and XQuery processor, as backend for our XML collections on each data server. BaseX offers two network communication interfaces: a Java Client API, directly using sockets and a REST interface over HTTP. In the following, we consider the concrete implementation of BaseX' network protocols as base for our evaluation. We decided to use the Java programming language for our distribution and querying prototypes because of its cross-platform support, but the fundamental approach in this thesis is also applicable to nearly all other programming languages. The prototype uses the above introduced algorithms for distribution. Furthermore, it supports distribution directly via sockets or REST using HTTP[1]. Both approaches are described in the next sub sections and the performance is analyzed in the section Results. The class diagram shows the structure of the distribution implementation.

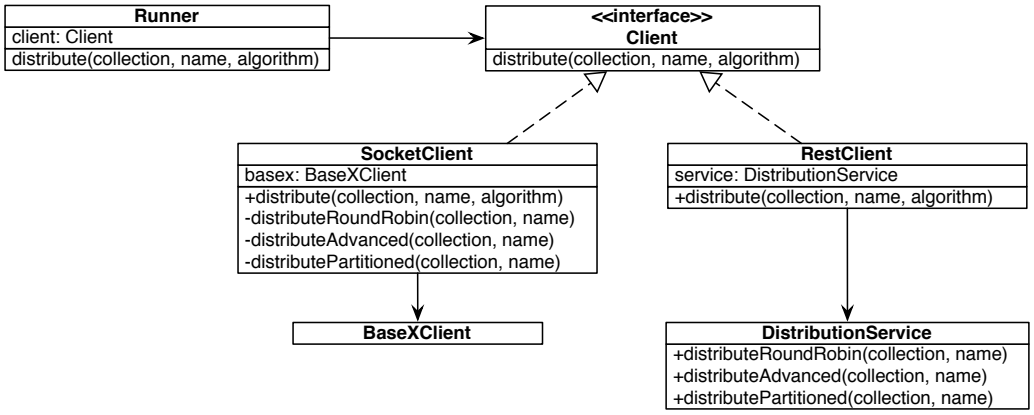


Figure 5.2: Distribution implementation.

5.2.2.1 BaseX REST API

In the class diagram, the *Runner* class decides, which *Client* implementation is used for the execution of our distribution process. In the REST case, the *RestClient* is used for

[1]We use in this thesis the terms sockets and REST over HTTP for the concrete protocol implementation used by BaseX.

distribution. This class implements the method *distribute(collection, name, algorithm)* and decides, based on the algorithm parameter, which distribution is executed. It delegates the execution to the *DistributionService* class, which implements the algorithms based on the HTTP protocol. The *distributeRoundRobin(...)* method distributes, as introduced above, each document by initiating an HTTP call to the responsible XML data server. We developed an improved version sending chunks of documents with only one connection to one particular data server; see section 5.2.2.3 for more information. BaseX accepts the PUT and POST requests, and adds documents to the new collection. The *distributeAdvanced(...)* method checks the size of the collection and compares it with the meta information corresponding to the data servers. If a collection is small enough, it is distributed to only one data server; otherwise it is split to few data servers as possible. The *distributePartitioned(...)* operation is a further optimization of the *distributeAdvanced(...)* algorithm. With *distributeAdvanced(...)*, we obtain the problem that if our collection is about 4.5 GB, but our main memory on each server has only 4 GB, our collection is distributed to two servers with sub collection sizes of 4 and 0.5 GB respectively, which should be optimized especially for queries that do not benefit much from indexes. Therefore, the *distributePartitioned(...)* algorithm first checks how many sub collection are needed, and partitions all sub collections with almost the same size. Our example then yields two sub collections with approximately 2.25 GB sizes distributed to two different servers.

5.2.2.2 BaseX Client API using Sockets

When the *Runner* class decides to use *SocketClient* as *Client* implementation, it calls the *distribute(collection, name, algorithm)* method as well. Within the *SocketClient* we choose the distribution method based on the algorithm parameter. *distributeRoundRobin(...)* distributes each document in round-robin manner to a data server. The socket version uses only one connection to a given data server and uses it for all documents corresponding to the server. BaseX clients send commands through the API and wait for the server response. The *distributeAdvanced(...)* method has the same functionality as in the REST version with the exception, that we use only one connection to the server for all corresponding documents. *distributePartitioned(...)* is implemented fundamentally equal to the REST approach. For the communication with the XML data servers and thus, to the BaseX servers we use a modified *BaseXClient* class version implemented by the BaseX team.
There are some differences between the REST and the Java Client API over sockets

within BaseX. It is possible by the Java Client API to add several documents to an existing collection over one socket connection. BaseX' REST interface is designed to accept only one document per HTTP request. Therefore, it is necessary to create a workaround, if you need to send more than one document with one HTTP request, which is introduced in the following sections. On the other hand, it is only possible with the REST interface to automatically wrap results to ensure a well-formed XML result.

5.2.2.3 Importance of Chunk Sizes

As stated before, distributing each XML document with one separate HTTP call is expensive and not useful. Consequently, we have to omit to use many requests. One possible solution is to use only one connection per server and to distribute all documents corresponding to a server as introduced in the algorithms *distributeAdvanced(...)* and *distributePartitioned(...)*. A challenge occurs, when an HTTP request fails because, e.g., a data server connection disappeared for some moments. If we tried to send a sub collection file with, for example 3 GB of content, we have to send it again from the beginning. This problem does not occur in the pure socket version because we use the *BaseXClient* implementation, a protocol implementation invented by the BaseX team and described in [Wei10]. The BaseX command *ADD* adds each document separately to the BaseX database or collection, since we call it for each document. Nevertheless, we hold only one socket connection. To overcome this problem, we use several chunk sizes to define the number of HTTP requests to one existing server. If we divide, e.g., our 3 GB example in three packages and set the chunk size to 1 GB, we thus need three HTTP calls. Therefore, if a network failure occurs during the third HTTP call, we only have to resend the third package.
In the next sub section we present our results of our introduced different algorithms with several data set sizes. Furthermore, we illustrate the results of our analysis using chunk sizes with the REST approach.

5.2.2.4 Bulk Import of whole XML Collections or Subcollections

Currently, BaseX servers do not support importing whole collections in one client call. This is due to the difficulties to extract file information from one stream required to store a document with a corresponding URI. Therefore, we have to add documents using the *ADD* command within the BaseX Java client or to execute an *ADD* operation with REST for each document within a collection for the naive approaches. To enable chunked

transmission of several documents a workaround was created. On the client side, we create a new *subcollection* root element and add *document* elements as its children. The document element contains a path attribute, which describes the document path for the needed ADD operation. As child of *document* we add the content of one XML document from the collection. Afterwards, we send the sub collection XML to the server. On the server side, we store this XML file in a temporary database and execute an XQuery file to create a new database, a collection of documents using the data from the temporary database.

5.2.3 Results of our Collection Distribution Approach

We use five workstations to simulate our distributed cluster. All of them have two Intel(R) Core(TM) i7 CPU 870 processors with 2.93 GHz and 8 GB of main memory. Four of the available workstations are used as data servers and one workstation is used as coordinator. We use the New York Times Article Archive[2] (NYT) as XML data set for distribution and querying. The NYT data set is a collection of small news article XML files, where a common XML file has a size of approximately 10 KB. The structure of all XML files in the collection is equal. An NYT XML article file with title *European Union Warns Google on Possible Violations of Privacy Law* is depicted in Figure 5.3. The structure of the XML file is regular and the tree has a level of six. The contents of the documents are meta information, like the publication year, as well as full-text articles. We decided to create several collection sizes, 43 MB, 110 MB and 1.2 GB to evaluate the performance differences between the REST and the sockets approach. Furthermore, we present the performance difference between performing REST requests for each document in a collection and approaches, which use chunk sizes.

We use BaseX version 7.0.2 as XML database on all data servers and use either the REST or the Java Client (sockets) approach for our distribution examples. We consider the distribution execution time of our collections for our comparison of both network APIs.

Figure 5.4 depicts the comparison of the round-robin approaches. The round-robin simple approach distributes all documents to all available BaseX servers. As expected the Java Client API performs better than the REST approach (a). This is due to the fact that

[2]It is available on the http://www.nytimes.com/ref/membercenter/nytarchive.html web site. This thesis is supported by the Graduiertenkolleg, Computer and Information Science, University of Konstanz with the focus on explorative analysis and visualization of large information spaces.

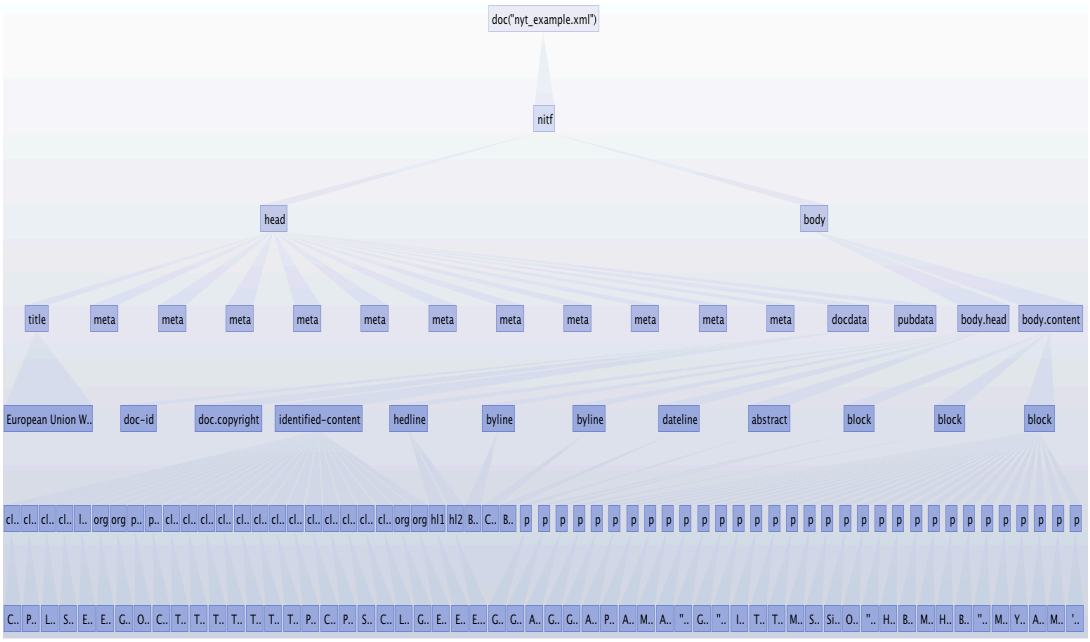
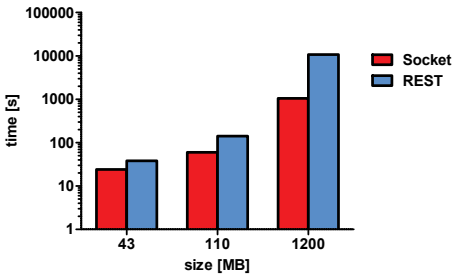


Figure 5.3: Treeview of the NYT example document.

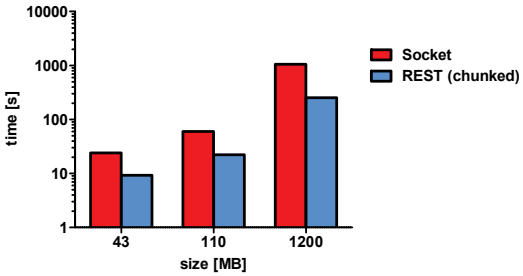
the initial REST distribution implementation uses one HTTP call for each document in contrast to the Java Client API, which uses one connection per server. Unfortunately, the REST API allows only to add new documents to existing collections by using one HTTP call each. If we consider the 1,200 MB NYT collection, we notice about 146,000 XML documents. The REST variant initializes 146,000 HTTP calls to distribute all documents to the available cluster. This overhead of initializing an HTTP connection is an evident drawback if there are many XML documents in our collection. Additionally, a second drawback of the REST approach is identified. In contrast to the Java Client API the REST variant performs an index update after each call. This implicates a lower distribution execution performance, if we increase the number of documents. The Java Client API performs index updates first after a bunch of update requests. We increased the Java Client API performance by setting the client property *AUTOFLUSH* to *false* and enforce the flushing using the *FLUSH* property at the end of our distribution process. Thus, the database buffers will not be flushed to disk after few updates.

The main reason for allowing only one document per HTTP request within the REST approach is that we need additional information about the contained documents. The



[s]	43 MB	110 MB	1200 MB
Socket	24.04	59.82	1048.82
REST	38.10	141.26	10698.96

(a) Round-robin simple approaches.



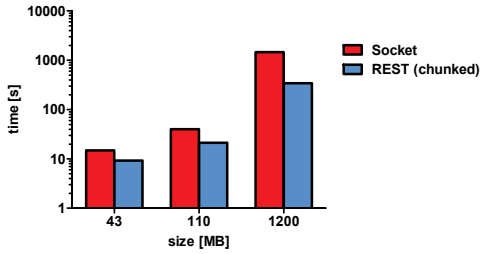
[s]	43 MB	110 MB	1200 MB
Socket	24.04	59.82	1048.82
REST (chunked)	9.28	22.13	253.02

(b) Round-robin simple and chunked approaches.

Figure 5.4: Distribution using the round-robin approaches.

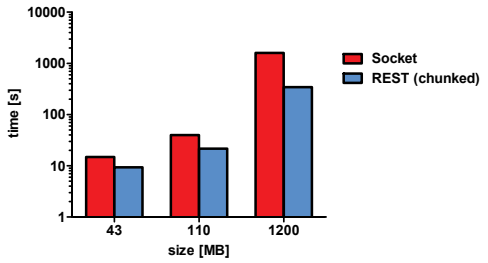
REST API adds a new document with a URI name identified by the request URL of the PUT request. It is not possible to add two or more documents within one request, because we cannot map it to one URL. Therefore, we decided to create a bulk-like workaround for the REST approach to use also only one server connection as the Java Client API. We wrap our documents with a *subcollection* root element node. Then, we wrap each document with a *document* element node consisting of the attribute *path* containing the document URI and the document itself as element content. Following, we insert the *document* node as child to the *subcollection* element. We send our sub collection to the corresponding server. On server side, we create a temporary database for our sub collection and start a refactoring operation to create a new database with the original state, a collection of XML documents. The results of this REST workaround are depicted in Figure 5.4 (b). In this situation, the REST approach outperforms the Java Client API, although using also only one connection per server, which is due to the BaseX protocol implementation, discussed later.

Figure 5.5 depicts the results of the advanced and the partitioned distribution algo-



[s]	43 MB	110 MB	1200 MB
Socket	14.84	39.89	1459.60
REST (chunked)	9.21	21.21	341.37

(a) Advanced simple and chunked approaches.



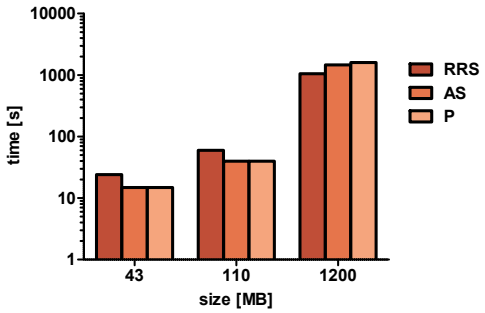
[s]	43 MB	110 MB	1200 MB
Socket	14.83	39.84	1602.46
REST (chunked)	9.32	21.67	342.64

(b) Partitioned simple and chunked approaches.

Figure 5.5: Distribution using the advanced and partitioned approaches.

rithms. Both algorithms distribute the XML collection using server meta information like the available server RAM size. If a collection is smaller than a computed threshold, which considers the server RAM memory, e.g., operating system RAM reservation, it is distributed to only one data server. Otherwise, it is distributed to few servers as possible. The *advanced* does not consider uniform distribution. The *partioned* algorithm distributes the collection uniformly to few servers as possible. In our test collection sizes both algorithms work equal, because the collections are smaller than this defined threshold and are stored on one data server. The distribution execution times are consequently almost the same between (a) and (b). For both approaches we used the REST chunked version, consisting of the created sub collection, as mentioned above. Again the REST approach outperforms the Java Client API.

One interesting consideration was that we also have a performance loss when we compare, for example the BaseX socket implementation for round-robin (RRS), advanced (AS) and partitioned (P) approaches. RRS is slower for smaller collection sizes, because



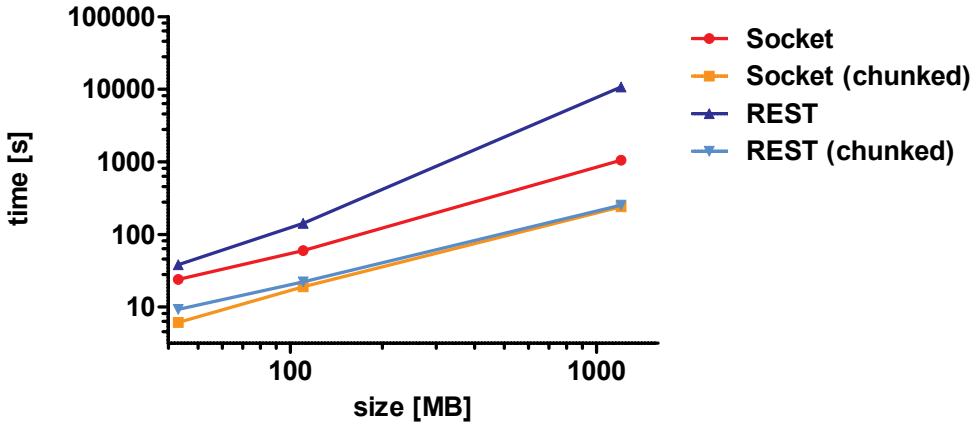
[s]	43 MB	110 MB	1200 MB
RRS	24.04	59.82	1048.82
AS	14.84	39.89	1459.60
P	14.83	39.84	1602.46

Figure 5.6: Comparison of socket algorithms.

we switch the server for each document. AS and P were much slower with larger collection sizes, as presented in Figure 5.6. This is due to the fact that *AUTOFLUSH* is set by default to true and index updates become more costly. We are able to solve this problem by setting this property to false and flush manually. Another drawback, compared to the REST chunked version is that although we use only one connection for both network APIs, the Java Client API sends each document with a new request through the socket. The Java Client API sends first a command, e.g., *ADD*, and afterwards the document to the server. Furthermore the client API waits for a server response after each sent document, which contains information about the command execution. To overcome this drawback, we created the same workaround for the Java Client API as for the REST approach and introduced the *subcollection* solution. A final comparison of the simple and the chunked RRS versions is depicted in Figure 5.7. As in the initial situation, the Java Client API performs best again. In contrast to the initial situation, the modified Java Client and the REST APIs are more close together.

5.2.4 Challenges

If we do not consider the workaround solutions using the *subcollection* wrapping, the Java Client API performs best. We are able to improve the performance by disabling the *AUTOFLUSH* property on server side. The first challenge we retrieve is that we must flush manually and define a threshold deciding when a flush has to occur. If we perform flushing at a late point in time, we have a high probability of data loss. When we execute a flush, e.g., after 10 updates, the performance will go down. If we use the workaround solution, we must consider two further challenges: first, if we build one document of a bunch of several XML documents and send it through the network, we



[s]	43 MB	110 MB	1200 MB
Socket	24.04	59.82	1048.82
Socket (chunked)	6.13	18.91	240.67
REST	38.10	141.26	10698.96
REST (chunked)	9.28	22.13	253.02

Figure 5.7: Comparison of round-robin algorithms.

have to define the appropriate chunk size. We are able to send it completely in one run, but the disadvantage is that if a network failure occurs we lose all documents and have to resend everything. Thus, we define smaller chunk sizes for larger collections than sending the whole collection at once. We defined the chunk size of 64 MB as base for our distribution. Another challenge is that if we distribute large collection sizes, the refactoring on server side becomes very costly due to the recreation of a complete database to reconstruct the original structure.

5.2.5 Conclusion

We introduced three different distribution algorithms and evaluated them using the BaseX Java Client API over sockets and BaseX' REST approach over HTTP. Although, the HTTP network protocol is able to have a similar performance as to communicate directly over sockets, it is necessary to build workarounds to allow this using the BaseX implementations. The RRS distribution performed best for large data sets and the BaseX Java

Client API yield the best performance for the current implementation. We use therefore the BaseX Java Client API also for our query distribution in the upcoming sections.

5.3 Querying

After successful distribution of an XML collection, we focus on querying the distributed XML data. We concentrate our analysis within this thesis on queries without updating operations due to time restriction of this master thesis. Querying XML data is not trivial, because of the dynamic nature of XQuery, i.e., it is possible to open and query several documents and collections out of an XQuery expression. Consequently, we decided to be as nonrestrictive as possible. It is difficult to limit user XQuery expressions only to one collection or document, because you have to parse the query in advance and identify all documents. This becomes more complex if users decide dynamically, which document they want to use. Furthermore, it is disadvantageous to cut the mightiness of XQuery. The user has to be able to write a query as usual, but in a distributed context. Furthermore, the kind of distribution affects the query performance depending on the query and its possibility to benefit from index structures. The next sub sections describe the used query architecture, the implemented prototype, and the results that serve as base for the discussion.

5.3.1 Architecture

We choose a centralized approach for our basic query architecture similar to our distribution approach because of the simplification of book keeping of data servers meta data. The user communicates only with the coordinator server and does not know about the data servers. This independence allows us to add and remove servers without user dependencies. One objective of this thesis is to allow an XQuery developer to cope with large data within an acceptable timeframe and with as few restrictions as possible. A user is able to write one query, which will be distributed to the responsible data servers and a second query, which aggregates the results of the distributed query. This architecture is depicted in Figure 5.8. The *Coordinator* is responsible for forwarding queries in parallel to the data servers, which contain the necessary databases. Afterwards, it is responsible to aggregate the results from the data servers using the second user query. After aggregation, it sends the results to the requesting user. The three main processes

in this figure are *Coordinator, Map Process* and *Reduce Process*, which are described in detail now.

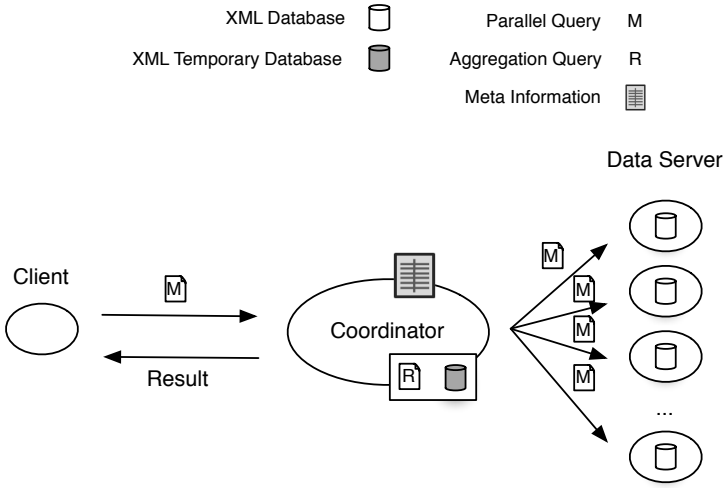


Figure 5.8: Querying basic architecture.

5.3.1.1 Coordination and Book Keeping

The *Coordinator* is the central unit in the current distributed query architecture. It accepts requests from the user and distributes the user queries to the responsible data servers. Furthermore, it is dedicated to perform aggregation of the results. Before the distribution of queries begins, the coordination process consults additional meta information. Several information is stored on the coordinator machine during distribution of the collections:

- Hardware specifications of the available data servers, e.g., hard disk sizes, RAM or CPU.
- Mapping of documents and collections, the distributed collections and the responsible servers are stored in a mapping file.
- Sizes of distributed documents, which are used to hold statistics about data servers workload.

This information is used to distribute the query only to data servers, which hold the requested collection. In the case where a query contains documents that are not identified at runtime, the query is executed on all data servers. The *map process*, representing the

distributed parallel querying, is completed by a user defined *reduce process* combining all intermediate results. Both processes are discussed now in detail.

5.3.1.2 Map Process

The *Map Process* illustrates the idea of distribution of a query, which is evaluated on all data servers. That means that this type of query returns intermediate results, which are aggregated in a second step to receive the complete query results. The XQuery file is distributed in parallel to all available data servers. Afterwards, the local XML database executes the query on each data server and sends the result to the *Coordinator*. The main advantage is that a large collection is split into several pieces, which are queried in parallel. Querying in parallel allows to evaluate expressions on collections, which do not fit on one machine. Furthermore, it decreases the query execution time for queries that do not benefit from existing index structures.

5.3.1.3 Reduce Process

The *Reduce Process* is responsible for aggregating the intermediate results from the map processes and transmitting it to the requesting user. Basically, the user has two possibilities to perform the reduce process: First, to omit the reduce function at all, then the *Coordinator* concatenates the results as they come in and forwards it to the user machine or second, defining a reducer query. If the user decides to perform the reduce process by using an own query, e.g., to perform sorting of results, the design of the map results has some restrictions. They must be well-formed XML, because we need a temporary database on the reducer machine to evaluate the reducer query. Thus, the map process results are stored in a temporary database and the reducer query is executed within the context of this database instance. The reducer query results are then transmitted to the user machine. A map and reduce word count example query looks like:

```
Complete query:
Give me the top 10 articles with the most 'the' occurrences

Map query:
Calculate the top 10 articles with the most 'the' occurrences
from one database server
```

```
Reduce query:
Calculate the top 10 articles with the most 'the' occurrences
from all map results
```

This query has to deliver the top 10 articles with the most 'the' occurrences out of a distributed articles collection. The user has to define the part, which is evaluated in parallel as map query and the reduce query aggregating the map intermediate results. This concrete example is depicted in Figure 5.9.

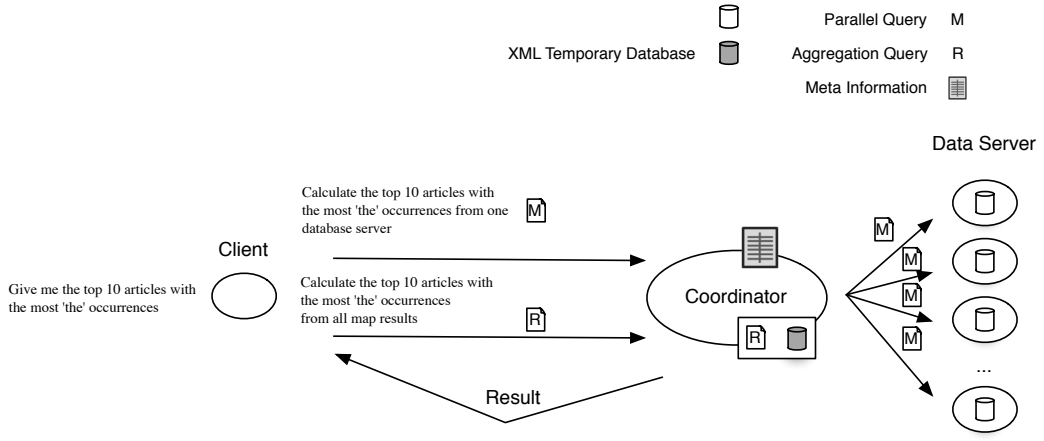


Figure 5.9: Top 10 articles example.

5.3.2 Prototype

The prototype uses BaseX as XML database backend on each data server. Furthermore, it uses BaseX also for evaluation of the reducer query within a temporary database. It performs distributed querying using also the Java Client API, directly over a socket protocol, and the REST approach using HTTP as network protocol. The querying prototype is again implemented in Java due to its cross-platform support. We used the same distribution algorithms as introduced in 5.2 as base for distribution of our test collections. The prototype listens for queries defined by a client. The client sends a map query and the optional reduce query as XQuery files. The coordinator then distributes the query to the responsible data servers either by the Java Client or the REST API. The local XML database executes the map query and sends the results back to the coordinator. If the client specified a reducer query, a temporary database is created on the *Coordinator* machine. The intermediate results are stored in the main memory database and afterwards,

the reducer query is executed. The complete results are delivered to the client. The class diagram is depicted in Figure 5.10.

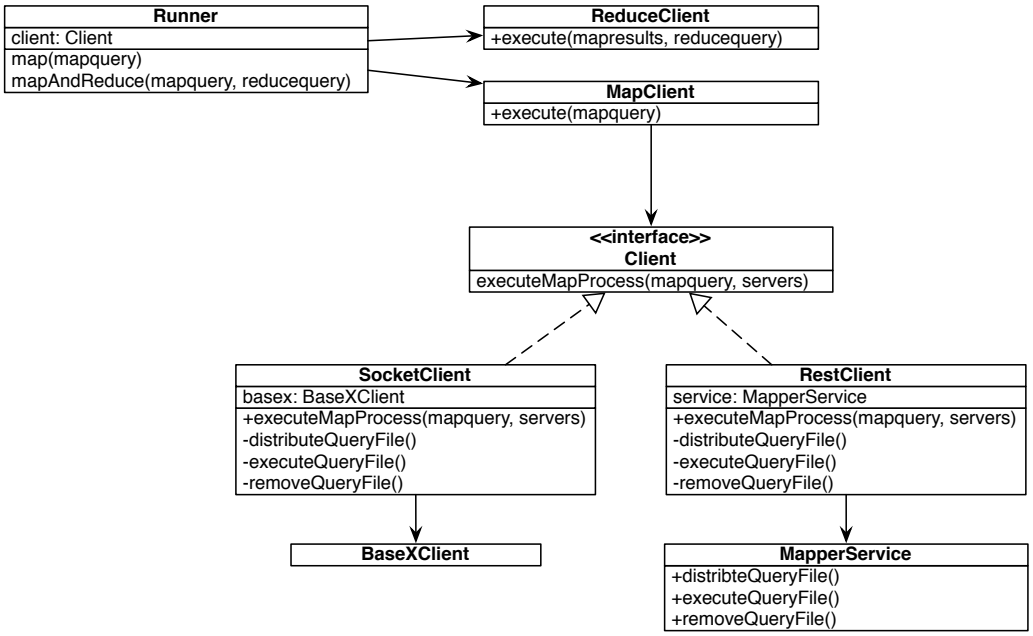


Figure 5.10: Query class architecture.

5.3.3 Results

Again, we use five workstations to simulate our distributed cluster. All of them have two Intel(R) Core(TM) i7 CPU 870 processors with 2.93 GHz and 8 GB of main memory. Four of the available workstations are used as data servers and one workstation is used as coordinator. We use also some XML collections of the NYT news articles collection with sizes of 43 MB, 110 MB, 1,200MB, 10,000MB and 16,000 MB as base for our querying. All collections are distributed using the round-robin (RRS), advanced (AS) and partitioned (P) algorithms to evaluate if partition sizes and uniform distribution play a role for query execution. We used ten runs for all defined queries. The prototype implements both BaseX API's for querying of the distributed data. The user is able to decide, which network protocol has to be used by the prototype. This is done by specifying a parameter when starting the querying process. The results generated and depicted in this chapter use the Java Client API. This thesis focuses on query execution time as characteristic variable for the measurements and thus, all results depict the execution

time, which is build of query execution time and transmitting of the query to the data server and the query results to the client.

We defined four classes of test queries:

- Queries using no further reduce step and do not benefit from an index (Q1, Q2, Q3)
- Queries using no further reduce step, but benefit from index accesses (Q1', Q2', Q3')
- Queries using a reduce function and benefit from an index access (Q4 and R4)
- Queries using a reduce function and using an index (Q5 and R5), but need a lot of index accesses.

The following map queries are used for querying all test collections:

```
Q1:
for $d in collection('nyt')
where $d/nitf/head/title/text()=
      'NASA Optimistic on Space Station Repair'
return $d
```

```
Q1':
collection('nyt')/nitf[descendant::text()=
  'NASA Optimistic on Space Station Repair' ]
```

```
Q2:
for $d in collection('nyt')
where $d/nitf/head/pubdata/@date.publication='20070619T000000'
return $d/nitf/head/title
```

```
Q2':
for $d in collection('nyt')[descendant::node()/@date.pupblication=
  '20070619T000000' ]
return $d/nitf/head/title
```

```
Q3:
declare function local:countThe($nitf as node()*) {
let $texts := $nitf/descendant::text()
for $t in $texts
for $token in tokenize($t,"\s+")
where $token="The" or $token="the"
return $token
};
```

```
for $d in collection('nyt')
where $d/nitf/head/pubdata/@date.publication='19870514T000000'
return <nitf><title>{$d/nitf/head/title/text()}</title>
       <thes>{count(local:countThe($d/nitf))}</thes></nitf>
```

```
Q3':
declare function local:countThe($nitf as node()*) {
let $texts := $nitf/descendant::text()
for $t in $texts
for $token in tokenize($t,"\s+")
where $token="The" or $token="the"
return $token
};

for $d in collection('nyt')[descendant::node()/@date.pupblication=
  '19870514T000000' ]
return <nitf><title>{$d/nitf/head/title/text()}</title>
       <thes>{count(local:countThe($d/nitf))}</thes></nitf>
```

Query Q1 is responsible to find a special news article containing a defined text node. Query Q2 returns all titles of news articles, which are published at a given date. Query Q3 counts special words in an article and returns the sum of the word and the title of the article. Q4 and Q5 are queries that search for articles, which contain specified words in the full-text of the article. Q1 and Q2 are queries, which are executed during the map process. The results are returned directly to the requesting client and a special reducer is not called. Q3 - Q5 are also executed during the map process, but user defined reducer queries are performed on the coordinator node. The reducer queries are defined below. R3 is responsible to aggregate the results from Q3 and to sum up all 'the' words from all articles. R4 and R5 are the reducer functions for Q4 and Q5 and their content is the same. They sort the news article titles using their publication date, which are received from the map functions.

```
Q4:
for $d in collection('nyt')/nitf[descendant::text()
   contains text 'Google' ftand 'Microsoft']
return <nitf><title>{$d/head/title/text()}</title>
       <date>{$d/head/pubdata/@date.publication}</date></nitf>
```

```
Q5:
for $d in collection('nyt')/nitf[descendant::text()
   contains text 'Google' ftand 'the']
return <nitf><title>{$d/head/title/text()}</title>
```

```
       <date>{$d/head/pubdata/@date.publication}</date></nitf>

R3:
let $res:=/descendant::nitf
let $sum:=sum($res/thes)
return <complete-result><all-the>{$sum}</all-the>{for $nitf in $res
return <nitf><title>{$nitf/title/text()}</title>
       <the-count>{$nitf/thes/text()}</the-count></nitf>}
       </complete-result>

R4:
for $n in /descendant::nitf
order by $n/date/@date.publication descending
return <article><title>{$n/descendant::title/text()}</title>
       <date>{$n/descendant::node()/@date.publication}</date>
       </article>

R5:
for $n in /descendant::nitf
order by $n/date/@date.publication descending
return <article><title>{$n/descendant::title/text()}</title>
       <date>{$n/descendant::node()/@date.publication}</date>
       </article>
```

The following table outlines the queries and whether they use existing index structures. The below results use these queries as map and reduce functions. Q1' - Q3' are redefined queries, which benefit from existing index structures, but return the same results as Q1-Q3.

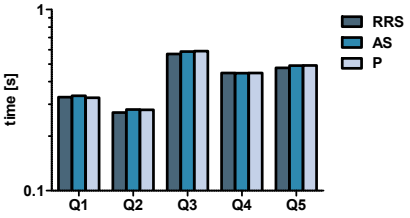
Query	Applying Index	Notes
Q1	No.	-
Q2	No.	-
Q3	No.	-
Q4	Yes.	-
Q5	Yes.	-
Q1'	Yes.	Reformulated Q1 to benefit from a text index.
Q2'	Yes.	Reformulated Q2 to benefit from an attribute index.
Q3'	Yes.	Reformulated Q3 to benefit from an attribute index.

Table 5.2: Queries using index structures.

Figure 5.11 depicts the map and reduce query results on different collection sizes that were distributed using the defined algorithms RRS, AS and P. RRS distributes all documents uniformly to all machines. AS considers the RAM size minus the operating system requirements of the data node and P distributes the collection using also RAM size of the data node and partitions the collection parts uniformly. For collection sizes of 43

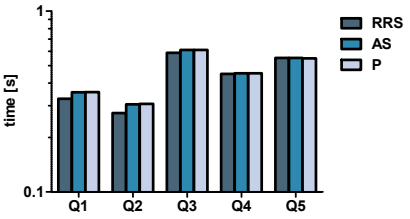
MB, 110 MB and 1,200 MB, AS and P work the same way because the collection is not partitioned and distributed only to one machine.
One interesting thing is that the RRS distribution algorithm is the best base for query execution for all tested collection sizes. AS and P perform almost the same for the collection sizes smaller than the RAM partitioning parameter (Figure 5.11 (a)-(c)).
P outperforms AS on the 10,000 MB collection due to the bad partitioning strategy of AS, which distributes about 7,000 MB to one data server and the other 3,000 MB to another. P partitions equally and distributes 5,000 MB each.
Figure 5.11 (e) shows similar performance results for queries on collections distributed using AS and P algorithms because the complete collection is stored on two data nodes, when considering RAM as partitioning parameter. Nevertheless, RRS performs best.
The second important issue is that the shortest distributed query evaluation costs about 300 ms, which is due to the network communication between *Coordinator* and the data nodes and the query evaluation time. We analyse this factor below in detail.

When we compare distributed query evaluation to local BaseX evaluation, we realize that performance is dependent on the defined XQuery expression. Figure 5.12 depicts the comparison of the test queries using the map and reduce processes. We consider only collection sizes of 10,000 MB and 16,000 MB for this evaluation because querying of distributed smaller collections of only few MBs is not competitive to local query execution. Q1 to Q3 do not benefit from an index, which is the reason for the bad performance. Q4 uses an index and performs better than the distributed alternative on both collection sizes. When the query complexity is increased or more index accesses are needed within one query, distribution becomes more attractive and outperforms local query execution. Furthermore, if queries are not able to benefit from an existing index or range queries are a more common use case, distribution is a good option.



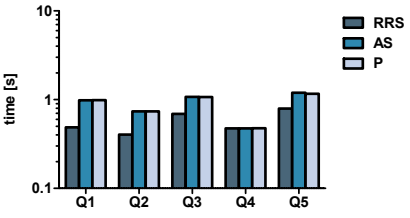
[s]	Q1	Q2	Q3	Q4	Q5
RRS	0.33	0.27	0.57	0.45	0.48
AS	0.33	0.28	0.59	0.45	0.49
P	0.33	0.28	0.59	0.45	0.49

(a) 43 MB collection.



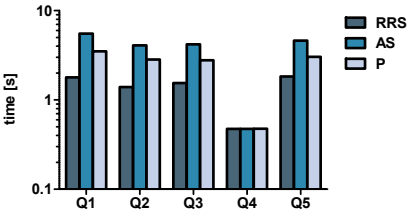
[s]	Q1	Q2	Q3	Q4	Q5
RRS	0.33	0.27	0.59	0.45	0.55
AS	0.36	0.31	0.61	0.45	0.55
P	0.36	0.31	0.61	0.45	0.55

(b) 110 MB collection.



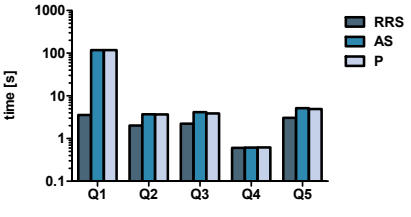
[s]	Q1	Q2	Q3	Q4	Q5
RRS	0.49	0.40	0.69	0.47	0.79
AS	0.98	0.74	1.07	0.47	1.19
P	0.99	0.74	1.07	0.48	1.16

(c) 1,200 MB collection.



[s]	Q1	Q2	Q3	Q4	Q5
RRS	1.79	1.40	1.55	0.47	1.83
AS	5.53	4.08	4.19	0.47	4.62
P	3.51	2.83	2.78	0.48	3.04

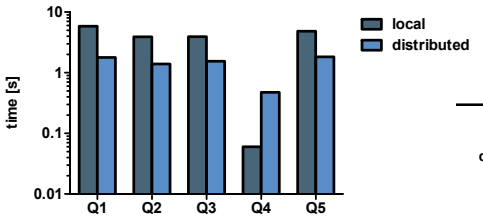
(d) 10,000 MB collection.



[s]	Q1	Q2	Q3	Q4	Q5
RRS	3.56	2.02	2.23	0.61	3.07
AS	118.34	3.69	4.14	0.62	5.13
P	118.34	3.67	3.87	0.62	4.92

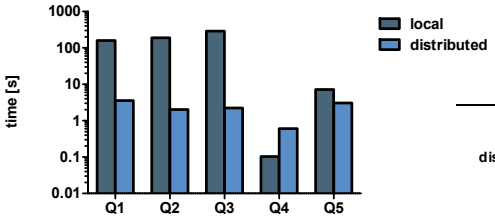
(e) 16,000 MB collection.

Figure 5.11: Queries Q1 - Q5 on different distributed cluster sizes.



[s]	Q1	Q2	Q3	Q4	Q5
local	5.83	3.90	3.92	0.06	4.83
distributed	1.79	1.40	1.55	0.47	1.83

(a) 10,000 MB collection.



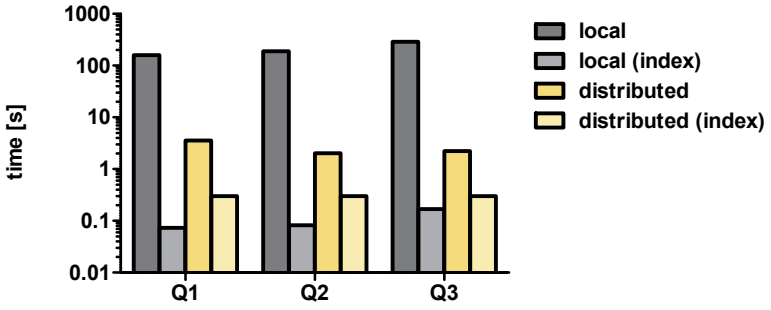
[s]	Q1	Q2	Q3	Q4	Q5
local	159.81	188.75	288.27	0.10	7.17
distributed	3.56	2.02	2.23	0.61	3.07

(b) 16,000 MB collection.

Figure 5.12: Queries Q1 - Q5 on different local and distributed cluster sizes.

Figure 5.13 depicts the results of the reformulated Q1 to Q3 queries that benefit from index structures. As expected, using indexes increases query performance. In this case local query execution outperforms distributed query execution, which also uses the reformulated queries in contrast to the query comparison that do not use index structures. Furthermore, this figure depicts the minimum measured distributed query time of about 300 ms for all three queries that benefit from indexes.

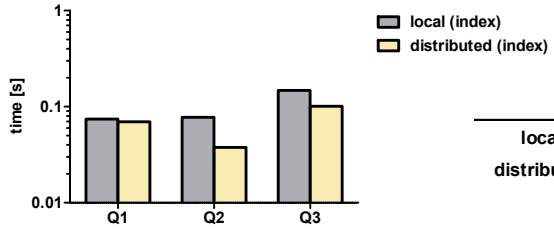
As in Figure 5.13 depicted the 300 ms boundary is very slow. We therefore investigated the least possible distributed query execution time because the network is fast and a ping of the data server lasts only 1 - 2 ms and thus the network is not the expected boundary. In the previous approach, we tested each distributed query execution in a new Java process. Furthermore, we used the Hadoop MapReduce idea to first distribute the query to the data nodes and to store them in a database. Afterwards, we executed the query using a second network call. Additionally, we printed the query results to the console, which is also not cheap. The new evaluation results, depicted in Figure 5.14, present the best performance results achieved with the queries Q1' to Q3' using only the index supported variants. The distributed executed queries outperform the local executed queries. We reused the JVM process, sent the query directly and wrote



[s]	Q1	Q2	Q3
local	159.81	188.75	288.27
local (index)	0.07	0.08	0.17
distributed	3.56	2.02	2.23
distributed (index)	0.30	0.30	0.30

Figure 5.13: Q1-Q3 applying text and attribute index.

the results in a file. The least achieved empty query execution time in our distributed evironment was 6.5 ms.

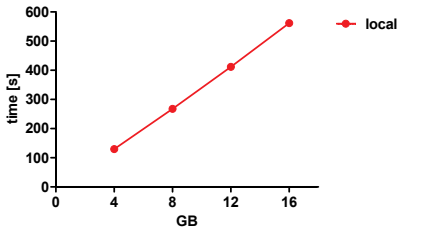


[s]	Q1	Q2	Q3
local (index)	0.074	0.077	0.148
distributed (index)	0.070	0.038	0.101

Figure 5.14: Q1'-Q3' applying text and attribute index using the tuned execution approach.

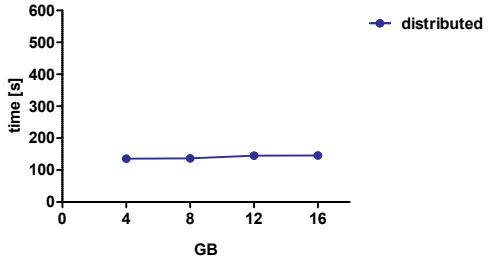
5.3.4 Scalability with the Top 10 Example

We developed a query, the *Top 10 Example*, which retrieves the top 10 articles with the most 'the' word occurrences within the NYT XML collection[3]. We used four NYT data sizes to perform local query execution in comparison to the distributed approach using one to four data nodes. We again performed ten runs and used the average for our comparisons. Figure 5.15 depicts our results. With increasing of the data and adding new data nodes as new resources, the execution time remains constant.



[s]	4 GB	8 GB	12 GB	16 GB
local	129.80	267.96	411.83	561.62

(a) Local Top 10 execution on different sizes.



[s]	4 GB 1 server	8 GB 2 servers	12 GB 3 servers	16 GB 4 servers
distributed	135.89	136.76	145.39	145.64

(b) Distributed Top 10 execution on different sizes and several servers.

Figure 5.15: Scalability of the distributed XQuery approach.

5.3.5 Challenges

The RRS distribution served as best base for our query evaluation. All available data nodes are involved into data distribution and therefore, also into query execution. There are several things that must be considered. First, all nodes are involved for all query requests, which is a major drawback, when many clients send many queries to different collections. This is due to the fact that the BaseX instance must handle all requests and the buffer manager often has to reorganize all available buffers. If we would consider updates as well as read-only queries, BaseX will lock each database server for a given update query, which slows down the complete query performance for other clients even

[3]You can find the defined queries and the example workflow in Chapter 6

if they tried not to access the same collection. Another problem arises when one data server fails, e.g., due to a hardware failure. All available databases lose data. Replication would be a solution to avoid such a situation. It would be interesting to analyse the distribution and querying of one collection size with different cluster sizes to find the threshold for managing parallel data nodes to measure the maximum parallel overhead join, see [Amd67].

Furthermore, we considered only three distribution strategies. We did not mind that *hot* collections existed, which are queried much more often. These collections could be redistributed or rebalanced to offer a better query performance. Another question is what happens with existing collections, if we would add an additional data node. It should also be considered whether we should have to redistribute all existing collections or not when using RRS.

In addition, the distribution and querying is dependent of the user queries. Simple queries, which can be optimized, e.g., by using an index structure, need another distribution strategy than queries that are difficult to optimize.

The introduced approach executes queries on the data nodes. If a query contains expressions that need data from other databases, it only accesses the local database server instance. As a consequence, when using RRS the sub queries are not routed as new map processes, which is an essential weakness. Consequently, there is a lot of place for improvement in this area. One possible solution of this problem is introduced in the next chapter.

We considered only an XML collection size of 16,000 MB. There are also a lot of collection sizes, which are beyond of the test data set. It would be interesting to investigate querying on several TB or even PB.

Currently, the reducer functions are executed in the context of a main memory BaseX database on the coordinator node, which is a restriction due to the available main memory size.

Further challenges and ideas are discussed in the chapter Future Work.

5.3.6 Conclusion

We investigated the three distribution algorithms RRS, AS, and P as base for our querying. Again RRS was the best starting position for our test query set. We developed four test query classes to leverage query performance on a distributed 16 GB XML collection. During this evaluation, we minimized almost all ideas from the initial Hadoop MapReduce approach to the core idea of the two functions *map* and *reduce*, which yielded great performance results. The most impressive results are generated for queries, which are not able to benefit from an index structure. Additionally, the results are dependent on the server properties like the available RAM size and whether index structures are able to remain in main memory. The performance results showed that many queries benefit from distribution of large collections and the overhead for small collections is not as big as expected. At the beginning of this chapter, we analyzed some ideas of distribution strategies that serve as the base for the query evaluations. The result is that the distribution strategy has a considerable influence on the query performance. Moreover, we considered only querying of collections of documents and not a fragmented and large XML tree, which is quite complex due to the right choice of fragmentation rules.

6 EXPath Packaging and Example Workflow using BaseX

The goal of this thesis was to investigate querying of distributed data using XQuery expressions and the best approach is to execute the distribution of queries directly out of XQuery. Currently, XQuery itself does not support distributed parallel querying. Although, several approaches already exist to perform parallel querying out of XQuery, most of these ideas extend the XQuery language specification to allow a user to perform distributed calls [FJM+07a, FJM+07b]. These approaches have the drawback that each XQuery processor must implement such an arbitrary XQuery extension. Aside from these methods, another idea is to integrate such a query distribution function directly into the XQuery processor, but the problem here is that each other XQuery processor has to implement an own distribution function. There is yet another attempt to cope with this problem EXPath [Con12a]. Here EXPath provides specifications to enable features, which are not part of XPath and XQuery and are used within several query processors. A useful specification is the *EXPath Packaging System*. It allows packaging any set of XML core technology files in an archive, which is integrated into a query processor [Con12b]. BaseX supports the EXPath packaging features and therefore, we decided to use the packaging idea for our query distribution. The advantage of this idea is that our approach does not need to extend the XQuery specification and all EXPath supporters benefit from it. In the next sub section, we give an introduction into EXPath.

6.1 EXPath Packaging System within BaseX

BaseX offers the possibility to add EXPath packages, which extend the BaseX functionality. Currently, BaseX supports sending HTTP requests to other resources via the EXPath HTTP specification, as described in [Bas12a]. The problem with this approach is that it is not possible to execute HTTP requests in parallel. It is only possible to execute HTTP requests in a loop, iterating through the XQuery item sequence. Thus, a new EXPath

package is created to support parallel distributed query execution. BaseX accepts packages, which are *.xar* archives that contain one or more extension libraries. Such libraries can be either XQuery libraries or Java libraries. Since we must execute parallel query execution through external Java code, we add a Java *jar* file to the xar archive. The structure of the xar archive is defined by the EXPath specification, [Con12b]. In general it consists of a descriptor XML file, which contains meta information about the package and its dependencies, e.g., the jar file. Furthermore, a *wrapper* XQuery file is contained using the *BaseX Java Bindings* to call the implemented Java classes within the jar package. This package is installed or deleted through the BaseX' commands REPO INSTALL or REPO DELETE. Afterwards, the user is able to use the installed module by defining its module namespace in the query scripts. A detailed description is available on the BaseX packaging documentation web site [Bas12b].

6.2 Distributed Querying

Due to the better performance results of the Java Client API compared to the BaseX REST API introduced in Chapter 5.2, we decided to distribute the queries using BaseX' Java Client API. The user is able to define the query, which has to be distributed and enter it as function parameter to the packaged and installed module. This module distributes the defined query in parallel to all defined BaseX servers. The queries are evaluated on each BaseX server and results are sent back to the initiator BaseX application. This approach will now be described in detail using some example workflows. All examples are based on the NYT 16 GB data set, which is distributed to four BaseX servers using the round-robin approach. The workflow of our example is depcited in Figure 6.1.

The XQuery file contains a query, which is defined as String value. The query is delegated to the wrapper XQuery file, which is responsible to instantiate the corresponding Java class. The Java class contains a method to distribute the query in parallel, which is invoked by the wrapper XQuery file. After parallel distribution and evaluation of the query (map process), the intermediate results are transformed to XQuery data types and returned to the wrapper file. The wrapper file then delivers the complete results to the user defined query. The user query is able to further use the distributed query results for other queries, i.e., aggregation of the results (reduce process).

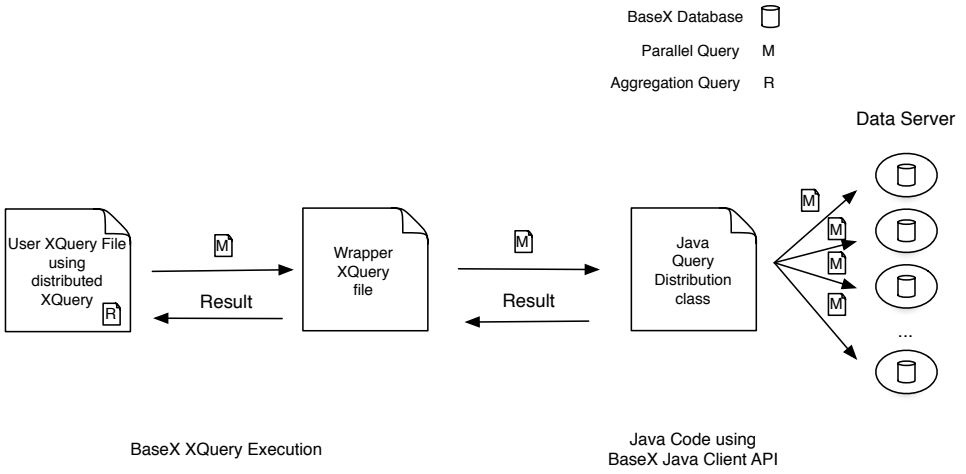


Figure 6.1: EXPath workflow example using BaseX.

6.2.1 Map execution

We defined two queries for our examples that represent user queries: DQ1 simply delivers all documents, which contain the keywords *Microsoft* and *Google* in the full-text of the news articles (as in Q4 in the previous chapter), and DQ2, which returns a *Top 10* list of documents with the most 'the' occurrences. DQ1 peforms only the map process, because the distributed query results are printed directly as complete results and no further reduce step is performed. The DQ1 query is defined as follows:

```
(: Import of necessary module namespace :)
import module namespace d="http://basex.org/modules/distribute";

(: List of BaseX servers for the distributed evaluation :)
let $urls:=('server1.example.com:20000',
'server2.example.com:20000',
'server3.example.com:20000',
'server4.example.com:20000')

(: Definition of distributed query :)
let $distributedquery:="<dq-result>{
for $d in collection('nyt')/nitf[descendant::text()
   contains text 'Google' ftand 'Microsoft']
return $d
}
</dq-result>"
```

```
(: Execution and visualization of distributed query results :)
let $dqresult:=d:query($mapquery,$urls)
return $dqresult
```

The important items in the DQ1 query are declaring the namespace of the distribution module, the definition of BaseX servers, which have to be queried and the distributed query definition. The *d:query($mapquery,$urls)* call then executes the defined XQuery function in the *wrapper* XQuery file:

```
(: The module namespace used in the other queries. :)
module namespace d="http://basex.org/modules/distribute";

(: The java namespace defines the Java class
which will be responsible for its methods. :)
declare namespace java="java:org.distribution.Query";

(: The instance of the Java class. :)
declare variable $d:instance := java:new();

(: The function which executes the Java method Query#query(...)
and returns the results as nodes. :)
declare function d:query($q as xs:string, $urls as xs:string*)
as node()* {
for $i in java:query($d:instance, $q, $urls)
return parse-xml($i)
};
```

The *wrapper* XQuery file delegates the query task to the implemented Java code in the class org.distribution.Query by executing *java:query($d:instance, $q, $urls)*. BaseX initializes the defined Java class and translates the XQuery data types to the corresponding Java types. The simplified Java method is depicted as pseudo code in Algorithm 3.

This class is called when a distributed query request is initiated by the wrapper file. The *query* method receives the user defined query and a list of URLs as input parameter. Within the method BaseX types represent XQuery types and are used for the XQuery expressions to omit too many conversions between XQuery and standard Java types. Within this method there are as many threads as URLs exist created. Each thread ex-

ecutes the query by using the BaseX Java Client API. Afterwards, when all threads are done, the results are collected and returned as *Item* array. This array is then translated into an XQuery sequence.

Algorithm 3 query($query$: Str, $urls$: Value) : Item[]

```
1  resultItems:=new List
2  for u in urls do
3    item:=distribute query using BaseXClient API in a separate Java thread
4    resultItems.add(item)
5  end for
6  return resultItems
```

6.2.2 Reduce

A simple reduce process can be easily done within the initiator BaseX application as DQ2 illustrates:

```
(: Import of necessary module namespace :)
import module namespace d="http://basex.org/modules/distribute";

(: Counts the available 'the' words :)
declare function local:countThe($nitf as node()) {
let $texts := $nitf/descendant::text()
for $t in $texts
for $token in tokenize($t,'\s+')
where $token='The' or $token='the'
return '1'};

(: list of BaseX servers for the distributed evaluation :)
let $urls:=(...)

(: definition of distributed query :)
let $query:="
declare function local:countThe($nitf as node()) {
let $texts := $nitf/descendant::text()
for $t in $texts
for $token in tokenize($t,'\s+')
where $token='The' or $token='the'
return '1'};
```

```
declare function local:map(){
for $d in collection('nyt')
let $ct:=count(local:countThe($d/nitf))
order by $ct descending
return $d
};

<dq-result>{for $d at $p in local:map()
where $p<11
return $d}</dq-result>
"

(: execution of distributed query results and afterwards
computing Top-10 list using local queries :)
let $mapresult:=d:query($query,$urls)
let $reduceresult:=
<reduce-result>{
 for $topk at $p in {
  for $sr in $mapresult let $ct:=count(local:countThe($sr))
  order by $ct descending
  return $sr
 }
 where $p<11
 return $topk
}</reduce-result>
return <dq-results>{$reduceresult}</dq-results>
```

Thus, the user is able to use the distributed results as new input for the local queries to perform further aggregation. In the above example, the distributed query results return a top 10 list of news articles for each data node. Afterwards, the local query performs a new top 10 list computation using only these distributed query results (reduce process).

The presented configuration needs only one installation of the module on the BaseX initiator application, which is responsible for the query distribution. All other data servers remain standard BaseX servers and do not need any further configurations.

6.2.3 Reduce Extension

The local reduce approach performs well, but the requirements for the BaseX initiator application are not negligible. If the intermediate results from all data servers are quite large, the local reducer has to cope with large sequences of XML nodes. It is obviously

not the best approach, for example for a mobile device, since it has many restrictions like processor and main memory sizes. The introduced BaseX integration is able to delegate the map and reduce process to another, e.g., a more powerful, BaseX server or even to an existing data server. To enable this feature, only a modified module version has to be installed to all BaseX servers. The modification only concerns the *wrapper* XQuery file:

```
(: The module namespace used in the other queries. :)
module namespace d="http://basex.org/modules/distribute";

(: The java namespace. :)
declare namespace java="java:org.distribution.Query";

(: The instance of the Java class. :)
declare variable $d:instance := java:new();

(: The function which executes the Java method Query#query(...)
and returns the results as nodes. :)
declare function d:query($q as xs:string, $urls as xs:string*)
as node()* { for $i in java:query($d:instance, $q, $urls)
return parse-xml($i) };

(: HTTP request body :)
declare function d:querybody($query as xs:string){
let $rest-query:=
<rest:query xmlns:rest="http://www.basex.org/rest">
  <rest:text>{$query}</rest:text>
</rest:query>
let $body := <http:body media-type ="application/xml">
{$rest-query}</http:body>
return <http:request method='post'>{$body}</http:request>
};

(: HTTP request execution :)
declare function d:querymr
($q as xs:string, $urls as xs:string*, $rs as xs:string?)
as node()* {
let $b:=d:querybody($q)
return if($rs='') then http:send-request($b, $urls[1]) else
http:send-request($b, $rs)
};
```

The workflow is illustrated in Figure 6.2, where the BaseX client delegates the dis-

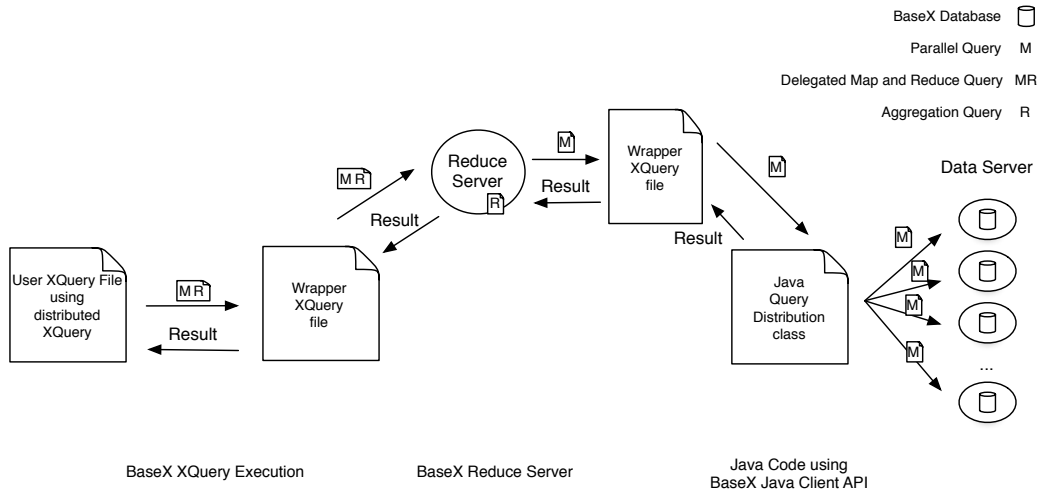


Figure 6.2: Workflow of the introduced reduce extension.

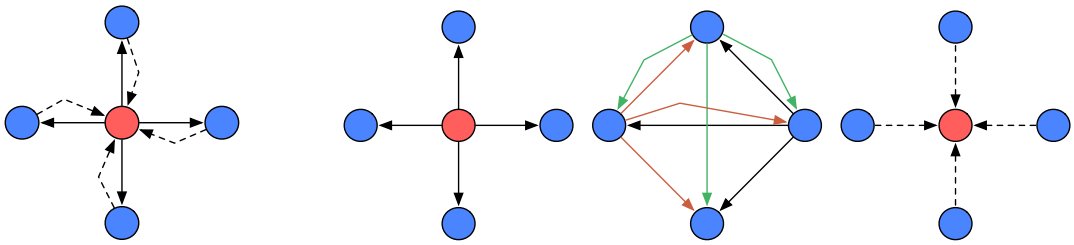
tributed query execution to a dedicated reduce server. The main idea is to use the HTTP module to delegate the complete distributed query execution to another BaseX server. *d:querymr(..)* function sends the map and reduce process (map and reduce query) to the reduce server. The reduce server then executes the distributed queries using again the same module on all defined data servers and computes the aggregation results afterwards. Following, it returns the final results via the HTTP module to the BaseX initiator application. The map and reduce query looks like:

```
import module namespace d="http://basex.org/modules/distribute";
(: list of BaseX HTTP servers for the distributed evaluation :)
let $urls:=(...)
(: map and reduce query :)
let $mapreducequery:="
import module namespace d='http://basex.org/modules/distribute';
(: Top 10 reducer counter :)
declare function local:countThe($nitf as node()) {...};

(: list of BaseX servers for the distributed evaluation :)
let $urls:=(...)

(: Mapping tasks :)
let $mapquery:=&quot;
(: Top 10 mapper counter :)
declare function local:countThe($nitf as node()) {..};

declare function local:map(){
for $d in collection('nyt')
let $ct:=count(local:countThe($d/nitf))
order by $ct descending
return $d
};
<map-result>{for $d at $p in local:map()
where $p<11 return $d}</map-result>
&quot;

(: Remote reduce process :)
let $mapresult:=d:query($mapquery,$urls)/descendant::nitf
let $reduceresult:=for $topk at $p in
  for $sr in $mapresult let $ct:=count(local:countThe($sr))
  order by $ct descending
  return $sr
 where $p<11
 return $topk
return
<dq-results>
{$reduceresult}
</dq-results>
"
(: Execution of map and reduce process on server 1 using HTTP :)
let $httpcall:=d:querymr(
$mapreducequery,$urls,'http://server5.example.com:20002/rest')
return $httpcall/dq-results/nitf
```

6.3 Challenges

The two main challenges with this BaseX integration approach are that the underlying XML database has to implement the EXPath Packaging System specification and the XQuery developer must decide whether a query has to be evaluated locally or whether the distributed query approach has to be used. Thus, the developer controls the execution type and the execution is not performed transparent within the BaseX system. Furthermore, the developer has to specify the data server locations before the distributed query execution. The distributed map and reduce queries are defined as a String type, which makes it difficult to debug the XQuery expressions.

Another great challenge is to join items with data of more than one collection.



(a) Join occurs on reduce node.

(b) Join occurs on each data node.

Figure 6.3: Join problem of several collections.

Figure 6.3 depicts the two possible solutions to join documents of two collections, which are distributed to a four node cluster using the RRS algorithm. An example here is to join articles of the NYT collection with an RSS collection to cluster news items concerning the same mentioned person. In this figure (a) proposes to perform the join on the reducer node (red node in the sub figure). The obvious disadvantage is that both collections must be collected and transferred to the reducer node to perform a join, which is not possible if both collection sizes are larger than the memory of the reducer node. The second sub figure (b) proposes another strategy to perform a join. First, the distributed query is distributed to all data nodes. Afterwards, the data nodes broadcast a query to the other data nodes to search for join partners. After having received the join partners, they perform the join on each data node and return the results to the initiator applica-

tion. This approach also has the disadvantage that the network has to cope with many messages, a problem that should be investigated in more detail.

6.4 Conclusion

This chapter introduced a light-weight alternative to the Hadoop MapReduce framework. It does not need a distributed file system, and the user has the opportunity to call all data contained on data servers and must not think in records. The user remains in the XQuery world and only has know our introduced EXPath module to benefit from parallel query evaluation. It is possible to aggregate the results of the distributed queries with a local reduce step. Futhermore, the user is able to delegate the whole map and reduce queries to another server, either a dedicated reducer machine or a simple data node, which allows the user application to be executed on a constrained device. Thus, each data node is also a coordinator node.

7 Future Work

In the wide research field of distribution and querying data important topics deserve special attention and need to be looked at in detail to further improve general processes. Aside from the replication of data ensuring availability, security issues, distributed transaction processing, and failure tolerance, improvement is of utmost importance in the areas of *Distribution*, *Querying*, and *Updating*. Following, is our detailed analysis of these processes.

7.1 Distribution

This master thesis introduced three possible distribution algorithms, RRS, AS, and P. A suitable distribution is needed to be able to store large data sets that cannot be stored on one machine. Furthermore, the choice of a distribution algorithm affects the query execution time. In our example the RRS algorithm was the best base for distributed querying. However, this distribution algorithm has several disadvantages: First, when one server fails, all collections will be affected. Additionally, all servers have to be called to access a collection, which means that all servers must be able to cope with many client requests and have to organize the buffer satisfactorily. To add a new data server in order to offer more storage capacities represents yet another challenge. It is therefore important to investigate whether the existing data has to be reorganized or not.
Furthermore, this thesis focused on distribution on a document level, which is an approach that is applicable to almost all other XML database implementations and not only with BaseX. On the other hand, it could be more suitable to distribute implementation specific to achieve an improved distribution performance. In the case of BaseX it should be investigated whether a distribution on the storage layer (table) could improve distribution performance.
When considering the architecture, we used a centralized distribution architecture to allocate data servers. This coordinating server is a single-point-of-failure and therefore, it should be analyzed whether a decentralized approach could be used instead of this

proposed, while still maintain the distribution performance.
Another challenge is not to focus on collections of XML, but to research how one large XML instance could be fragmented and distributed to several data servers. This topic is not only a challenge for distribution, but also a challenge for querying distributed trees.

7.2 Querying

In the area of querying, we focused so far only on querying distributed data and neglected *hot collections*, which are queried much more often than other distributed collections. One difficult task is to organize the buffer to allow a high client throughput. A solution would be to introduce replication of the distributed data to relax the hot collection requests.
Another interesting problem is to introduce an index to detect, which documents of a collection are located on which data server to omit calling all data servers that hold the named collection. With the introduced BaseX integration approach it is possible to peform distributed queries on all data nodes, which means each data node is able to act as initiator of a distributed query. An index must then be able to detect on all data servers on which other servers the requested documents are located. One possible solution is a hashing function considering the distribution of a document URI on a data node location, such as in peer-to-peer networks.
Moreover, the current approach forces an XQuery developer to define if a collection has to be called local or in a distributed way using the module distribution function. It would be better if the underlying system detects whether a collection is available on the local machine or if the collection is distributed and then makes the decision without constraining the XQuery developer. Furthermore, a distributed query optimizer could improve distributed query execution.

7.3 Updating

Although, update operations on distributed data are an important issue, this thesis did not allow to further analyse this topic. Updates affect the distributed data fragments enormous. For example, it would be possible that one distributed fragment could be nearly deleted, and the uniform distribution is no longer guaranteed. The same problem arises if additional documents have to be added to a given data server. A database system should consider such issues and, i.e., redistribute the data with a cost-saving algorithm.

7.4 More

Currently, there are also compression approaches to improve the performance of the introduced ideas. Here further investigation is needed to find out whether compression algorithms would increase the distribution proccess, e.g., to apply a compress algorithm before the fragment will be sent to a data server. Compression could also be applied within the querying process, where large intermediate results could be compressed before sending them to reducers.
Moreover, if a distributed query contains requests to several distributed collections, it should be analyzed whether a suitable parallelization algorithm could improve performance. Furthermore, it should be analyzed how network hopping could be avoided and network transmission minimized.

8 Conclusion

In this master thesis, we investigated distribution and querying of XML collections in detail. We evaluated the application of Hadoops MapReduce framework for distribution and querying of XML documents. As stated in Chapter 4, this approach works only for analyzing large data, where short reponse times are not as important as, e.g., failure tolerance. We introduced in Chapter 5 an alternative approach implemented in Java, which is able to use three different distribution strategies. The RRS distribution algorithm performed best for larger collection sizes and was the best base for querying afterwards.

In the case of BaseX, a native XML storage and XQuery processor, we evaluated the available API performance for distribution and querying and proposed to use the Java Client API for both requirements. Querying is done directly out of an XQuery expression and no further Java code has to be written to enable parallel query execution. Furthermore, as shown by our results, distributed query execution performs well.

We introduced a querying architecture that can be easily adapted of all EXPath specification supporters. The advantage is that XML database providers do not extend the XQuery language or implement own distribution functions within their architecture.

To the best of our knowledge our implementation, which is built on the top of BaseX, is the only open source XML database that supports parallel querying of distributed XML collections out of XQuery.

Bibliography

[ABC+03] Serge Abiteboul, Angela Bonifati, Grégory Cobéna, Ioana Manolescu, and Tova Milo. Dynamic xml documents with distribution and replication. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 527–538, New York, NY, USA, 2003. ACM.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[Bas12a] BaseX. HTTP Module. http://docs.basex.org/wiki/HTTP_Module, January 2012.

[Bas12b] BaseX. Packaging. http://docs.basex.org/wiki/Packaging, January 2012.

[BC07] Angela Bonifati and Alfredo Cuzzocrea. Efficient fragmentation of large xml documents. In *Proceedings of the 18th international conference on Database and Expert Systems Applications*, pages 539–550, Berlin, Heidelberg, 2007. Springer-Verlag.

[BCF+07] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Recommendation. http://www.w3.org/TR/xquery, January 2007.

[BCFK06] Peter Buneman, Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis. Using partial evaluation in distributed query evaluation. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 211–222. VLDB Endowment, 2006.

[BF05] Sujoe Bose and Leonidas Fegaras. Xfrag: A query processing framework for fragmented xml data. WebDB'05, 2005.

[BG03] Jan-Marco Bremer and Michael Gertz. On distributing xml repositories. WebDB'03, 2003.

[BPSM+08] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation. http://www.w3.org/TR/REC-xml, November 2008.

[CNP82] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, SIGMOD '82, pages 128–136, New York, NY, USA, 1982. ACM.

[Con12a] H2O Consulting. EXPath. http://expath.org/, January 2012.

[Con12b] H2O Consulting. EXPath Packaging System. http://expath.org/modules/pkg/, January 2012.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters . In *OSDI*, 2004.

[Edl11] Prof. Dr. Stefan Edlich. NoSQL - Not only SQL. http://nosql-database.org/, October 2011.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[FJM+07a] Mary Fernández, Trevor Jim, Kristi Morton, Nicola Onose, and Jérôme Siméon. Dxq: a distributed xquery scripting language. In *Proceedings of the 4th international workshop on XQuery implementation, experience and perspectives*, XIME-P '07, pages 3:1–3:6, New York, NY, USA, 2007. ACM.

[FJM+07b] Mary F. Fernàndez, Trevor Jim, Kristi Morton, Nicola Onose, and Jérôme Siméon. Highly distributed xquery with dxq. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1159–1161, New York, NY, USA, 2007. ACM.

[Fou11] Apache Software Foundation. Hadoop. http://hadoop.apache.org/, August 2011.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, 2003.

[GHM+07] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C Recommendation. http://www.w3.org/TR/soap12, April 2007.

[GKW08] Sebastian Graf, Marc Kramis, and Marcel Waldvogel. Distributing xml with focus on parallel evaluation. DBISP2P'08, 2008.

[GLG10] Sebastian Graf, Lukas Lewandowski, and Christian Grün. JAX-RX - Unified REST Access to XML Resources. Technical Report, University of Konstanz, Konstanz, BW, 2010.

[Gra08] Sebastian Graf. Verteilungsansätze von großen Datenmengen. Master's thesis, University of Konstanz, Germany, October 2008.

[Grü10] Christian Grün. *Storing and Querying Large XML Instances*. PhD thesis, University of Konstanz, Germany, 2010.

[KATK10] Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos, and Nectarios Koziris. Distributed indexing of web scale datasets for the cloud. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, MDAC '10, pages 1:1–1:6, New York, NY, USA, 2010. ACM.

[KCS11a] Shahan Khatchadourian, Mariano Consens, and Jérôme Siméon. Chuql: processing xml with xquery using hadoop. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '11, pages 74–83, Riverton, NJ, USA, 2011. IBM Corp.

[KCS11b] Shahan Khatchadourian, Mariano Consens, and Jérôme Siméon. Having a chuql at xml on the cloud. AMW'10, 2011.

[Kos00] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32:422–469, December 2000.

[MMWK10] Ashok Malhotra, Jim Melton, Norman Walsh, and Michael Kay. XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition). W3C Recommendation. http://www.w3.org/TR/xpath-functions, December 2010.

[NCWD84] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9:680–710, December 1984.

[PM02] Vassilis Papadimos and David Maier. Distributed queries without distributed state. WebDB'02, 2002.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, pages 974–985, 2002.

[VCL10] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 495–506, New York, NY, USA, 2010. ACM.

[Wei10] Andreas Weiler. Client-/Server-Architektur in XML Datenbanken. Master's thesis, University of Konstanz, Germany, September 2010.

[ZZYH10] Qi Zhang, Yue Zhang, Haomin Yu, and Xuanjing Huang. Efficient partial-duplicate detection based on sequence matching. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 675–682, New York, NY, USA, 2010. ACM.

List of Figures

List of Tables