ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Master's Thesis Nr. 70

Systems Group, Department of Computer Science, ETH Zurich

in collaboration with

28msec Inc.

Big Data Query Parallelization

by

Julien Ribon

Supervised by

Prof. Dr. Donald Kossmann
Dr. Ghislain Fourny
Dr. Matthias Brantner
Dr. Till Westmann

September 2012 – March 2013

inf | Informatik
Computer Science

28 msec

Master's Thesis Nr. 70

Big Data Query Parallelization

Julien Ribon
jribon@student.ethz.ch

supervised by
Prof. Dr. Donald Kossmann
Dr. Ghislain Fourny
Dr. Matthias Brantner
Dr. Till Westmann

Systems Group
Institute of Information Systems
Department of Computer Science
ETH Zurich

in collaboration with
28msec Inc.

September 2012 – March 2013

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
inf Informatik Computer Science
Systems@ETH Zürich
28 msec

Acknowledgments

My first indebted thanks go to Donald Kossmann for allowing me to perform my thesis in collaboration with 28msec and on such an enthralling topic. This thesis was a very enriching experience because it gave me the opportunity to work in the industry and, at the same time, to still take part in a research process as it would be the case at ETH. Seeing how a company was concretely working (including teamwork, brainstorming sessions, conference calls, project planning and release) provided me with valuable knowledge that would be useful for my entire professional career.

I was thrilled to collaborate with all my colleagues at 28msec. I am very thankful for their time and support, especially to Ghislain Fourny who inspired me a lot with his vision and way of thinking and who always gave me precious time and advice though my thesis. I owe my deepest gratitude to Matthias Brantner and Till Westmann for providing me with constructive and very much appreciated remarks during the realization of this thesis. Many thanks, again, to Matthias who keeps the objectives crystal clear and guides the project toward the right direction.

I am also very grateful to my other colleagues who took the time to answer my questions even if they were not directly related to the project. I would like to express my deepest gratitude to David Graf who came to my help when I was stuck despite of having other responsibilities. Many thanks to Dennis Knochenwefel for explaining me the functioning of Sausalito, as well as Alexander Kreutz for showing me how to integrate my code into the Web portal. I would also like to thank William Candillon and Luchin Doblies for the great time at XMLPrague and their active work on the Cloud9 IDE used during the project development.

28msec was a very stimulating environment which gave me the opportunity to delve into the NoSQL technologies and allowed me to participate in XMLPrague 2013 which was my first IT conference outside the framework of ETH. During this conference I had the pleasure to meet brilliant people from the FLWOR Foundation and other associations or companies. Let me thanks them all as well.

I am very grateful to my mentor, Gustavo Alonso, for his continuous support during the accomplishment of my Master at ETH, and for allowing me to become a teaching assistant of the Systems Group.

Affectionate thanks to my parents, Luigi and Liliane, brothers, Stéphane and Florian, and my other relatives for their kind words and presence during the most challenging times. Warm thanks also to all my dear friends, especially Aurore, for their invaluable support.

Abstract

The amount of data stored worldwide has reached the exabyte range (one quintillion of bytes). The first generation of databases based on relational algebra is hitting its limits. Therefore, companies dealing with Big Data such as Google, Facebook, or Twitter had to come up with their own frameworks and database systems in order to cope with such data scale. In parallel, a second generation of databases based on a semi-structured model like XML, YAML, or JSON rather than traditional relation databases has been emerging during the last ten years. All these systems are part of the NoSQL family which provides higher scalability and availability than traditional relational databases.

In order for storage systems to scale, the data is partitioned horizontally into shards that are then replicated across several nodes. As for querying at large scales, declarative - parallelizable and highly optimizable - query languages are needed. This is what XQuery is to XML. Since the core of XQuery is free of side-effects, a huge parallelization potential lies in the execution of programs written in this language. Data-intensive queries can be distributed on a farm of computers, each of them working on a shard, while computation-intensive queries can be parallelized by duplicating sub-parts of the query plan.

In this master's thesis, we investigate how execution of queries on a large amount of highly distributed data can best be orchestrated using XQuery. We come up with use cases to clearly illustrate the parallelization challenges, and present a logical model, called the Tree-Hammock pattern, that allows to parallelize XQuery programs. We also design and implement a framework which exposes parallelization capabilities to XQuery developers and allows them to manually parallelize their queries. A first version of the parallelization infrastructure is developed on top of the 28msec's platform and used to evaluate some of the use cases presented in this thesis. Experiments are then carried out to validate our approach and demonstrate performance improvements compared to sequential query processing, as well as to study which factors have the most impact on our framework.

Contents

Chapter 1

Introduction

Nowadays, limits in the size of data sets that can entirely be processed in a reasonable amount of time are on the petabyte scale. Data sets keep growing day after day and this trend is not about to change. In a near future, we will need to cope with quantities of data that will have increased by an order of magnitude. It is already beyond the ability of traditional database systems to manage and process big data in a tolerable amount of time. First-generation relational databases are hitting their limits as they require massively parallel software running on hundreds or thousands of processing nodes. As a consequence, a second generation of database systems have been developed to deal with issues related to big data, such as scalability and availability. These systems do not rely on relational model and belong to a new category of databases, called NoSQL.

Databases from the NoSQL family aim therefore at providing a high performance alternative to relational databases. They have been especially designed to store, manage and retrieve great quantities of data. There are several kinds of NoSQL storage systems, but we essentially distinguish between two of them: key-value stores and document-oriented databases. The data representation of key-value stores is based on attribute-value pair, where the data model is expressed as a collection of $<$*key, value*$>$ tuples. On the other hand, document-oriented databases are based on a semi-structured model, where data items, called *documents*, follows a tree-like structure.

In order for NoSQL storage systems to scale, their data are partitioned horizontally into chunks that are then replicated among multiple nodes. Querying at a large scale and on highly distributed (possibly denormalized) data becomes therefore necessary. Unfortunately, one major issue of these second-generation database systems is that they do not provide a high-level declarative query language like SQL. Instead, querying these data stores is data-model specific and often requires the use of data-intensive frameworks, like MapReduce, for processing their large data sets. Even though NoSQL frameworks provide an alternative to query languages, they lack in capabilities to query nested data representation such as documents. Moreover, these frameworks are limited in runtime performance and insufficient to support flexible workloads. In particular, they suffer from severe performance issues when multiple passes on the data are necessary.

When it comes to querying semi-structured data at a large scale and in a distributed environment, declarative query languages are needed because they offer highly parallelizable and optimizable opportunities. XQuery is a declarative and functional programming language that is precisely designed to query collections of semi-structured data formatted in XML. Because the core of XQuery is side-effect free, a huge potential to parallelize processing lies in this language, both at the instruction and data level. As data-intensive queries can be

distributed on a cluster of machines to process different portions of the data, computation-intensive queries can be parallelized by duplicating sub-parts of the query plan. Besides, since XQuery is both declarative and functional, it can be used to combine different database technologies ranging form parallel processing to data-intensive computing.

In this thesis, we investigate how parallel query execution over large amounts of highly distributed data can be orchestrated using XQuery. At the outset, our goal was to develop a logical model, called Tree-Hammock pattern, that could be used to parallelize and distribute queries written in XQuery. The Tree-Hammock pattern is a dual topological model that handles both parallel data and instruction processing. This model can therefore be subdivided into two distinct parts, where the tree pattern is computation-intensive and the hammock pattern data-intensive. The tree pattern enhances the concept of query plan and allows the use of traditional parallel techniques. As for the hammock pattern, it is designed to parallelize the tuple streams that flow through FLWOR expressions. Within this thesis, we essentially focus on the hammock pattern, since data-intensive processing represents the most challenging part when it comes to querying distributed data sets.

We also aim at designing and implementing a framework that leverages the Tree-Hammock pattern to evaluate XQuery queries in parallel. To this extent, we come up with some use cases to point out the key challenges and to illustrate how queries can be run on top of our framework. The resulting scalable infrastructure combines ideas and techniques coming from different database technologies, such as distributed query processing, parallel database, semi-structured data management, and data-intensive computing.

The first version of our framework allows users to manually parallelize predicate queries while using some parallelization capabilities to improve query evaluation. When a user invokes our parallel framework, computation is brought to the data by wrapping the user query into a function to be later imported by query processors. The wrapped query is split into several jobs representing the whole query plan and shipped to an asynchronous queuing system. Once jobs have been scheduled by this queuing system, query processors fetch jobs from the queue (one at a time) and process the query on some portion of the data. Data results generated by the query processors are materialized inside a new collection that is ultimately retrieved by the end-user.

In order to validate our complete approach, we carried out some experiences including bulk loading of data in the cloud and answering point queries on a large data set. These experiments have confirmed our expectations by demonstrating clear performance improvements from our parallel infrastructure compared to serial processing.

The remainder of this thesis is organized as follow. In chapter 2, we present earlier work done in parallel databases, distributed XQuery processing, and data-intensive computing. This chapter also provides the reader with some basic concepts and background knowledge. Chapter 3 describes our new Tree-Hammock topology used to parallelize XQuery expressions, as well as the design principles and structure of our system architecture. Chapter 4 then presents some use cases to illustrate the challenges of processing large datasets in the cloud. In chapter 5, we present a first implementation of our system architecture that allows predicate queries to be run in parallel. Chapter 6 presents some experiments together with their results, which were carried out to validate both the overall approach and its implementation. Finally, we suggest possible enhancements for our parallel infrastructure in chapter 7 and end this document with some concluding remarks in chapter 8.

Chapter 2

Related Work

In this chapter, we present previous works done in the field of parallel databases, distributed XQuery processing, as well as recent frameworks used for processing large data sets.

2.1 Parallel Databases

In parallel processing, operations are performed simultaneously as opposed to serial processing, where computation happens sequentially. Parallel processing has been the subject of intense research in the last decades and remarkable efforts have been done to integrate this technology in data management systems. Parallel databases represent a very mature technology that already counts numerous SQL-based products developed by IBM, Oracle, Teradata, or Tandem. Parallel databases aim at improving performance of various operations, such as data bulk loading, index creation, and query evaluation by exploiting multi-core processors and parallel disk I/O operations.

Distributed databases constitute another effort to improve database performance (in reliability and availability) by scattering the data among several machines. In these systems, a query is split into a set of sub-queries or tasks that are then executed in parallel. The notion of *task* as construct of a parallel query is not a new concept and can be found in [11]. The idea is to use horizontal and/or vertical partitioning so that sites can concurrently process an operation on different portions of a relation.

Historically, distributed databases are not considered as parallel databases, because they involve autonomous geographically-distributed computers with independent software, while parallel databases involve a single (local) system. Nevertheless, with the arrival of cloud computing and virtualization techniques the separation between these two types of databases is not as clear today. Hence, when we talk about parallel databases, we usually refer to the more general category of parallel *and* distributed databases.

In the remainder of this section, we give a brief description of the reference architectures of parallel databases. We also introduce the different parallelism techniques used in these architectures and present some previous works from the old relational database literature.

2.1.1 Parallel Architectures

There exist three major types of resources in databases system: processors, memory, and disks. Based on the way these hardware components interact with each other, parallel databases system can be separated into three categories:

- *Shared Everything* or *Shared Memory*: disk and memory are shared among processors and the entire system is connected with a fast network (such as a bus). General-purpose multi-processor computers belong to such architecture.

- *Shared Nothing*: each processor has its own memory and disk that together form a processor-memory-disk group called node. No direct access from one node to the memory and disk of another can take place. Instead, nodes communicate by sending messages over an interconnection network.

- *Shared Disk*: only disks are shared in this architecture. Processors have their own memory buffer to store data read from disks and are not allowed to access in-memory data from other processors.

Shared-memory architecture has two key advantages: first, it eases load balancing and, second, has no communication cost, as message exchange and data sharing among processors are done through the shared memory. However, this architecture has a major drawback, i.e. it is hardly scalable beyond 40 processors. On the other hand, shared-nothing architecture excels in scalability but has a much higher communication cost. This architecture has a potential to scale up to thousands processors and provides a better reliability and availability. In order for shared-nothing systems to perform well, the processing load must be approximately the same between nodes. If the data is highly skewed, one or more nodes will become bottlenecks and stale parallel operations. This problem can be resolved by using good load-balancing algorithms. Shared-disk architecture is a compromise between the two other approaches. In this sense, this architecture scales better than shared-memory but not as well as shared-nothing. In the same manner, shared-disk wins in term of communication costs against shared-nothing but loses against shared-everything architectures. A more precise comparison of these architectures with their respective advantages and drawbacks can be found in [2].

Over the years, it seems that consensus for parallel architecture based on shared-nothing has been widely adopted. Other approaches combine these different architectures to create hybrid designs, proposed in [2, 3, 4]. An example of such hybrid architecture consists of a set of clusters linked by an interconnection network, where the clusters form a shared-nothing architecture and each individual cluster consists of a shared-memory architecture. We will see that this design is, in some sense, close to our infrastructure presented in chapter 5.

2.1.2 Types of Parallelism

Parallelism is natural for processing in relation database systems. Relational queries are ideally suited for parallel execution because they are composed of uniform operations applied to uniform streams of data [1].

As a consequence, multiple operators can be combined into a highly parallel dataflow graph, such that one operator consumes the new relation produced by another. By chaining operators together and letting the data stream flows from one operator to another, *pipeline parallelism* can be used to allow operators to work in series. Each operator then performs one step of a multi-step process, where each tuple is consumed by the next operator as soon as it has been produced. Furthermore, by partitioning the input data among multiple processors, an operator can often be split into many independent sub-operators, each working on a certain portion of the data. This technique, called *partitioned parallelism*, de-clusters tuples among several nodes such that operators (or entire queries) can be executed independently from each other.

Different levels of parallelism can therefore be applied to query processing:

- *inter-query parallelism*, where several independent queries are executed concurrently at different sites.
- *inter-operator parallelism*, where the operators of a single query are run concurrently, possibly following a pipeline fashion.
- *intra-operator parallelism*, where multiple processors or nodes work together to compute a single operation, e.g. scan, sort, or join.

As data are distributed across several sites, we can consider two more general types of parallelism, namely *data parallelism* and *instruction parallelism*. Data parallelism focuses on distributing the data across different parallel computing nodes, while instruction parallelism simply exploits parallelism between operations inside a query plan. In this thesis, we focus essentially on data parallelism.

2.1.3 Further Works in Parallel Databases

Much further works have been done in the field of parallel databases, covering subjects such as data partitioning, load balancing, parallel operators, or parallel query optimization.

Important research efforts were devoted to parallel algorithms for various operations such as sorting and join. Parallel sorting is widely used to facilitate other operations such as merge join, duplicate elimination, and aggregation functions and, thus, has received particular attention [5, 6, 7].

Join is another frequently used operation. Hence, a comparable effort has be expended in the development of efficient join algorithms [3, 8, 9], especially in the case where two very large relations must be processed. Among many algorithms, the well-know *Grace* hash join [13] exploits parallelism by first partitioning the data into buckets using a hash function and then applying smaller join operations in parallel on independent buckets.

We can also find significant works that tackle other issues from parallel database systems, such as data skewness [10, 11] or parallel query optimization, where [12] was one of the first to highlight major issues in developing optimization techniques in a parallel environment. Load balancing and data-skew avoidance play a central role in our architecture, as we try to keep data partitioned among many data chunks to both avoid bottlenecks and benefit from a high degree of parallelism.

2.2 XML and XQuery in the Cloud

In this section, we introduce recent work done on distributed XQuery processing and XML data treatment in the Cloud. We distinguish between two categories of work. The first category does not consider processing of large amounts of data and rather focus on distributing XQuery queries on remote servers [15, 16, 17, 18]. The second category of research work is based on the extension of existing Big Data frameworks to add support for XML processing [19, 20, 33].

2.2.1 Distributed XQuery Processing

Techniques presented in this section focus exclusively on distributing XQuery processing across remote machines and are not based on data-intensive computing. Even if these techniques actually achieve parallelization of XQuery queries, they significantly differ from our approach as they do not treat large data sets. Furthermore, to offer programmers the ability to manually distribute their queries, the papers [15, 16, 17, 18] presented an extension of the XQuery language. In this thesis, we do not make any language extension and rather use pure XQuery to enable parallel query processing.

XBird/D

In their paper [15], the authors of XBird/D use a divide-and-conquer approach that divides a query into sub-queries, recursively, and ships these sub-queries on remote query processors. These recursive invocations form a hierarchy of servers, each dealing with a sub part of the query. Their approach uses a simple asynchronous, consumer-producer architecture to exchange data and sub-queries between the processors. Queries and data are wrapped into so-called *proxy sequences*, which are sent to and ultimately evaluated by remote servers in a lazy fashion.

XRPC

XRPC [16] is mechanism that enables a query to be distributed and executed over different XQuery processors, like Saxon or Zorba. It focuses on inter-operability between heterogeneous XML data sources and aims at serving as a target language for a distributed XQuery optimizer. XRPC enhances XQuery functions with the concept of remote procedure call (RPC). The idea is to use the functional aspect of XQuery together with query decomposition techniques [14] to transform data-shipping queries into function-shipping queries such that the processing goes to the data [25]. Decomposition techniques are used to split the full query into multiple sub-queries, recursively, that are then wrapped into RPC function and shipped to servers for remote execution. This forms a tree of remote query invocations similar to the one presented in XBird/D [15].

DXQ

DXQ [17, 18] is based on a new infrastructure that gives the ability to deploy networks of servers and use them to remotely invoke XQuery programs. This infrastructure allows to run modern applications that go beyond the traditional client-server architecture, following a P2P architecture. DXQ exports a module of XQuery functions which can then be called remotely from every client application or DXQ server. This technique allows, in particular, DXQ servers to move arbitrary parts of a query among each other to achieve parallel processing.

We are convinced that using an asynchronous architecture, as in XBird/D, is the right approach when it comes to parallelize queries. Moreover, using function or closure to encapsulates and distribute (sub-)queries across servers seems to be a promising strategy.

Nevertheless, all these approaches [15, 16, 17] suffer from similar issues when its comes to handling large volumes of data. In particular, the hierarchical invocations of remote query processors and the exchange of intermediate results both constitute severe bottlenecks. Also, even if P2P systems allow independent executions of remote queries, producing an optimized distributed query plan becomes difficult because there is no overview of available resources.

Finally, using a synchronous, blocking communication mechanism like XRPC may not be appropriated when large amounts of data are exchanged between system components.

2.2.2 Extension of Frameworks

The second category of work [19, 20, 33] attempts to bring XML and XQuery support to existing Big Data frameworks like MapReduce [21] and Hadoop [35]. We present these frameworks independently from XQuery in the next section.

ChuQL

ChuQL [19] is an example of research effort devoted to extend MapReduce with XQuery support. The ChuQL language is a XQuery extension that exposes the Hadoop framework to XQuery programmers. Its implementation incorporates the key-value data model of MapReduce and uses it to distribute XQuery computation among multiple engines.

Xadoop

Another work that brings support for XML processing on top of MapReduce is Xadoop [33]. Xadoop aims at placing XQuery as an alternative language for Hadoop next to Pig, Hive, or Jaql. It attempts to automatically parallelize XQuery code and transfer data between concurrent *map* and *reduce* tasks.

MRQL

MRQL is a whole query language for large-scale analysis of XML data, which is also deployed on a MapReduce environment [20]. MRQL is similar to Xadoop and ChuQL in the sense that it enables XML processing using MapReduce, but differ from them because it does not offer XQuery as a query language.

It must be pointed out, here, that we do not base out approach on existing Big Data frameworks. Our approach, presented in chapter 3, is an attempt to parallelize XQuery expressions on a new system architecture that is precisely designed to handle large amounts of higly distributed data.

2.3 Big Data Frameworks

As data is rapidly growing, it has become necessary to develop highly efficient and scalable frameworks. Therefore, a second generation of database systems, called *NoSQL* (Not Only SQL), have been developed to query large amounts of data in a reasonable amount of time. In order for these NoSQL databases to scale, they use different data models than the traditional relational data model.

In this section, we present some state-of-the-art, data-intensive computing platforms [21, 22] that have been especially designed to store, manage and retrieve Big Data.

2.3.1 MapReduce

MapReduce is a programming model, developed by Google, for processing and generating large data sets which is typically used to distribute computing on top of a cluster of machines [21]. There is also an open-source project, called Apache Hadoop [35], that is derived from MapReduce and Google File System (GFS).

In order to query data using MapReduce, users have to specify a *map* function and a *reduce* function that are both defined with respect to the <*key, value*> data model. The *map* function takes a series of <*key, value*> pairs in a certain data type domain, processes each, and returns a list of pairs in another domain: $map(k1, v1) \rightarrow list(k2, v2)$. Once the map function has been applied to all data items, the *reduce* function is invoked in parallel on each group of values associated with the same intermediate key. The reduce step then produces zero or more values in the same domain as the input: $reduce(k2, list(v2)) \rightarrow list(v3)$.

Additionally, a shuffling stage takes place between the map and the reduce function, which starts as soon as the first map task completes. This stage is needed in order to group data items that share the same key, such that they can be processed together during the reduce stage. The reduce stage can therefore not start before the previous map stage has been entirely completed. The shuffle can sometimes take longer than the time taken by map and reduce computations, as it requires most data items produced during the map stage to be send over the network in order to be reduced.

Hence, MapReduce is designed to process and potentially produce large data sets. It is best suited to query <*key, value*> data stores but lacks support for semi-structured, deeply nested data such as JSON or XML. This is the reason why many attempts have been undertaken in order to integrate some support for semi-structured data [19, 20, 33]. Another problem of MapReduce is that many real-life scenarios require not one but several map-reduce steps. Multiple map-reduce passes on large volumes of data can truly impair the overall processing, as, in each step, a new shuffling stage is required to re-partition the data. In this thesis, we are interested in efficiently querying semi-structured data naively using XQuery, while avoiding the overhead of running several map-reduce steps.

2.3.2 BigQuery

BigQuery is a web service, developed at Google, that enables interactive analysis of large datasets [34] and that can be used complementary to MapReduce. BigQuery is based on Dremel [22], a distributed and scalable query system using a column-oriented layout to store nested data.

In BigQuery, users express their queries in SQL before sending them to the Web API. Most queries are supported as long as they do not involve large intermediate results. This is because all intermediate data items need to be exchanged over the network between system components. For example, join operations are supported, but only in the case where one of the two joined tables is small enough. Hence, BigQuery is best suited for aggregation queries, which keep intermediate and final results reasonably small even with billions of input records.

BigQuery has a tree-like computation architecture that allows a user query to be divided into sub-queries and processed in parallel by a cluster of machines. The root node is used as master to coordinate and optimize distribution and parallelization of the query execution. After creating sub-queries, the root sends them down to leaf nodes in order for each sub-query to operate on a small portion of the data. Once the data is processed at the leaves, results are sent back to parent nodes and flow back up to the root. Finally, results are returned to the end-user in JSON format.

Having a tree-like architecture has some interesting properties that enables an intuitive way of querying distributed data. Indeed, there are three ways of thinking about such architecture:

- a B-Tree built on-the-fly to enable fast table scan, in the same manner as a real B-Tree data structure is used for indexing in traditional database systems;

- the MapReduce programming model, where the root node is the master, intermediate peers are the reducers, and the leaf peers are the mappers (with no shuffling stage);
- a way of decomposing query algebra into inter-parallel portions, where (in the most extreme case) each peer is responsible for one operation.

Therefore, this hierarchical architecture finely embraces the structure of a query plan, where projection and selection are applied at leaves, aggregation and ordering both at leaf and intermediate nodes, and some final processing (e.g. further projection) at the master/root node.

Thanks to this architecture, BigQuery together with Dremel are able to take big data sets as input, aggregate intermediate data items together, and return a much smaller amount of data to users. We refer to this type of query processing as *big-to-small*. There are other aggregation frameworks such as BigQuery that allow *big-to-small* data-intensive queries [23, 43]. Unfortunately, this architecture presents some impediments such as the incapacity of dealing with large intermediate results.

In the next chapter, we present a new computation architecture that enables *big-to-big* query processing, i.e. that is able to take a big amount of data as input and return an equally large quantity of data as output, and can therefore support more than just small-join or aggregation queries.

Chapter 3

Approach: Underlying Model

In this chapter, we introduce a new approach that is used to execute XQuery expressions in parallel. This approach consists of two parts: a new schematic description of the arrangement of XQuery expressions, called the Tree-Hammock pattern, and a logical system architecture that consists of a cluster of identical nodes or peers.

Before introducing this approach, we analyze and categorize the different kinds of XQuery expressions [29] such that they can be naturally mapped on our topology.

3.1 XQuery Expressions

XQuery is a declarative language relying on the composition of its expressions, such that a parsed XQuery program can be represented by a tree of expressions. Each expression takes as input zero or more sequences of items through its operands and returns as output a sequence of items or an error.

XQuery expressions can be classified in two different types: those which contain variable bindings as operands and those which do not. We call them *complex expressions* and *simple expressions*, respectively. These two categories can in turn be subdivided into other categories.

In this section, we give one example of XQuery expression for each category. The complete classification of all XQuey expressions can be found in Appendix A.

3.1.1 Simple Expression

The vast majority of XQuery expressions are *simple* in the sense that they logically evaluate their operands only once and do not bind any variable before evaluating any operand. For example, an additive expression, like `1+2`, is evaluated as follows. First, both its left-hand-side operand and right-and-side operand are evaluated exactly once. Then, results of these operands are then added together to produce the final result item.

Simple expressions can themselves be subdivided into several distinct categories based on their arity, i.e. the number of operands that are taken as input. Figure A.1 presents these different categories with some examples.

0-ary Expression

This is the most simple type of expression. It has no operand and just produces one item.

Number and string *literals* as well as *named function reference* belong to this category (e.g. `1`, `"foo"`). For instance, the hello-world program

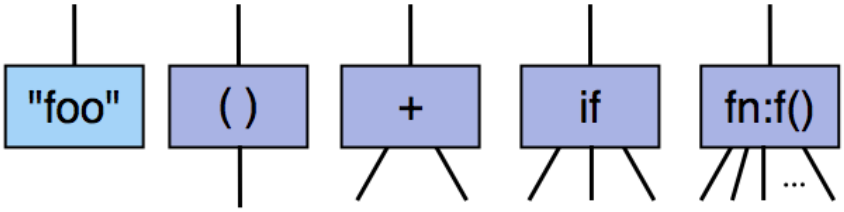


Figure 3.1: Simple Expressions

```
"Hello, World"
```

returns a sequence of one item of type xs:string, i.e. a string literal.

Unary Expression

Unary expressions have a single operand. Among all unary expressions, the most encountered is probably the *parenthesized expression*, but there are other expressions also belonging to this category such as *ordered/unordered expressions*, *unary arithmetic operators*, *expressions on sequence types*, *validate expressions*, as well as *extension expressions*.

A typical expression having only one operand is the parenthesized expression *"(" Expr ")"* that is used to enforce a certain evaluation order. This kind of primary expression can be seen as a simple forwarding operation that takes an operand as input, evaluates it, and outputs its result. For example, `(1)` contains a number that is evaluated as described in the previous section and then output by the parenthesized expression as a sequence of one item.

Binary Expression

As its name suggests, a binary expression has two operands that are both evaluated before combining them according to the semantics of a certain binary operator. There are plenty of binary expressions in XQuery 3.0 such as *binary arithmetic operators*, *string concatenation*, *comparison expressions*, *logical expressions* and *sequence expressions*.

It is important to highlight here that some binary expressions presented in this section are associative, such as sequence expression, string concatenation, logical expressions, as well as addition and multiplication operation. As a consequence, operands of associative expressions do not require to be evaluated in a specific order, and can therefore be considered as N-ary expressions once multiple operators are used together.

XQuery provides binary addition, subtraction, multiplication, division and modulus operators (+, -, *, div, idiv, mod). Arithmetic expressions are grouped from left to right. Therefore, expressions from the left-hand side are first evaluated before combining their results with the right-hand sided operands. For example, an operation such as `3*7 mod 5` returns `1` because the multiplication is evaluated before the modulus.

Ternary Expression

There is only one type of expressions having exactly three operands: *conditional expressions* are based on the combination of keywords **`if`**, **`then`**, and **`else`**. For example:

```
if (1<2) then "foo" else "bar"
```

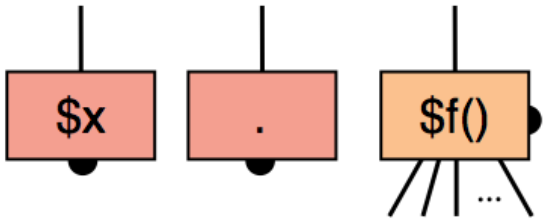


Figure 3.2: Complex Expressions

Conditional expressions evaluate all three operands, and output the result either of the *then* or *else* clause according to the boolean value computed in the *if* clause. In the above example, `"foo"` is returned because `(1<2)` is always `true`.

N-ary Expression

This kind of expression has a certain number of operands, often greater than three. These expressions evaluate all operands and concatenates their values, while preserving spatial ordering. *Static function calls*, *inline function expressions*, *partial function applications*, *try/catch expressions*, *switch expressions*, *typeswitch expressions*, and *constructors* all belong to this category.

All static (build-in or user-defined) function calls such as `fn:concat("a","b","c")` are considered as N-ary expressions, where N is the number of parameters that can be passed to the function. Such expressions have as many operands as the function's arity indicates. It evaluates all operands, calls the function and outputs the resulting sequence of items.

3.1.2 Complex expression

A complex expression is an expression that has bindable expressions as operands (or accesses a bindable expression after getting a reference thereto from an input) and evaluates them by binding values dynamically. Such expressions are typically used when the values of bindable expressions is changing at run-time, for example, in a loop.

Figure A.2 illustrates the different complex expressions. As opposed to simple expressions, we do not take arity into account to sub-classify complex expressions. Instead, we only distinguish between primitive bindable expressions (having exactly one free variable) and expressions using or consisting of bindable expressions (having one or more free variables and possibly other simple input expressions). Bindable expressions are represented by semicircles.

Primitive Free-Variable Expression

A primitive bindable expression is an expression that requires a single variable binding to occur before it can be evaluated. We distinguish two expressions belonging to this category: variable expressions and context-item expressions.

Variable expressions such as `$x` can only be evaluated if the variable `$x` has been bound to a value. If no value has being assigned when evaluated, an error occurs.

Context item expressions `.` and `..` represent the current item and its parent respectively. Their reference (bound item) is likely to change at run-time. As for variable expressions, the context item `.` can only be evaluated if the context item has been bound to an item. If no item is bound to the expression, an error is returned.

Expressions with Free-variable Expressions

Potentially all simple expressions can become complex expressions as soon as one replaces one of their operands by a free variable. However, there are some complex expressions that do not have a simple counterpart, i.e. these expressions cannot be used without free-variable expressions. *Dynamic function calls*, *filter expressions*, *path expressions*, *simple map operator*, *quantified expressions*, and *FLWOR expressions* are such kind of complex expressions. In this section, we only present FLOWR expressions, as most other complex expressions can be expressed using FLWORs. Other complex expressions can be found in Appendix A.

The FLWOR expression is a flexible, multipurpose expression that is composed of multiple clauses. It can be used for different tasks such as iterating over sequence, joining multiple documents, or grouping and aggregating some query results. FLWOR expressions are analogous to SQL statements (select, from, where, etc.) and provide join-like functionality to XML documents in the same way as SQL can be used to join SQL tables. *FLWOR* is an acronym for its clauses (*for*, *let*, *where*, *order by*, and *return*) available in XQuery 1.0. However, in XQuery 3.0, three new clauses were added, namely *window*, *count*, and *group by*.

The evaluation of FLWOR expressions is different from other XQuery expressions. Its semantics is based on the concept of *tuple stream*, i.e. an ordered sequence of many tuples, where a *tuple* is defined in the XQuery specification [29] as being a set of multiple named variables, each bound to a XDM instance. An individual tuple steam is uniform, meaning that all its tuples contain variables with the same names and static types. Every FLWOR expression generates its own tuple stream, which is then successively modified by its clauses. Therefore, each clause consumes an input tuple stream produced by the previous clause and produces a new output tuple stream. Only the first clause does not take any tuple stream as input, but simply creates a new one.

A FLWOR expression starts with an initial clause, followed by an indefinite number of intermediate clauses and ends with a final (return) clause. An initial clause is the first expression of a FLWOR expression, it is either a for, window or let clause. The initial clause generates a sequence of bound variables, i.e. a tuple stream, that will be used as input by the next clauses. An intermediate clause can be any possible clause (except *return*). There can be zero or more intermediate clauses inside a FLWOR expression. The final clause is always a return clause. A detailed description about how each clause behaves is given in the remaining of this section.

The *for* clause is used for iterating over an input sequence. Each variable in a for clause is bound in turn to each item present in the sequence. If the clause contains more than one variable bindings, it has the same semantics as if multiple for clauses were used instead. In this clause, a tuple represents a single item from the sequence.

```
for $x at $i in (2,4,6,8,10), $y at $j in (1 to $x)
```

The *let* clause declares one or more variables and assign (bind) a value to each of them.

```
let $a := "foo", $b := "bar"
```

The *window* clause (as the for clause) iterates over its binding sequence and generate a sequence of tuples. However, a tuple now represents a whole *window*, where a window is a sequence of consecutive items draw from the binding sequence. A window may be defined by one to nine bound variable together with a start and end condition. There are two kinds of windows: *trumbling windows* that do not overlap (i.e. an item can occur only once in exactly one window), and *sliding windows* that may overlap (i.e. an item can appear in diffeent windows).

```
for tumbling window $w in (1 to 20)
  start $s at $spos previous $sprev next $snext
    when $s gt 5
  only end $e at $epos previous $eprev next $enext
    when $e - $s eq 2
```

The *where* clause acts as a filter on tuples from previous clauses. It tests the tuples against a constraint (evaluated once for each of them). If a tuple satisfies the condition, it is retained in the output tuple stream; otherwise, it is discarded.

```
where ($i mod 2 = 0) and ($j mod 5 = 0)
```

The *count* clause simply enhances the tuple stream with a new variable that is bound to the original position of each tuple.

```
count $counter
```

The *group-by* clause modifies the tuple stream in which each tuple now represents a group of tuples (from the input stream) that have the same grouping keys. A group-by clause can have multiple grouping keys wich are then applied to the tuple stream from left the right.

```
group by $category, $language
```

The *order-by* clause is used to enforce a certain ordering on the tuple stream. Thus, the input and output streams contain the same tuples, but eventually in a different order. Multiple sort keys (separated by commas) can be specified in either ascending or descending order.

```
order by $prize descending
```

The *return* clause is required at the end of every FLWOR expression and occurs only once. This clause is evaluated once for each input tuple and the results are concatenated to produce an XDM instance, i.e. a sequence of items, which is the final result of the FLWOR expression.

```
return <item>{ $x }</item>
```

To conclude this section, we give an example of how FLWOR expression can be used:

```
1 for $x in (1 to 9)
2 let $m := $x mod 3
3 group by $m
4 return <numbers>{ $x }</numbers>
```

In the above example, the operand `$x` is evaluated nine times, each time the variable `$x` is bound to an integer between 0 and 9. Next, the let clause computes the modulo of base 3 of `$x`. The group by clause then groups the numbers based on their remainder `$m`. Finally, the return clause produces an element for each of these groups.

This gives the following result sequence:

```
<numbers>3 6 9</numbers>
<numbers>1 4 7</numbers>
<numbers>2 5 8</numbers>
```

Again, it must be emphasized that the way FLWOR expressions are evaluated differs greatly from other XQuery expressions. Therefore, one cannot use a single strategy to execute every XQuery expressions in parallel. This is what we would like to develop in the next section.

3.2 The Tree-Hammock Topology

In this section, we present our new Tree-Hammock topology that allow us to evaluate any combination of XQuery expressions in parallel. The Tree-Hammock topology can be divided into two distinct parts, namely a tree pattern and a hammock pattern. Both parts are independent from each other in the sense that they do not follow the same processing model. Nevertheless, they can both appear together inside a query and even be interleaved with each other.

As mentioned in chapter 2, there are two types of parallelism: instruction-level parallelism and data-level parallelism. Instructions can be executed in parallel using techniques such as pipelining, and inter/intra-operation processing, while data partitioning allows the same query to be run simultaneously on different portion of the data. Both levels of parallelism can be exploited by our Tree-Hammock pattern. The tree part of the topology is well suited for computation-intensive queries, whereas the hammock part is well suited for data-intensive queries.

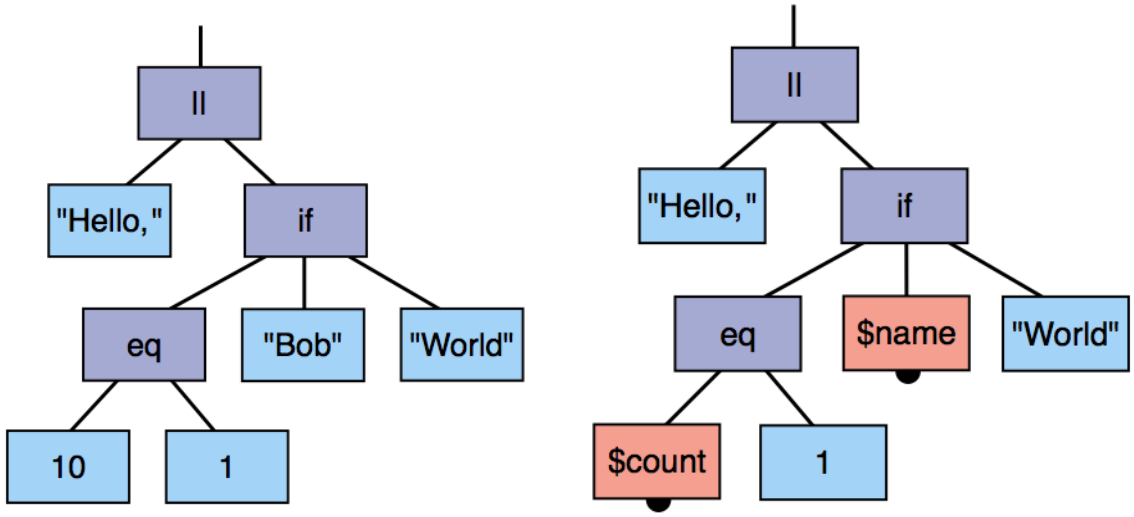
3.2.1 Tree Pattern

Trees are well-known structures in the relational database management systems and query processing literature. They are used to represent query execution plans. In our new topology, we also use trees to represent queries where every XQuery expression is treated as an operator that consumes and produces XDM instances.

Tree of Simple Expressions

Any combination of XQuery expressions forms a query that can be organized in a rooted tree made of simple and/or complex expressions. If a tree is only composed of simple expressions, leaves are limited to simple 0-ary expressions such as literals, while nodes consist of any other simple expression having one or more operands. A simple tree, i.e. tree composed only of simple expressions, is presented in Figure 3.3(a).

In order to evaluate an expression that arise at an upper level in the tree, its operands first need to be evaluated. Therefore, to compute the result of an expression, evaluations of its operands are first triggered. If those operands are leaves, their values are directly passed to the operator without requiring further processing. But if the operands are expressions having their own operands, these (deeper) operands first need to be evaluated recursively prior to



(a) Simple Tree (b) Complex Tree

Figure 3.3: Trees of Expressions

the parent expression. Once an expressions has been evaluated, its result is passed to the parent expression and acts as an operand value. Hence, results of all descendant (or child) expressions first need to be computed before evaluation of their parent expression can take place.

To summarize, evaluation of the root expression propagates down to the leaves by recursively triggering evaluation of intermediate nodes. Results of child expressions are then aggregated by their parent expression and pulled back up to the root expression, which lastly returns the final XDM instance.

Tree of Complex Expressions

In addition to simple expressions, trees can also contain complex expressions such as path or FLWOR expressions. In this case, leaves are no more restricted to simple 0-ary expressions, i.e. they can also contain primitive free-variable expressions. This implies that those free variables must be bound prior to evaluating the tree. Internal nodes can also hold complex expressions. The way a node is evaluated then depends on the complex expression it contains.

Figure 3.3(b) shows a complex tree, i.e. a tree having at least one bindable expression, where two literals from the simple tree were replaced by primitive variables.

Calls on static user-defined functions hold a reference to the functions that are implemented somewhere else in the code. Thus, after evaluating expressions passed as parameters and binding returned values to parameter variables, the function call is expanded as a new (nested) tree that needs to be evaluated in turn. Variables inside leaves are bound to their respective (parameter or non-local) values, before the nested tree is evaluated and result is returned to the callee. If another user-defined function call occurs inside the function body, this process is repeated recursively.

Dynamic function call contains a free-variable expression that is referencing a function rather than a value. Once the variable is bound to a function, the call may be evaluated in two different ways depending on the type of function referenced by the base expression. In the case of a reference on (declared or inline) user-defined function, the evaluation is done as

explained above. However, if the reference points to a built-in function, the static function appears as a black-box operator and is evaluated as such.

Evaluations of filter expression, path expressions, simple map operators, as well as qualified expression have a similar behavior if we consider them as binary operations. Their left-hand sided operand is evaluated only once, while their right-hand sided operand is evaluated many times, i.e. once for each item returned by the left operand. This means that, even though we parallelize these expressions, we will not be able to get the full potential of instruction-level parallelism, since evaluations of the right operand are computed iteratively. We need therefore to consider parallelism at the data level. That is to say, instead of evaluating the right-hand side operand one-by-one for each item, this operand is unfolded and its input sequence is partitioned in order to be evaluated in parallel. Individual results of an unfolded operand need to be fused back (by a multiplexer) once completed.

The same mechanism can be applied to FLWOR expressions. Nevertheless, FLWORs are versatile expressions that follow a much more complicated semantics based on tuple streams. They possess blocking clauses that must be evaluated in-order, on the whole set of items and are thus incompatible with a tree topology.

This shows that most complex expressions do not follow the topology of a tree of instructions unless their right-hand sided operand is unfolded to exploit data-centric parallelism. However, this is not always the case for FLWOR expressions which have a more fine-grained semantics, especially if they contain blocking clauses such as *group-by* or *order-by*. Consequently, since all complex expressions cannot all be expressed following our tree pattern, we need to design an additional topology that can cope with such expressions, namely the *hammock pattern*, presented in section 3.2.2.

Parallelized Tree

Before introducing the hammock topology, we demonstrate how a combination of connected expressions can be mapped on a pool of processes. Processing nodes, or processes, can be subsequently combined to form a processing tree with respect to the order of expressions and independently from the underlying infrastructure.

A first possibility to run a tree in parallel is to assign every individual expression on a single worker or processing node. Each expression is then instantiated by its associated node taking zero or more input sequences and returning exactly one output sequence. A processing node can be seen as a function that transforms its inputs into an output according to the semantics of its mapped expression.

Another option is to give a whole *connected sub-tree* to one processing node instead of a single expression. In this case, the sub-tree is wrapped as a single function passed to the node, which then returns the result of the entire sub-tree. This can be seen as a more coarse-grained method to run parts of a query in parallel. It is worth mentioning that this does not restrict parallelism, but rather suggests another way to organizing the evaluation of expressions. A set of machines or servers could, for instance, have multiple internal processes which in turn could be made of multiple threads. In this scenario, we have several levels of processing nodes, each representing either servers, processes, or threads. Thus, parallelism could still happen inside a single node if it has a finer level of nested processes.

3.2.2 Hammock Pattern

As mentioned in the previous section, the FLWOR expression is a special kind of expression that used tuple streams to pass bound variables and items from one clause to another. If a

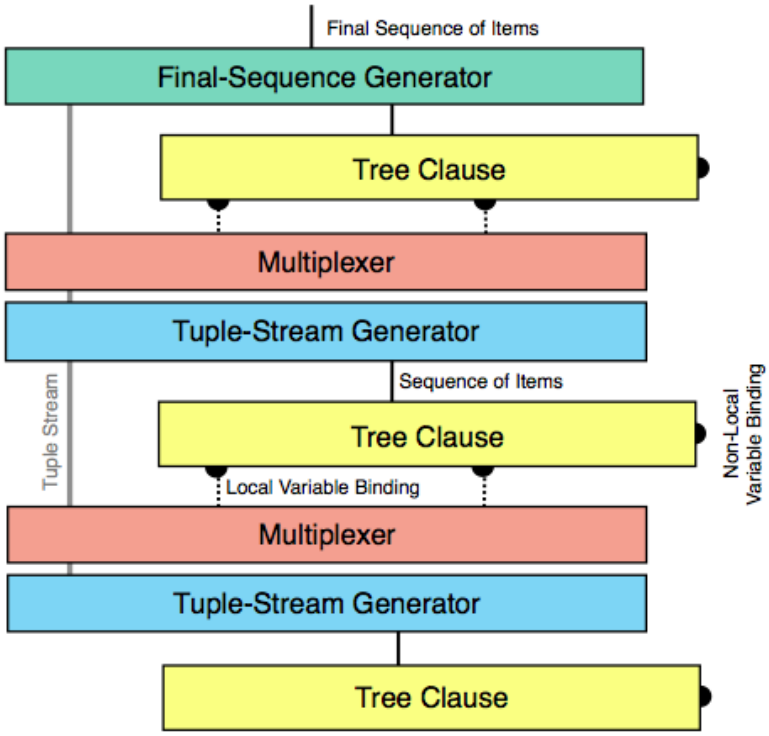


Figure 3.4: Hammock Pattern

query contains a FLWOR expression, the cluster topology differs from that of a tree pattern and looks more like a hammock.

The hammock pattern directly depends on the structure of a FLWOR expression and its number of clauses. It is however possible to generalize the topology as follow: a hammock always begins with an initial clause, followed by a stack of zero or more intermediate clauses, and ends with a final return clause. The first clause produces a tuple stream based on an input sequence of items. The tuple stream then flows down though the stack. Each time a new clause is encountered, the tuple stream is consumed and used to produce an new stream., which is then passed as input to the next clause. The process is repeated until the last clause is encountered. The final clause consumes the given tuple stream and produces a new sequence of items that represents the result of the complete FLWOR expression.

One can clearly see that the evaluation of FLWOR expressions does not follow the topology of a tree and is much similar to a chain of connected pipes: tuples are passed from one clause (pipe) to another. Nevertheless, hammocks share a common property with bindable trees, i.e. they have the same interface. If we consider hammocks as blackbox operators, they can be interpreted as (connected) sub-trees and used as such to form even bigger trees.

As its name suggests, a hammock links trees together. However, its purpose is not only limited to this, and the way a hammock may interleave with other expressions is actually more complex than just joining trees. In fact, every clauses of the hammock pattern can themselves consist of any composition of expressions. Furthermore, hammocks can also be part of a tree, i.e. "attached" to one of its branches.

Before delving into more details of the hammock topology, it must be mentioned that this pattern can also be used for other complex expressions. As shown in section 3.1, some

complex expressions such as path expressions or simple map operators can be reduced to FLWOR expressions and, thus, can directly benefit from our hammock topology.

Hammock Components

Hammocks, as trees, are logical views of data flows. Conceptually, a hammock is not simply composed of FLWOR clauses or, rather, its clauses do more than just evaluating expressions. Clauses also modify tuple streams, handle sequences of items and manipulate bindable expressions. A hammock can therefore be subdivided into different components, each having a predetermined functionality, namely *Clause Tree*, *Tuple-Stream Generator*, *Multiplexer*, and *Final-Sequence Generator*.

Figure 3.4 shows an example of hammock with one initial, one intermediate and one return clause, as well as other hammock components nested between them.

Clause Tree: Each clause tree can be instantiated as a bindable tree, i.e. a tree containing free-variable expressions. Each variable is bound either to its corresponding value contained in each individual tuple, or to a non-local value computed prior to the FLWOR expression. Once evaluated, a clause tree outputs a sequence of items that is sent to the next component, namely the tuple-stream generator.

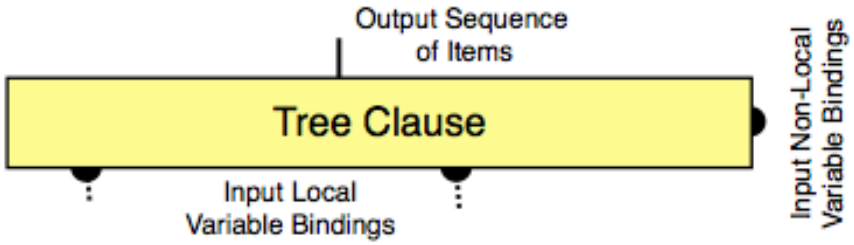


Figure 3.5: Clause Tree

Tuple-Stream Generator: A Tuple-Stream Generator has either one or two inputs depending on the clause preceding it. As its name suggests, it produces a new tuple stream based on a given sequence of items and, if available, an existing tuple stream. If the parent clause is an initial clause, the generator takes only one input, namely a sequence of items. When the evaluation of such single-input generator is triggered, i.e. when its output is pulled, by the next component, only the sequence of items is used to produce the new tuple stream.

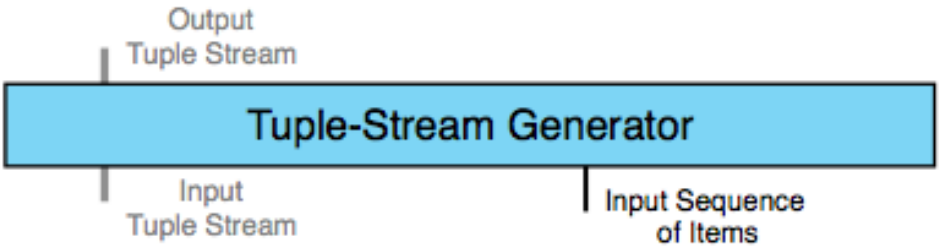


Figure 3.6: Tuple-Stream Generator

For intermediate clauses, however, the generator takes two inputs: a sequence of items from the previous clause and a tuple stream from the parent multiplexer. When evaluated,

this generator pulls both inputs to produce a new stream. Each output tuple of this new steam is created out of the current input tuple, the sequence of items, and according to the clause semantics. For example, applying *let* clause `let $y := 2` on input tuple [`$x=1`] will result in an output tuple [`$x=1, $y=2`]. A Tuple-Stream Generator always outputs a new tuple stream that is then passed to the next component, i.e. a multiplexer.

Multiplexer: A multiplexer takes only one input tuple stream from its upper tuple-stream generator. Its purpose is to read tuples one by one and use their content to bind primitive free-variable expressions contained in the bindable tree of the next clause. When this component is evaluated, it pulls the next tuple and sets the appropriate variables of the following clause tree for later evaluation.

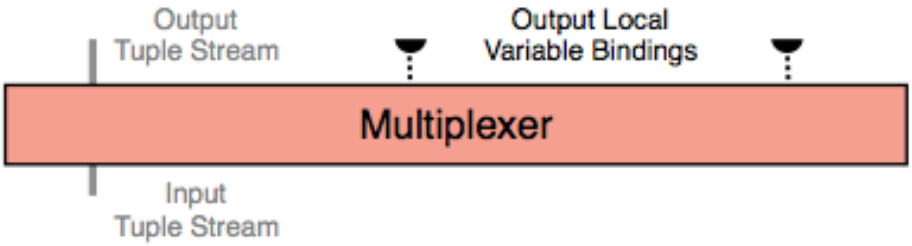


Figure 3.7: Multiplexer

A multiplexer has several outputs connected to the next clause tree. For each local free-variable expression, it produces a sequence of items out of the current tuple. Also, the multiplexer forwards the same unmodified tuple to a lower (tuple-stream or final-sequence) generator. For instance, if tuple `[$x=1, $y=2]` arrives in a multiplexer and that the next clause tree consists of bindable tree `where $x lt 10`, then the multiplexer will have exactly two outputs, one to bind `$x` and another to forwards the tuple.

Note that if the clause tree following a multiplexer does not requires any local variable bindings, the multiplexer simply forwards tuples without further processing.

Final-Sequence Generator: The final-sequence generator has the same inputs as a tuple-stream generator except that it always takes two inputs, i.e. a sequence of item from the final return clause and a tuple stream from the multiplexer preceding the return clause. As its name suggests, this generator aims at creating the last sequence of items that will be output by the FLWOR expression.

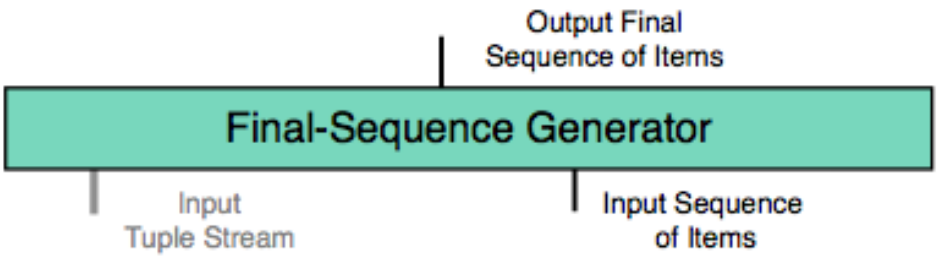


Figure 3.8: Final-Sequence Generator

Once its evaluation is triggered, the final-sequence generator pulls its input multiplexer for the next tuple as well as its return clause tree for a sequence of items. This sequence of items is then processed such that the document order is preserved and that duplicate items

are eliminated. Ultimately, the final-sequence generator outputs an XDM instance which gives the result of the FLWOR expression.

Parallelized Hammock

As explained above, the hammock pattern is close to the structure of a pipeline. Each hammock component can be seen as a separated stage that consumes and produces sequence of items and/or tuples. As a result, *pipelining parallelism* can directly be applied to execute the hammock topology in a time-sliced fashion.

Considering the pool of processing nodes mentioned in section 3.2.1, it is possible to run a hammock pattern in parallel by mapping each component into an individual node and allowing multiple components to overlap in execution. Processing nodes are then connected in series to form a pipe such that the output of one node is the input of another one. In this configuration, processing nodes do not wait for their results to be pulled, but rather forwards them directly to the next component without requiring their adjacent nodes to finish processing. Of course, a node may not be restricted to a single hammock component. It may actually consists of multiple connected components, e.g. a tuple-stream generator together with its multiplexer.

This means that, when applying pipeling parallelism, a FLWOR expression may eventually starts to output results even though it has not yet completely processes its tuple stream. Still, it depends on the clauses contained inside the FLWOR, because some of them require the whole set of (intermediate) tuples to be present before being able to output any results.

Parallelism of the Hammock topology is not only limited to pipelining. We can use intra-operator parallelism to further enhance the above Hammock execution. The idea is now to assign more than one processing node to a single Hammock component such that the internal evaluation of individual component can also be done in parallel.

A possible strategy is to use multiple processing nodes to run individual bindable trees in parallel in the same manner as we parallelize connected sub-trees in section 3.2.1. In other words, we use instruction-level parallelism on trees of expressions contained inside clauses to improve the execution of FLWOR expressions.

A more interesting strategy, which constitutes the essence of our approach, is to logically duplicate clause trees such that each clause being at the same level in the hammock pattern contains the exact same bindable tree. Each duplicated clause is then assigned to one node. Processing nodes mapping the same clause are identical in the sense that they execute the same piece of instructions. In this approach, we use data-level rather than instruction-level parallelism to execute the hammock pattern. Tuples and items are shuffled between processing nodes, such that they will not work on the same portion of data.

The later strategy requires some modifications to our Hammock topology. First, duplicating clauses requires the previous multiplexer to have more outputs: the initial number of outputs (for one single clause) is multiplied by the number of adjacent, identical clauses. For instance, if a clause has M inputs and that the degree of data parallelism for this clause is N, then the total number of multiplexer's outputs is $M * N$. Each time one of the duplicated clauses ask for inputs, the Multiplexer gets the next tuple, meaning that for N duplicated clauses the multiplexer will work M times faster.

Second, since we have duplicated clause trees, each of them require a "partial" tuple-stream or final-sequence generator. We need to manifold all these hammock components such that each clause tree will have its own generator.

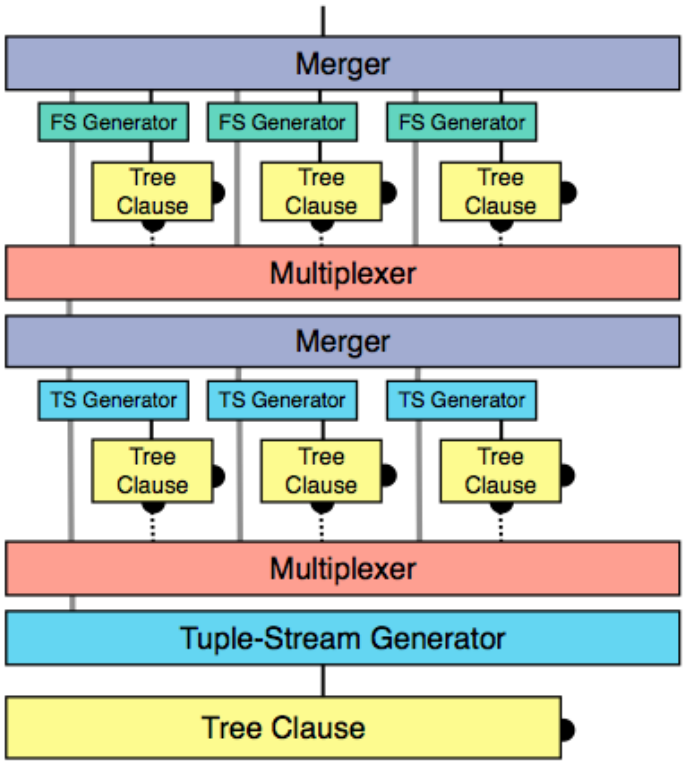


Figure 3.9: Parallelized Hammock

A last but not least modification is the need for *Merger* components after all duplicated generators. Their purpose is mainly to regroup intermediate results output by generators but also to preserve ordering of tuples and items. The number of inputs of a merger component corresponds to the number of duplicated clauses. For N duplicated clauses, the merger will thus have N input tuple streams or sequence of items forwarded by its preceding generators.

Enhancements and Simplifications

The hammock topology as shown above has some bottlenecks, as entire tuple streams are sent between single-purpose components. In this section, we describe several applicable improvements and simplifications that lead together to an alternative design of our hammock pattern.

Tuple-Stream Pre-Processor: A first enhancement is to let tuple streams flow though clause trees instead of forwarding them from multiplexers to generators and vice versa. Consequently, clause tree I/Os need to be modified such that they will allows tuples to pass though them. For instance, inputs and outputs could now consist of $<$*tuple, sequence of items*$>$ pairs rather than simple XDM instances.

Pursuing this modification, we can introduce a local *Tuple-Stream Pre-Processor* component that receives tuples as input from the multiplexer and returns $<$*tuple, sequence of items*$>$ pairs as output to the local Tree Clause. This component endorses part of the role of a multiplexer, namely the task of producing a sequence of items and binding local variable based on the given tuple. As such, the semantics of a tuple-stream pre-processor varies depending on the clause type.

Clause Worker: We can see that all inputs and outputs of hammock components consist of tuple streams except for the clause tree I/Os which also have sequence of items. A possible simplification here is to coalesce local pre-processors and generators together with their respective clause trees. This considerably simplifies the topology because, first, multiple components are merged into a more general structure and, second, the sole communication mechanism is only made of tuples and no more of items.

The new hammock component that results from this unification of components is called *Clause Worker*, because it does the actual job of a clause, i.e. it binds values (contained in the current tuple) to free variables, evaluates the bindable tree and apply the clause semantics to result tuples.

Since the I/Os between hammock components now only consists of tuple streams, the structure of all clause workers must be generalized. This implies that every final-sequence generators contained inside worker clauses dealing with return clauses must be replaced by tuple-stream generators.

Shuffler: Another obvious, straightforward optimization is to combine the multiplexer together with its input merger. This results in a new Hammock element, called *Shuffler*, that takes over the tasks of both components. A shuffler combines intermediate results and finalizes tuples that were pre-processed in parallel by the clause workers (or tuple-stream generator). Depending on the clause semantics, the shuffler either concatenates tuples (*let* and *for* clauses), regroups semi-grouped tuples (*group-by*), or resorts partially sorted tuples (*order-by*). After applying these changes to incoming tuples and generating new ones, this component re-dispatches the tuples to the next pool of clause workers (or tuple-stream pre-processors).

Initial Multiplexer and Final Merger: We can notice that the first clause of a parallelized hammock cannot be duplicated because there is no tuple stream yet. In order for this clause to also benefit from data parallelism, a special kind of multiplexer (with no input) is needed to partition the input data and create the initial tuple stream. This functionality is similar to that of a shuffler except that no input tuple stream needs to be merged. We call this special multiplexer a *Chunker* and use it to distribute the input data of FLWOR expressions over a first pool of clause workers.

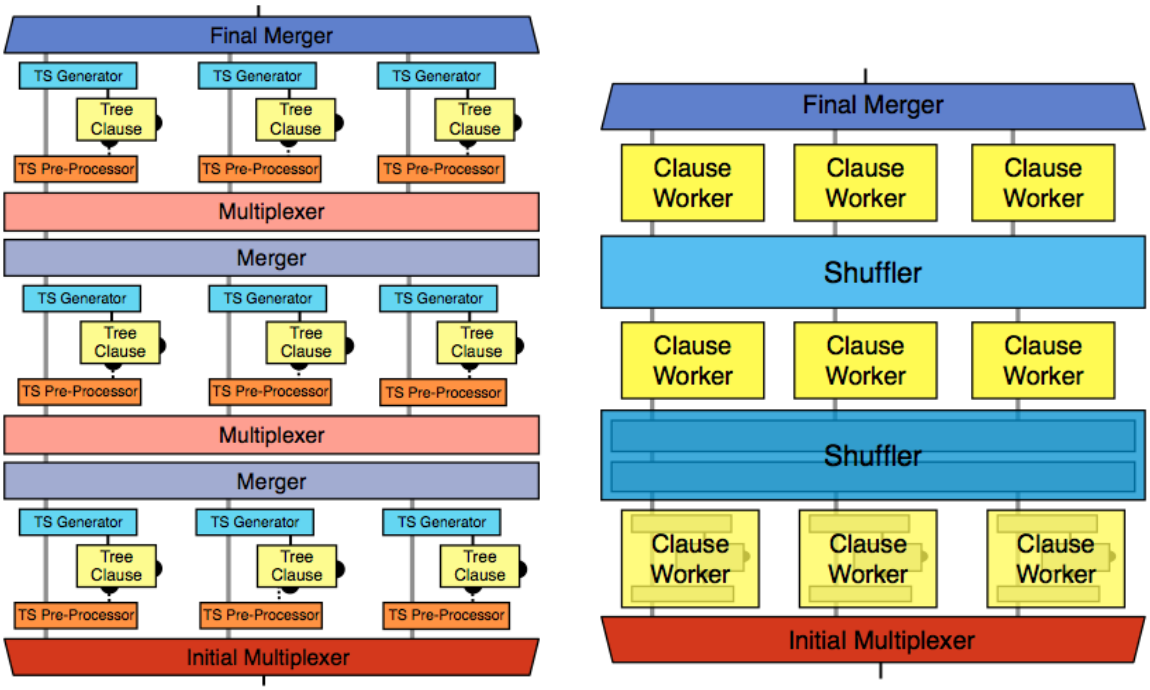
Another special merger is required as well at the end of the hammock pattern. It is needed to combine sequences of items output by the duplicated return clause trees. Furthermore, since the output of clause workers is only made up of tuples, the *Final Merger* is now in charge of transforming the compounded tuple stream into a final sequence of items.

Figure 3.10 depicts two variants of the original hammock topology.

The first variant 3.10(a) represents a enhanced hammock in which (1) the initial clause tree is parallelized thanks to an initial multiplexer, (2) tuple-stream pre-processors are included between multiplexers and tree clauses, (3) tuple-stream generators replace final-sequence generators at the end of the hammock, and (4) a final merger combines the resulting tuple streams.

The second variant 3.10(b) is a simplified hammock that hides most details of the enhanced hammock. The simplification is done by replacing couples of merger and multiplexer by a shuffler and combining clause tress together with their respective tuple-stream pre-processors and generators to form clause workers.

All together these adjustments transform the original hammock pattern into a more applicable pattern that depicted in the remainder of this section.



(a) Enhanced Hammock

(b) Simplified Hammock

Figure 3.10: Variants of the Hammock Pattern

First, the semantics behind this variant is much closer to the initial XQuery specification, which only mentions the existence of tuple streams flowing though clauses. Second, the hammock topology is now very similar to the MapReduce programming model but with some key differences. Both MapReduce and the hammock topology start with a partitioning stage where input data are partitioned among processing nodes, and end with a gathering stage, where partial results are concatenated to form the final result.

However, the MapReduce programming model is composed of only two main steps (namely the *mapping* and *reducing*) separated by one unique shuffling stage, while our approach is composed of multiple processing steps (one for each clause), each being separated by a stage that may perform ordering, grouping and/or redistribution of intermediate results. Another key difference is that communication between MapReduce components is based on <*key, value*> pairs, while communication in the hammock pattern is based on tuple stream or, more precisely, on <*tuple, sequence of items*> pairs.

In section 3.3, we will see in more details how the tree-hammock topology is related to MapReduce and why it has an inherent broader application than the MapReduce programming model.

Non-Blocking Hammock

As mentioned previously, FLWOR expressions may contain blocking clauses that require the input set of tuples to be processed as a whole in order to get the right answer. These clauses, namely group-by and order-by, require a shuffling phase before being able to start evaluating the clause tree and returning some results. But since shuffling is a costly operation during which all tuples are re-partitioned into new buckets, this phase should be avoided if possible.

Simple FLWOR expressions, i.e. FLWORs that do not contain blocking clauses, consist of an unbound number of for, let, and where clauses together with one final return clause. They can directly be evaluated as a pipe in which tuples flow from the initial down to the last clause. Therefore, simple FLWOR expressions do not need shuffling at all and can be

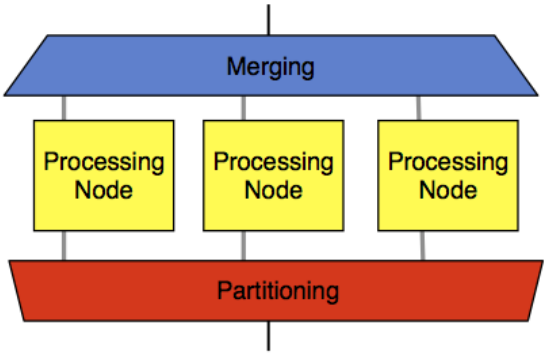


Figure 3.11: Non-Blocking Hammock

evaluated in parallel using a relaxed topology that we called *Non-Blocking Hammock*.

Another category of blocking FLWOR expressions are those which have two or more large data inputs and that perform a join between those *big* data sets. Such expressions require more complex processing with some kind of re-shuffling or tuple shipping based on the join predicate. Therefore, for the sake of simplicity, we consider as "simple" FlWOR expressions all FLWORs that consists of at most one *big for* clause, followed by any number of *small for*, *let*, and *where* clauses, and a final *return* clause. Note that, here, the term "simple" does not correspond to the definition made in section 3.1.

As a result, evaluation of the non-blocking hammock pattern only consists of three distinct phases:

1. Partitioning of the large input dataset into small chunks;
2. Parallel processing of every data chunks by a single layer of worker nodes;
3. Gathering and merging of separated output results.

Figure 3.11 illustrates this relaxed version of our hammock topology. Note that this variant is similar to a simplified hammock shown in Figure 3.10(b) except that we have only one layer of workers and no shuffling component, i.e. every clause workers are regrouped into a single level of processing nodes.

Other (non-simple) FLWOR expressions could also benefit from the non-blocking hammock, as explained below.

Because the hammock pattern is a data-intensive computation that requires an initial partitioning on its input data, intermediate shuffling stages may eventually be avoided if this phase is wisely done and accurate enough. Typically, a hash function applied on specific fields can be used to partition the data. For instance, if a FLWOR contains a group-by clause, hashing can be based on the grouping key(s). In the same manner, for a FLWOR to join two large datasets, hashing should involve the joining keys. The result of this partitioning is a set of input data chunks where (1) each partition is small enough to be run on one single processing node and (2) all data items having the same grouping key or hash value appear in the same partition, such that items between two different partitions are uncorrelated.

FLWOR expressions containing an *order-by* clause will not benefit from a more accurate partitioning, but could still be evaluated on top of a non-blocking hammock as follows: each partition can first be sorted by an individual processing node before the final merger sorts partially-ordered, intermediate results using the external merge-sort algorithm.

To summarize, the hammock pattern as well as its non-blocking counterpart is a data-oriented, divide-and-conquer approach to evaluate FLWOR expressions on large datasets. In general, the Tree-Hammock pattern aims at orchestrating data-level parallelism together with instruction-level parallelism in a more flexible manner while still focusing on data-intensive computing.

3.3 System Architecture

In this chapter, we describes a "scale-out" system architecture used to evaluate queries according to the tree-hammock topology. We start by highlighting the key principles that form the backbone of our architectural design. We also gives a description of every building block together with their respective roles and, finally, explain how they interact with each other.

3.3.1 Design Principles

Current data-centric architectures dealing with large datasets are limited in the kind of queries they can process and lack flexibility to exploit the full potential of parallel query processing.

MapReduce has a powerful yet too restricted processing model to execute interactive queries. Because there are only two processing steps, multiple passes of MapReduce may be required depending on the query. On the other hand, in hierarchical architectures, parallelism is progressively lost during query execution, as processing collides in higher levels of the tree. Each time many outputs of child processes are consumed by their parent the degree of parallelism is decreased, what gives rise to bottlenecks if the combined outputs becomes too large. In the context of Big Data, merging two or more intermediate results often leads to a new *big* dataset.

One idea behind our system architecture is therefore to execute interactive queries with an unbound number of processing steps while keeping a high, constant degree of parallelism.

Gray's third law for Big Data states that computation should be sent at the location where data resides rather than pulling the data on site where the query code lives [25]. Sending large data collections over the network will result in congestion and therefore in poor performance of the system. Thus, we would like to avoid transmission of data between processing nodes by materializing temporary results on site. Materialization, here, does not necessarily imply copying all output tuples into a new physical collection, but rather creating a transient set of references along with some additional data.

The idea is to keep data and intermediate results at their original location, move the computation as close to the data as possible, and only transmit (part of) the final result back to the query issuer.

Our modular system architecture uses a queue-based asynchronous communication mechanism. The main advantage is the decoupling of different system components, allowing them to process independently of one another. Asynchronous models do not only allow better flexibility in distribution and scaling, but also enables load balancing, as well as fault tolerance in both brief disconnections or permanent failures of components [26].

Using an asynchronous architecture results in a more flexible system where query issuers do not have to wait for the entire in-order workflow to complete, and can continue processing even if parts of the final result are not yet available. The workload on those systems can easily be spread out across multiple servers. Load balancing gives the ability to adjust distribution as needed and to accommodate to (abrupt) changes in the work flow of processing resources.

Moreover, this also allows better fault tolerance, since the whole system is not blocked if a single processing node fails. In an asynchronous architecture, jobs can be re-scheduled in case of failure and/or backup processes can be launched in parallel in response to slow processing nodes [21]. In a synchronous system, however, failure of a process causes the callee to wait indefinitely for the process to return.

Finally, we are convinced that an architecture having one centralized instance per query is necessary in order to keep an overview of available computing resources, to benefit from cost-based query optimization and to allow automatic parallelism. Some system architecture designed to process large data sets, like BigQuery [34] and MapReduce [21], are already using such approach.

To summarize, our "scaled-out, data-centric architecture adheres to the following design principles:

1. Focus on data-intensive computing (i.e. data-level parallelism) to process large volumes of data;
2. Bring computations to the data, rather than data to the computation (Jim Gray's third principle) [25];
3. Keep transient intermediate results partitioned among many chunks to avoid bottlenecks and benefit from a high degree of parallelism.
4. Build a modular architecture using asynchronous communication to allow flexible, independent processing between system components.
5. Let a controller, a.k.a. master node, to keep track of resources, handle the whole query plan and enable parallel query optimization.

3.3.2 Overview

The underlying model of our system architecture is based on data-intensive computing and is primarily designed to evaluate the hammock pattern. However, it can actually be used to evaluate queries in accordance with the complete tree-hammock topology, i.e. by separating data-level from instruction-level parallelism while allowing both to interleave.

This separation can typically be done as follows: first, data-intensive computing is used to distribute a query on a cluster of machines (each working on a specific data chunk) and, afterwards, instruction-level parallelism is exploited internally by processes (or threads) running concurrently on each multi-core machine.

In our system architecture, queries are instantiated by a controller on top of a farm of processing nodes, called workers, which are then rearranged to form a dependency graph between jobs and data chunks corresponding to the tree-hammock topology.

The system is composed of the following components:

- *Controller* inspects the user query, prepares the input chunks (one per job), schedules independent jobs and inserts them into a queue.
- *Job Scheduler and Queue* handle asynchronous communication between controllers and workers. The role of the job scheduler is to create and insert jobs inside its queue, as well as retrieve, delete, and update en-queued jobs.

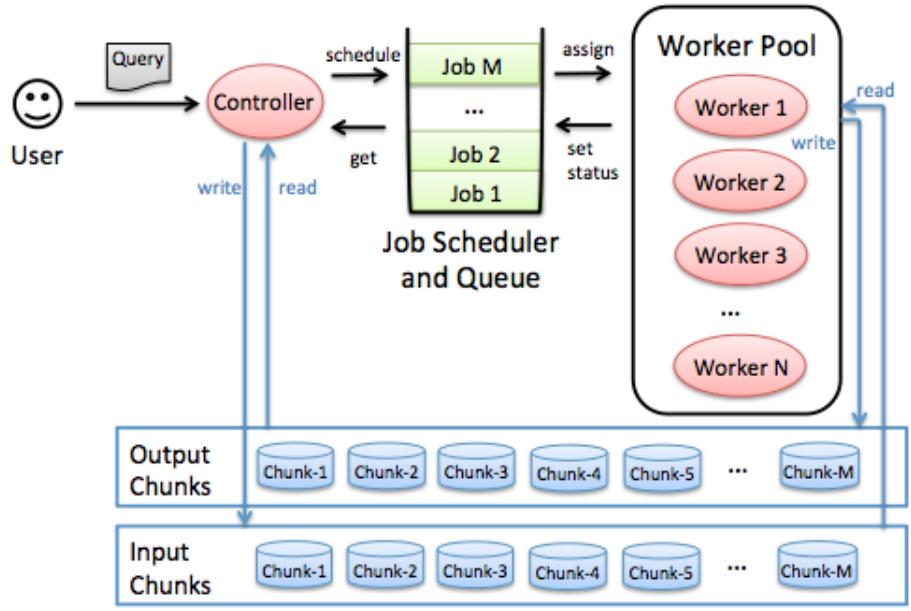


Figure 3.12: System Architecture

- *Workers* are identical processes that wait for jobs to arrive in the queue. Each worker runs one job at a time on a given input chunk, writes its results in an output chunk, and marks the job as completed once done.
- *Partitioned Collection and Data Chunks* are used as communication mechanism to pass data back and forth between controllers and workers. Chunks either contain the partitioned input collection, transient results from workers, or final results.

Figure 3.12 presents these different system components and shows how they interact with each other. In the remainder of this chapter, we explain in more details how the work flow between components takes place and what their specific roles are.

3.3.3 Controller

The controller or master node is the component with which users interacts directly. Its main purpose is to present an interface that abstracts the details and complexity of the back-end architecture and makes easy for a user to execute a piece of code in parallel. The only mandatory information that a user needs to provide to the controller is the query itself and the data on which that query must be run.

In addition to the query and input data, other optional parameters can be passed to the controller via its interface. One can typically configure the controller with a different hash function (useful for FLWORs with *group-by* clauses), maximum number of chunks or workers, and names for input/output chunks.

If the given data collection is not appropriately partitioned, the controller should first hash and distribute input data among many chunks based on the query code. The *big* input collection is then split into several chunks according to a hash function. This is a blocking operation similar to shuffling and must be completed before invoking the worker farm.

There is one controller per query or user application and, as a result, multiple controllers can belong to the same user. Each controller schedules a set of jobs that represents the work

flow to run a specific query in parallel and posts them inside the scheduler queue. Controllers are also responsible to keep track of which jobs belong to each of them. To this purpose, they maintain a list of *tickets* (one per job) that contain both a unique identifier and the status of their respective job.

Once the parallel execution is finished, the controller gathers all results and returns them back to the user. It may also delete completed jobs from the queue if no garbage collector is scheduled.

3.3.4 Job Scheduler and Queue

The Job Scheduler primarily aims at creating jobs on-demand for controllers and persisting them into a collection. This collection of job objects is used as a queue. The scheduler allows jobs to be retrieved and updated by workers, and to be deleted by a garbage collector or controllers themselves.

It is up to the developers to define what the structure of job objects is. Still, we think that a job must at least contain the following information: a *universally unique identifier* (UUID) to distinguish every individual jobs; a *reference to an input chunk* containing data on which the query will be executed; a *reference to an output chunk* containing results returned by the query; a *function name* and body (or reference pointing to the code location) containing the user query; the *job owner*, i.e. controller or user application; the *job status*, e.g. "pending", "in-process", or "completed"; a *timesamp value* used by workers for locking and refreshing purposes.

The timestamp field is periodically modify by a worker to tell others that it is still processing the job. For example, a non-empty timestamp value means that the job has been undertaken by a worker, and an old timestamp value (with a non-completed job status) means that the worker in charge failed before completing the task. More precision about jobs and their structure is given in chapter 5.

We must emphasize that the job scheduler is not an active process that atomically gets a job and assigns it to a worker. Instead, it simply presents an API to system components that is used by workers themselves to poll the queue and look for pending jobs. This avoids the introduction of a single point of failure in the system. It may however consist of one or more processes re-scheduling failed jobs and performing garbage collection of completed jobs.

Finally, since jobs inside the queue can belong to many controllers, the scheduler also needs to deal with security. In fact, it should be denied for a controller to modify of even access jobs belonging to another controller. A general solution to security issues is to have a public-key cryptographic system, where communication is encrypted by the sender using the receiver's public key and decrypted by the receiver using its own secret key. However, independently from encryption, the scheduler still needs to define access and update permissions for controllers and their jobs.

3.3.5 Pool of Workers

A farm of identical processes, a.k.a. workers, is present at the back end of the system. Workers are waiting for jobs to arrive by periodically polling the queue. If a pending job is found, it is fetched by a worker; otherwise, the worker gets back to sleep for a short period of time and retries later on.

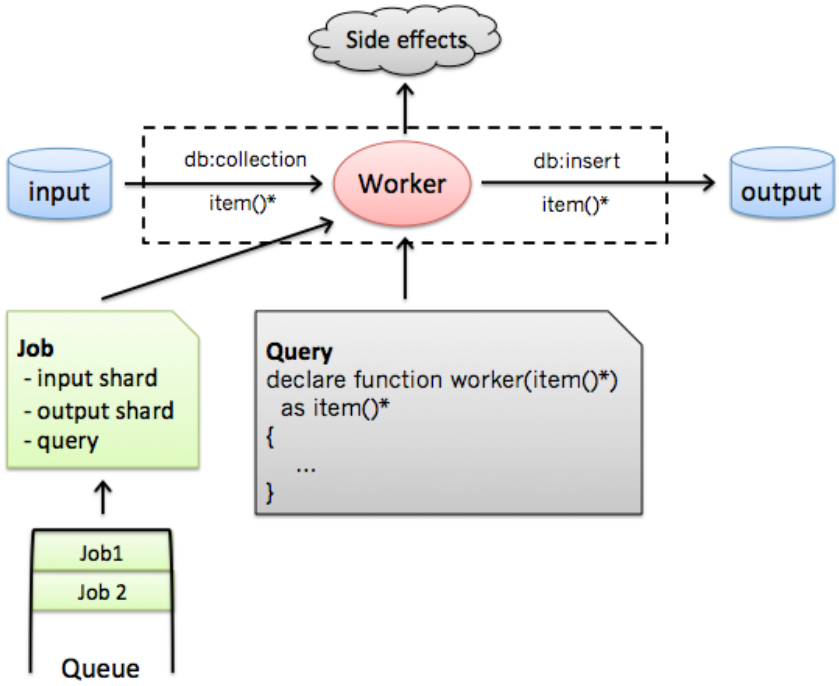


Figure 3.13: Single-Worker View

Workers are independent and self-sufficient (logical) nodes that are organized as a shared-nothing architecture. Because these nodes are both independent and identical, they can be (re-)arranged in any order and following any structural pattern. Typically, we want to organize workers to form a complete Tree-Hammock pattern allowing both data-level and instruction-level parallelism.

Figure 3.13 illustrates the system architecture from the point of view of a single worker. The worker context is always limited to a unique job with two data chunks, plus a function.

The workflow of a worker node is as follows. A worker first gets a job from the queue containing references to an input chunk, an output chunk and a function name. Using the function reference, the worker retrieves the function body from the user program (e.g. from a repository). The function body contains the user query to be executed in parallel. The worker node then runs the code on the data contained in the input chunk, which might cause some side effects, and writes results into the output chunk. Note that two different nodes always work on a different input collection, however, they may write into the same output collection. This allows results to be gathered at one place, e.g. at location where the user application lives. Finally, the worker marks the job as completed and polls the queue again for a new job.

In principle a worker has always a single input chunk, but in special cases (like aggregation queries) it may have multiple inputs. This way, the number of processing nodes stays high and results are only combined when really necessary, i.e. at the end of execution.

A possible analogy is to compare the worker pool as a reservoir full of processing units. The controller endorses the role of a tap that regulates the flux of workers allocated to the current query. Hence, in the same manner as water flows from tap, processing power flows from the cloud.

Cloud computing offers a potentially infinite number of processing resources that can be used to execute any kind of queries over big data sets.

3.3.6 Data Chunks

As explained above, input data chunks must be prepared by the controller prior to execution. This task is similar to a (blocking) shuffling, where the controller hashes and distributes data items of a given, potentially large collection among multiple chunks.

The main reason behind collection chunking is to obtain a high degree of parallelism and allow a maximum number of processes to work concurrently and independently on the same query. In this manner, each worker handles a small portion of data, avoiding single point of contention across the system.

Partitioning of the initial collection is a crucial step on which depends the whole parallel query execution. In this step, some parts of the query must be taken into account to avoid an eventual re-shuffling at a later stage. In particular, data items must be grouped according to (1) group-by keys, (2) join keys and eventually (3) the order-by keys, such that items having the same group-by and/or join value will end up into the same chunk. Thus, no inter-chunk operations shall take place such that no item needs to be shipped between worker nodes. If the number of items is too large for a single chunk, items are partitioned further and aggregated later on. Even sorting can benefits from this initial shuffling step, for example, by using parallel Quicksort (i.e. Partition Sort) [27], where the controller runs a first pass with N pivots to partition items, and workers sort internally their attributed items by applying other passes of this algorithm. Result items are then returned in-order without requiring further sorting among data chunks.

We think that if this initial step is carefully performed, one can avoid re-shuffling at a later stage and allow all workers to run independently without exchanging items until the final results is gathered. This initial shuffling should not be done sequentially by a single controller, i.e. this operation is costly and should also be performed in parallel.

In addition to containing partitioned user data, chunks are also used as communication mechanism not only between a controller and a set of workers but between workers themselves. In the later case, a worker read items from an input chunk, process every items and writes results into an output chunk. Afterwards, this output chunk is either used as a new input chunk by another worker or direcly read by the user. In the former case, this chunk is part of the final result returned to the controller. Once the final result is available, the controller accesses items though all output data chunks and forwards (part of) them back to the query issuer.

It is noteworthy to mention that we use persistent storage to decouple parallel executions in the same way as Hadoop [35] does. The data items are all materialized from collection to collection. Therefore, the final output collection can become large as well, depending on the predicate and number of results filtered out by the query. In a more advanced version, however, pointers or references could be used instead of copying entire items back and forth between collections.

Persisting data items physically on disk allow us to gain in reliability and be more failure tolerant, but this comes with a cost, namely the I/O time to read from and write to stable storage. Moreover, copying entire items as-is from collections to collections will take time, even parallelized. Using references pointing to data items inside the original collection or at least copying only the relevant part of items seem to be the right alternatives. Nevertheless, this configuration is adequate for the time being because, as we will see in the next chapter, items are not forwarded across workers (i.e. results stay on site) and only one pass on the data is required in our current parallel infrastructure.

Chapter 4

Use Cases

In this chapter, we group queries into three different types of use cases, each requiring a different workflow to be processed in parallel:

1. *Simple (Predicate) Query* use cases consist of queries that have only non-blocking clauses. Such query can contain any number of *for*, *let*, *where* and *return* clauses, with the constraint that only one single *for* clause is allowed to have a large dataset as input.

2. *Aggregation and Ordering Query* use cases concern all queries that use *order-by* clauses to sort data items or *group-by* clauses together with aggregation functions.

3. *Join Query* use cases show how a join can happen between two datasets. In particular, we consider the case where *for* clauses can have two large input collections.

We do not consider concrete use cases for the following queries: nested queries, windowing queries, combinations of aggregation, ordering and join, and joins between more than two large datasets. These are out of scope of this report and are left as future work.

4.1 Simple Queries

A simple query can be any XQuery 1.0 FLWOR expression without *order-by*, *group-by*, or *count* clauses. We also restrict *for* clauses to have at most one big input dataset.

The evaluation of these queries follows the shape of a non-blocking hammock. This means that simple queries do not require shuffling of their items and can run on any arbitrary portion of the input data. Hence, only one *for* clause, called *big-for*, is allowed to have a large data input, while all other *small-for* clauses are limited to small data collections or sequences of items.

We differentiate large from small collections as follow: a collection is considered as *big* if it contains billions of data items and, thus, needs to be partitioned and processed in parallel. On the other hand, small collections do not benefit from further partitioning and can be executed locally by a single node without slowing down the entire process.

4.1.1 Small Input

The following first use case does not involve Big Data at all, it is just used as an example to illustrate how simple queries can be run concurrently using our system architecture. It is worth to mention that partitioning a small input collection or sequence of items can result in poor performance because distributed computing comes with an overhead to setup the

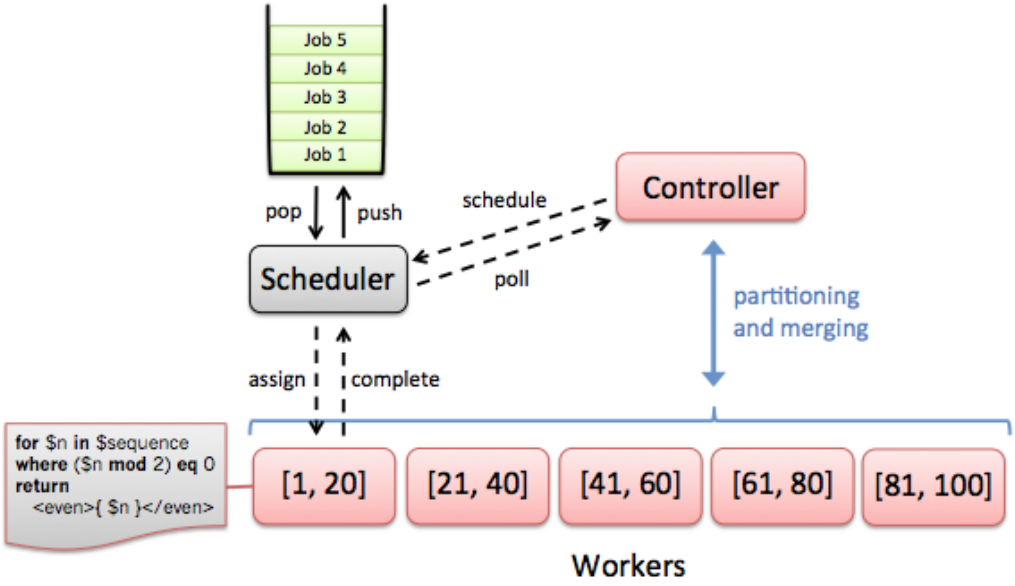


Figure 4.1: Simple Query on a small input

environment. Again, this use case is simply used to understand how our underlying model is working.

The following query takes a small sequence of 100 numbers as input, iterates over this sequence, filters out all odd numbers and finally returns even numbers.

```
1  for $n in (1 to 100)
2  where ($n mod 2) eq 0
3  return <even>{ $n }</even>
```

In order for a user to parallelize this query using our framework, he/she needs to slightly modify his/her application code. First, the query must be wrapped into a closure or function, such that it can be executed on different parts of the input data rather than on the entire collection. Second, the user must import a 28.io module and call the `map()` function in order to initialize the framework and actually run the query in parallel. This is illustrated in the following piece of code:

```
1  import module namespace parallel =
2    "http://www.28msec.com/modules/parallelization";
3
4  declare variable $local:collection := (1 to 100);
5
6  declare function local:query($sequence as item()*) as item()*
7  {
8      for $n in $sequence
9      where ($n mod 2) eq 0
10     return <even>{ $n }</even>
11 };
12
13 declare function local:application()
14 {
15     parallel:map($local:collection, $local:query)
16 };
```

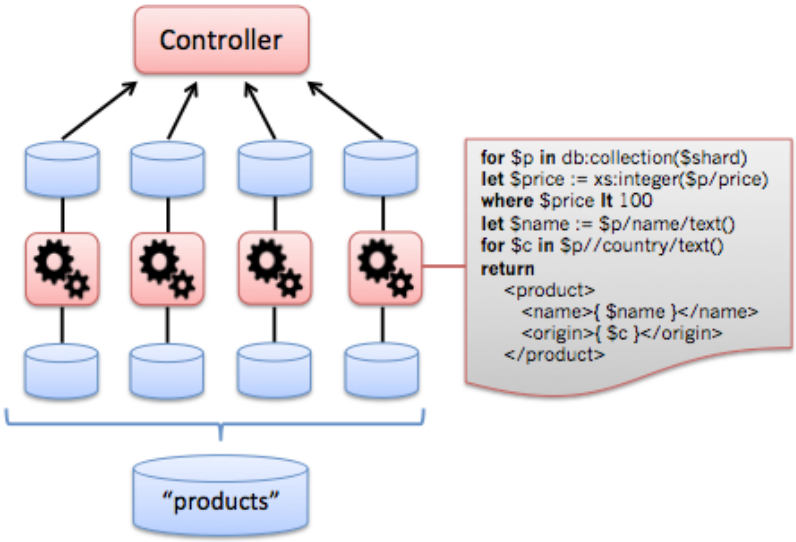


Figure 4.2: Simple Query on Big Data

Once the module is invoked, the query is passed to the controller for parallel execution, which splits the input sequence in five buckets of equal size, each containing 20 items, e.g. [1,20], [21,40], [41,60], [61,80], and [81,100]. Once jobs are scheduled and posted inside the queue, five workers wake up and take over the en-queued jobs. Each worker executes the query for the given portion of data, independently of other workers. Ultimately, items satisfying the predicate are held in five other buckets and wait for controller to retrieve them. This scenario is illustrated in Figure 4.1.

Note that the way numbers are grouped inside buckets does not matter for simple queries, i.e. any other partitioning would have been fine as long as buckets have equal size.

In this use case, the input data is partitioned such that the *for* clause is (partially) unfolded according to its sequence of items. The query is then executed as-is on individual portions of the sequence, without undergoing any further modification.

4.1.2 Large Input Dataset

We have seen in the previous section how a simple query can be instantiated on top of our framework. Now, we consider a simple query that takes as input a large data collection named "products". This collection contains *product* elements which are themselves composed of a *code* attribute, elements *name*, *price* and *category*, as well as a list of *country* elements.

```
1  for $p in db:collection("products")
2  let $price := xs:integer($p/price)
3  where $price lt 100
4  let $name := $p/name/text()
5  for $c in $p//country/text()
6  return <product>
7           <name>{ $name }</name>
8           <origin>{ $c }</origin>
9         </product>
```

This query retrieves all products that have a price lower than 100$ and, for each country of origin, constructs a new XML element containing both the product name and the current country.

To run the above query with our framework, the input collection "products" is partitioned in several chunks. The initial clause is then rewritten as follow: `for $p in db:collection ($chunk)`, where `$chunk` contains a chunk reference or identifier. The query is shipped to workers handling the scheduled jobs. As presented in Figure 4.2, this use case follows the same workflow as the simple query presented in section 4.1.1 except that it takes a large collection as input rather than a sequence of items.

4.2 Blocking Queries

The second category of use cases deals with queries containing *order-by* clauses and with aggregation queries (having *group-by* clauses).

Because these clauses are blocking and require all data items to be present on site, a more complex workflow is needed to execute those queries. In particular, both types of queries need an additional layer of worker nodes to group or (pre-)sort intermediate results.

Even though aggregation and ordering queries have the same issue, namely blocking clauses, they perform two very different operations and, as a result, can be solved using two distinct strategies with the help of our system architecture.

4.2.1 Sorting

Ordering queries are XQuery 1.0 FLWOR expressions that contain at least one order-by clause and, thus, return their results in a certain order.

A strategy to execute *order-by* queries using our architecture is to shuffle intermediate results and regroup them into buckets such that no further inter-bucket sorting is required later on. This can be done by running a first pass of parallel Quicksort (or Partition Sort [27]) using $N - 1$ pivots, where N is the number of chunks. Ordering is therefore enforced at the beginning of the work flow, what ultimately averts the controller to run an external merge-sort on all buckets at the end of query evaluation.

It is noteworthy that shuffling is not an operation to be executed sequentially by a single instance; it is done, as other operations, in parallel with an extra set of workers.

As example, we take the simple query in 4.1.2 and add an *order-by* clause in it:

```
1  for $p in db:collection("products")
2  let $price := xs:integer($p/price)
3  where $price lt 100
4  let $name := $p/name/text()
5  for $c in $p//country/text()
6  order by $name, $c descending
7  return <product>
8           <name>{ $name }</name>
9           <origin>{ $c }</origin>
10        </product>
```

Data items returned by this query are ordered primarily by product names from A to Z and secondly by countries from Z to A.

The two keys inside the *order-by* clause are different not only because of their ascending or descending aspect, but also because they are not applied on the same part of the data. The first ordering key is applied directly to root elements i.e. to the whole data set, while the

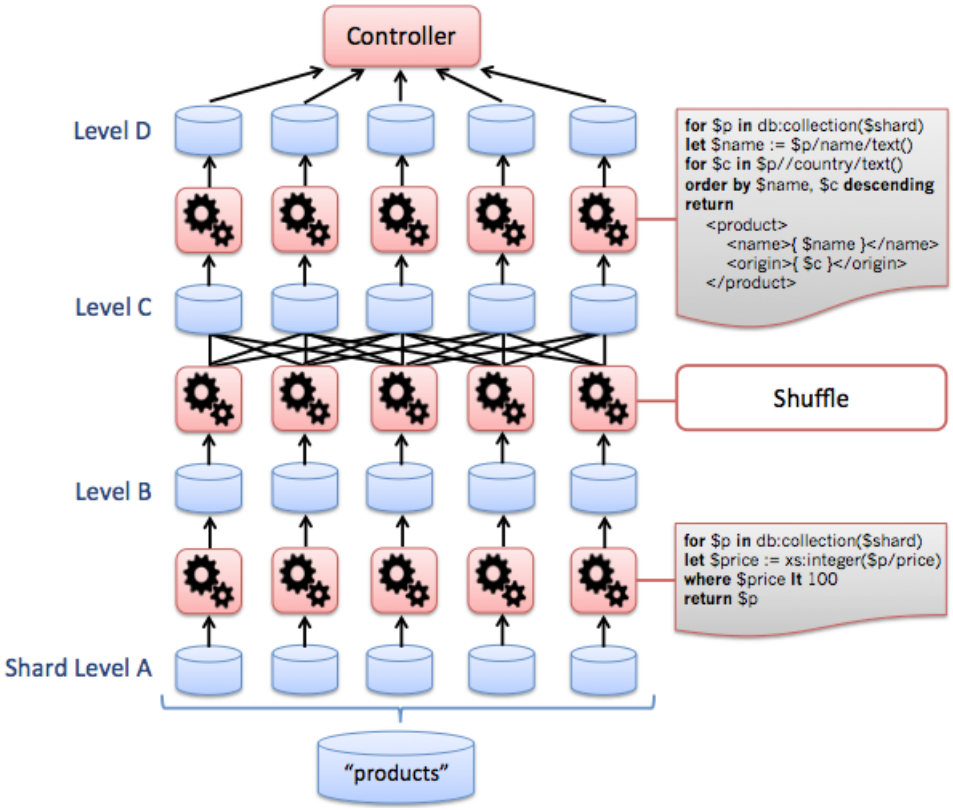


Figure 4.3: Ordering Query with Shuffling

second is only applied to a small sequence of items contained in each *product* element. It is thus not necessary to partition data items based on the later key.

Every query having one or more blocking clauses must be cut in two: the first part contains the query without its blocking clause, i.e. a simple query; the second part contains the actual blocking (order-by and group-by) clauses. The controller is in charge of splitting the query into two distinct sets of jobs, gathering results and eventually applying some post-processing operations.

The query presented above is cut such that its first part contains the top clauses up to the *where* predicate and returns the whole product object. As for the second part, it contains the remaining clauses and returns the constructed *product* elements, as shown in Figure 4.3.

In between these query parts, a shuffle stage takes place. Its role is to pre-sort data items from the big collection by redistributing them into new chunks. After this operation, data items contained in the first chunk should be smaller (according to lexicographical order on product name) than items in the second chunk, which in turn are smaller than items from the third chunk, and so on.

The reason why we split the query in two is to take advantage of filtering (done by the simple query part) and to reduce the total number of items that need to be redistributed. The complete parallel query processing can therefore be done on top of a *simplified hammock*, as figure 4.3 illustrates.

Notice that is is possible to obtain better performance with a more fine-grained decomposition of the original query.

4.2.2 Grouping and Aggregation

Aggregation queries are XQuery 3.0 FLWOR expressions that contain one or more *group-by* clauses and use aggregation functions [30] such as `fn:count`, `fn:sum` or `fn:max`.

Aggregation queries can benefit from our framework in the same manner as ordering queries. However, the workflow at the end of the hammock is slightly different from that of an ordering query.

```
1  for $p in db:collection("products")
2  let $price := xs:integer($p/price)
3  where ($price ge 10) and ($price le 50)
4  let $cat := $p/category
5  group by $cat
6  return
7      element category {
8        attribute count { fn:count($p) },
9        $cat
10     }
```

As use case, we consider the above aggregation query. There are two strategies to evaluate aggregation queries, both including special workers at the end of the hammock. These special workers take two (or more) chunks as input and combine pre-aggregated items together to produce the final result. The controller eventually performs a last pass of aggregation if necessary.

A first strategy is to use an earlier shuffle stage as for ordering queries. In this case, we assume that there are five categories in the input data, e.g. A, B, C, D, and E, such that products belonging to category A are hashed into a single bucket, products of category B are scattered between two buckets, categories C and D end up in the same bucket, and finally categories E and F are spread between three buckets, where two contain either category E or F and one has both categories. At the end of the hammock, one additional layer of workers aggregate intermediate results further. The two last steps of this approach are depicted in Figure 4.4.

Another strategy is to avoid shuffling and apply aggregation directly on existing data chunks. In this scenario, several layer of aggregating workers would be needed at the end of the hammock. This model is similar to the architecture of aggregation frameworks [43] and Google BigQuery [34].

Even though we do not consider concrete use cases that involve both aggregation and ordering, one can clearly see that such queries could directly be instantiated on a hammock based on the former strategy (with shuffling). This hammock will then require at least two special layers of workers, one to shuffle data items and another to aggregate transient results.

4.3 Join Queries

Up to now we only considered use cases on a single large data set. At the end of this section, we present a query that takes two big collections as input, but first we introduce a simpler join query between one relatively small input and one large data set.

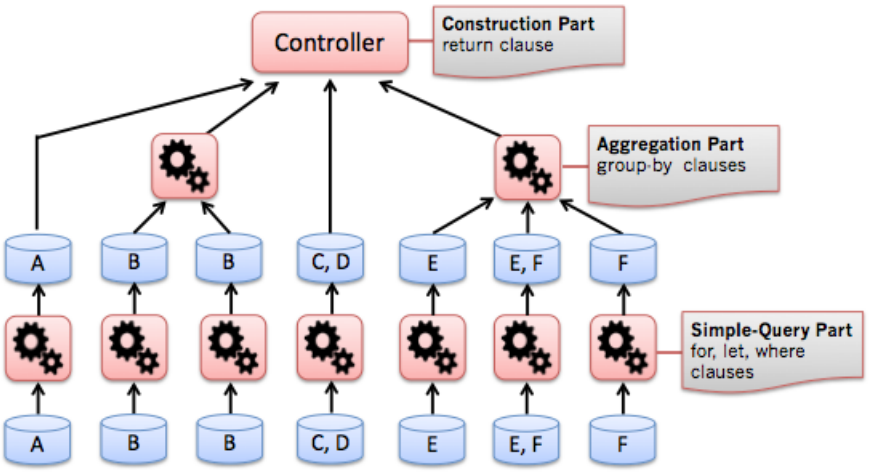


Figure 4.4: Aggregation Query

4.3.1 Small Join

This use case is used to illustrate how a join query with only one large input is handled by our framework. The work flow for this query follows a process similar to Google BigQuery [34], in which the small input collection is copied and broadcast to every worker taking part in the parallel query execution. The key difference with BigQuery in our case is that the small collection is only forwarded to workers that actually perform the join over the two data sets.

Considering the following query, its work flow is sketched in Figure 4.5:

```
1  for $p in db:collection("products")
2  let $p-cat := $p/category/text()
3  where $p//country/text() = "Italy"
4
5  for $c in db:collection("categories")
6  let $c-name := $c/name/text()
7  where $p-cat eq $c-name
8  return
9      <product>{ $p/name, $c/parent }</product>
```

We assume that the collection called "categories" consists of few records while the "products" collection contains billions of records. The "categories" collection contains *category* elements that are made of *name*, *parent* and *description* elements. The query joins both collections based on category names and outputs the names of all products that come from Italy together with their root or parent category.

In this use case, only one layer of worker nodes is required to join both data sets and execute the query, because the whole "categories" collection can be copied and sent to workers as long as it stays reasonably small. Moreover, no shuffling is required prior to the join, since all categories are present inside the small collection and, therefore, no join result can be missed.

However, as this collection becomes larger, one may not be able to entirely copy and send its content to all workers. As described in the next section, an additional shuffling step will then be needed to relax the join operation.

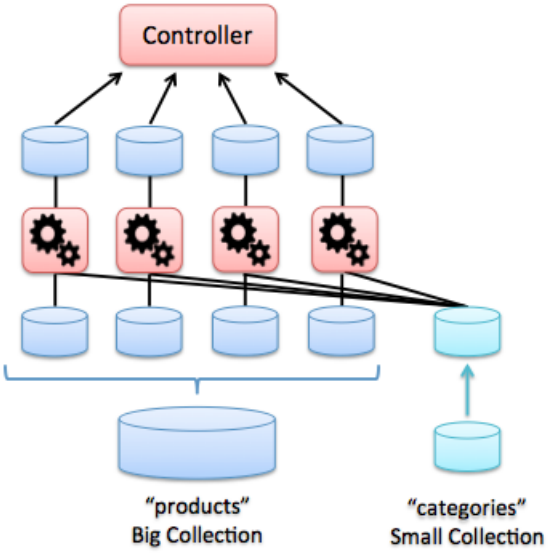


Figure 4.5: Join involving a small collections

4.3.2 Big Join

Joining two big collections is much more cumbersome than joining two collections where one of them is small. For joins having two *big for* clauses, i.e. *for* clauses taking a large dataset as input, both collections must be partitioned based on the joining key(s). The idea is to build an index on-the-fly that can then be exploited to evaluate join queries.

The following query associates two data sets, "products" and customers"". The "products" collection is the same as used so far, while the latter contains customers together with their purchases. Each customer is a document that consists of the following XML nodes: a *id* attribute, *first-name*, *last-name* and *nationality* elements, as well as a list of *purchase* elements that are themselves made of *product-code*, *count* and *purchased-date* elements.

The query outputs a list of orders that involve products with a price higher than 1000$ purchased by Swiss customers.

```
1  (: Big Products Collection:)
2  for $p in db:collection("products")
3  let $p-price := xs:integer($p/price)
4  where $p-price gt 1000
5
6  (: Big Customers Collection :)
7  for $c in db:collection("customers")
8  let $c-nation := $c/nationality/text()
9  where $c-nation eq "Swiss"
10
11 (: Join Predicate :)
12 let $p-code := data($p/@code)
13 let $c-codes := $c//purchase/product-code/text()
14 where $p-code = $c-codes
15
16 return
17   element purchase {
18       element customer {
19           attribute id { $c/@id },
```

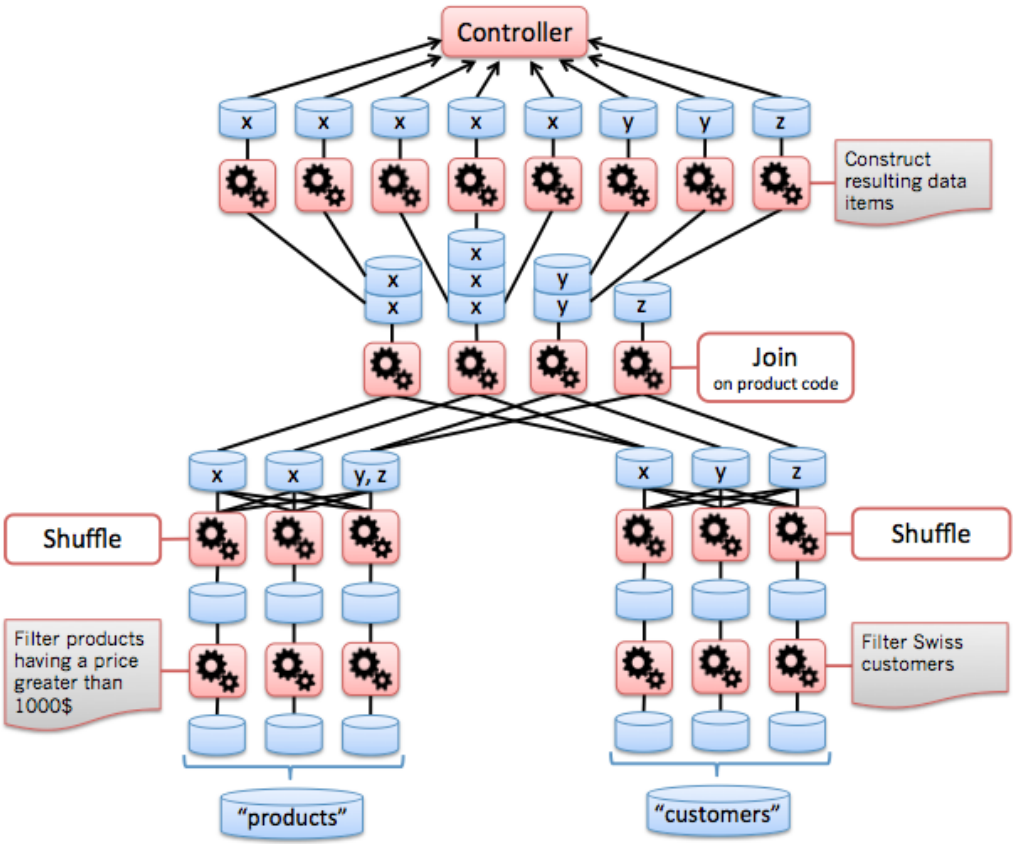


Figure 4.6: Join between two large collections (Big Join)

```
20              $c/first-name/text() || " " || $c/last-name/text()
21          },
22          element product {
23              attribute code { $p-code },
24              $p/name/text()
25          }
26      }
```

In the same manner as ordering queries are split into two parts, join queries are cut in three. The two first parts apply either to the "products" or "customers" collection and, hence, can be pushed down to the hammock in the same manner as relational predicates are pushed down when optimizing query execution plans. The first predicate query retrieves products having a price greater than 1000$ while the second filters out customers that are not Swiss. They are run concurrently using two independent sets of workers, one per collection.

After pre-filtering every data chunk, two other independent sets of workers perform a shuffle stage on each collection. In this step, a hash function maps data items to chunks according to the join key, i.e. product code. We assume that there are three join values, e.g. x, y and z, as shown in Figure 4.6. The hash function guarantees that items having equal product code are shipped to the same data chunk. Ideally, there should be exactly one chunk per join value but, in practice, it is common to have several keys mapped into one chunk (i.e. collisions) or several chunks per key if a single bucket is too small to contain all hashed items.

Once shuffling is done, the join operation is performed by another set of special worker nodes. The worker context, here, is limited to two input chunks that can potentially produce join results. This means that (1) those chunks must share the same hash value and (2) one chunk must contain *product* items while the other must hold *customer* elements. As

a result, the *big join* operation is unfolded into many (independent) joins running on small portions of the data. The execution time is reduced, from quadratic in collection sizes (M and N), to linear in the number of chunk pairs to be joined K and quadratic in chunk sizes S: $M * N \geq K * S^2$, where in the worst case $K = \frac{M}{S} * \frac{N}{S}$.

Joining two data sets can result in quadratic growth of intermediate results, such that the output of a joining worker does not fit in a chunk chunk. If further processing is required on resulting joined items, the upper layer of workers will then be proportional to the number of chunks produced by the *big join*, as in Figure 4.6. Again, keeping the number of chunks and workers high allows us to make the most of our shared-nothing architecture. Here, this last set of worker nodes simply construct the *purchase* elements.

The approach presented so far for joining two big collections is very similar to the *Grace Hash Join* proposed in [13], apart from the fact that computation and data are distributed. The Grace Hash join partitions two relations R and S using the same hash function into buckets R_1 to R_m and S_1 to S_m respectively, such that tuples in R_k can only join with tuples in S_k. In other words, this algorithm emulates an index on-the-fly by dividing the whole join operation into a set of partial joins, in the same way as we resolve the distributed *big join*.

Chapter 5

Implementation

This chapter presents the implementation of the system architecture shown in the previous chapter. Th chapter starts with a short description of the underlying technologies used to develop our parallel infrastructure. Afterwards, we briefly state the goals and limits of our system, present its structure and explain how the different components interact with each other. In the remaining sections of this chapter, we describe each system component in details.

5.1 Technologies

The major part of our system was developed using the 28msec's platform (28.io) [36]. 28.io is built on top of the query and script processor called Zorba [38] and uses the MongoDB database [42] as stable storage layer.

Almost all components of our architecture were implemented using JSONiq [40], an extended version of XQuery, that brings native support for JSON arrays and objects. The only system component that is not implemented in XQuery is the daemon process, presented in section 5.5.

28.io

28.io is a platform that is used to build and run web-based applications. This platform offers an integrated application server and database solution for data intensive applications. It uses Zorba as XQuery processor and enables data persistence with MongoDB.

In 28.io, XQuery is used as language for writing the application code, defining the collections and indexes, as well as accessing and modifying data. XQuery is therefore used as both a scripting language to code the business logic and as query language to access and update data. In other words, 28.io enables the use of a single programming language on all tiers and, thus, collapses web servers, application servers, and databases into a single programming stack.

Every 28.io project is identified by a logical URI. Presented in Figure 5.1, the typical structure of a project consists of a set of pre-defined directories that contain the application code, XML Schema and static files. The most important parts of a 28.io project are the handler and library modules that contain the XQuery code. Modules in the *handlers* folder are exposed and can be invoked by other applications or end-users. Modules inside the *lib* directory can only be imported by modules from the same project. For more information about 28.io's project structure, the reader can refer to [36].

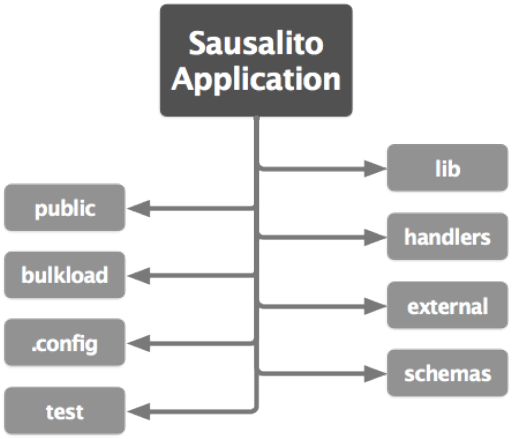


Figure 5.1: 28.io's Project Structure

Zorba

Zorba is an open source query processor written in C++ which implements several W3C XQuery and XML specifications. It provides a high-level query language, namely JSONiq [40], to process flexible data like XML and JSON. JSONiq is an extension that brings JSON support to XQuery.

By default, Zorba is built with a main memory store, meaning that it does not provide persistent data storage. All data collections and indexes are created and stay in memory. Zorba allows stores to be plugged to enables persistence of data. For instance, 28.io uses MongoDB under the hood as a data storage for Zorba.

MongoDB

MongoDB is an NoSQL, document-oriented database system developed by 10gen [42] that stores data in a JSON-like binary-encoded format called BSON [45]. It is an open-source project as well and is also implemented in C++.

MongoDB is a scalable data storage that can run over multiple servers. The database system balances the load and replicates data to keep the system up running even with the presence of hardware failures.

MongoDB uses *sharding* [44] to distribute data and spread a single logical database across a cluster of machines. Sharding uses range-based portioning to divide the database into data chunks, called *shards*, each containing a small portion of XML or JSON documents. MongoDB defines a shard has been a master node responsible for a certain portion of the data, together with one or more slave nodes that store a copy of the data contained in the master node. We will make active use of MongoDB shards to parallelize queries as stated in section 5.5.

5.2 Infrastructure Development

This section presents the structure of our parallel infrastructure and the goals and boundaries of our parallel infrastructure.

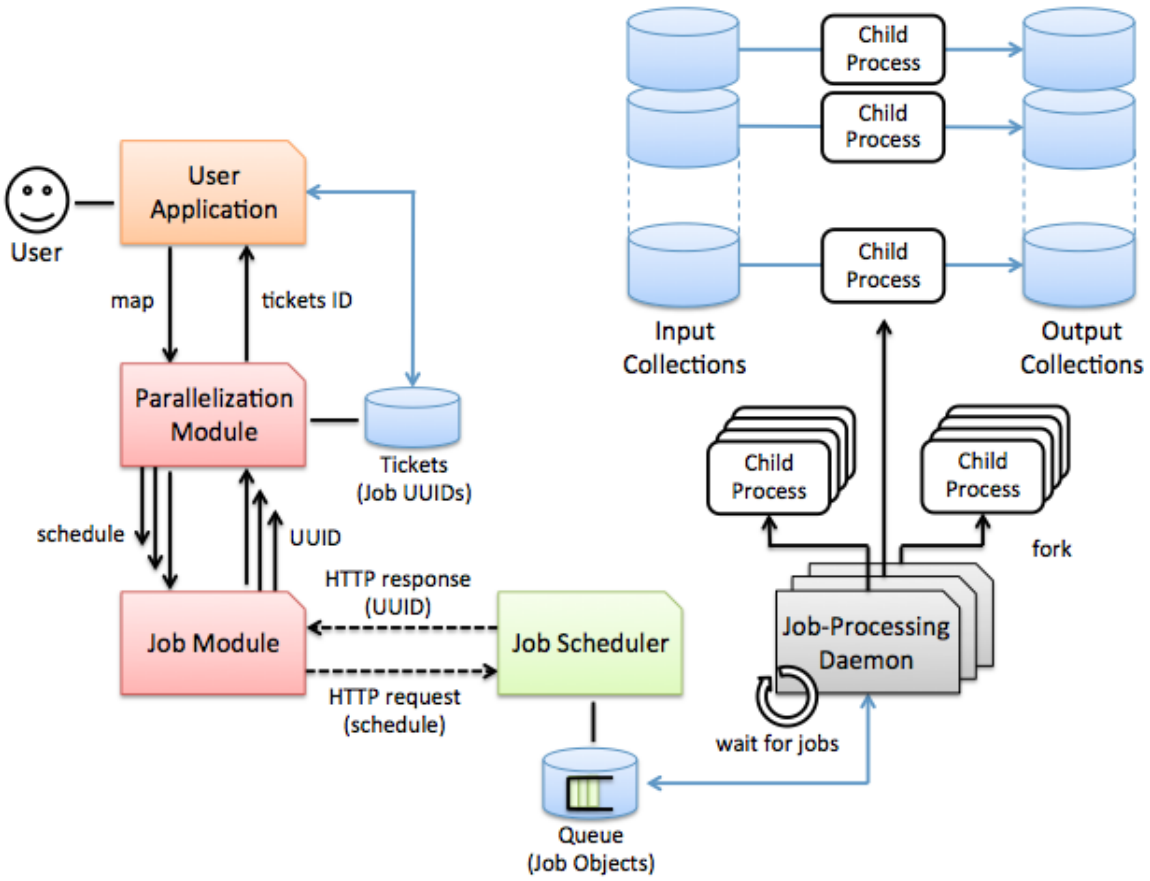


Figure 5.2: Infrastructure

5.2.1 Structure

Prior to this thesis, there was no subsystem in 28.io that would have enabled asynchronous communication or query parallelism. Since neither of these features were originally present, we had to implement both from scratch. For this reason, we divided the infrastructure development into two steps: first, we programmed a queuing system to manage asynchronous communication in 28.io and, second, we used this new system to execute parallel queries.

The development of the asynchronous architecture is described in section 5.3. Literally, this queuing system consists of the job scheduler presented in section 3.3.4. As for the parallel query processing infrastructure, including 28.io modules and the job-processing daemon, its implementation is outlined in the two last sections of this chapter.

The complete infrastructure, illustrated in Figure 5.2, is composed of four main system components, namely the *Parallelization Module*, *Job Module*, *Scheduler*, and *Daemon*, which interact with each other as follows.

Once a user application calls the parallelization module, the job module is invoked multiple times to schedule a set of jobs representing the parallel query. For every invocation, the job module receives a set of parameters representing a job. There are four possible parameters: the function EQName wrapping the user query, an input collection name containing the data, an input chunk representing the portion of data that the query must process, and an (optional) output collection where results are written. Notice that the input chunk is extracted from the given input collection thanks to another module, called *Store Module*, that is not shown in Figure 5.2. Once these parameters are known, the job module sends an HTTP request

	Infrastructure	Architecture	Description
1	Parallelization Module	Controller	This module presents a simple interface to users that abstracts and hides the complexity of the back-end infrastructure. It allows users to easily parallelize their queries and to retrieve query status and results. The module also automatically partitions and hashes input items if necessary.
2	Job Module	Controller, Scheduler	Using this module, users or other applications can schedule one job at a time, as well as retrieve the status and results of a specific job. Jobs can have three different kinds of input, namely collection, data chunk or sequence of items.
3	Asynchronous Queuing System	Scheduler	This module is responsible for asynchronous communication between any applications or other system components. It enables scheduling, retrieval, update, and deletion of job objects, and implements security and user permissions.
4	Job-Processing Daemon	Worker	As its name suggests, a daemon processes pending jobs from queue (one at a time). It creates a parallel process that executes the given query on a small portion of data and writes results into another output collection.

Table 5.1: Description of Infrastructure Components

containing this information to the job scheduler. The scheduler then extracts the parameter values from the message body and use them to create the actual job object. Before pushing the job inside a materialized queue, the scheduler adds the creation time and sets the job status to "pending". In the background, multiple daemons (one per machine) are periodically polling the scheduler queue for new jobs. Once a daemon finds a pending job, it atomically fetches the job and creates a new internal process that will handle the job. This process executes the user query on the appropriate data and writes results to the given output collection.

Table 5.1 shows the separation of concerns between the different system components and compare them with their architectural counterparts. The table also provide a short description of their roles. A more precise description of these components is given in the remaining sections of this chapter.

5.2.2 Goals and Limitations

We decided to slice the implementation as explained above in order to (1) quickly produce a first version of the complete infrastructure that can solve a subset of FLWOR expressions, (2) get to a first release with only minor changes to Zorba and 28.io, (3) obtain validation of the approach, and (4) gather experience before starting to change the FLWOR runtime and compilation in Zorba. All in all, the idea behind this implementation is primarily to get the bigger picture regarding our approach and to have a first parallel system in production.

Project Boundaries

The *Tree-Hammock* topology is a holistic approach to solve the problem of distributing and parallelizing queries in general. More specifically, this technique enables both distributed

data access and parallel execution of all kinds of XQuery expressions, as explained in chapter 3. Unfortunately, implementing the entire topological model on an infrastructure developed from scratch requires a very long effort that would have exceeded the available time for this thesis.

Data-level parallelism represents the key challenge when querying Big Data in the cloud. The first version of our parallelization infrastructure is therefore focused on data-intensive computing, i.e. on the execution over hammock patterns. Moreover, in the framework of this thesis, we decided to restrict the hammock topology further by narrowing the set of applicable use cases to *simple queries*. Remember that simple queries consist of any number of *for*, *let*, and *where* clause and taking only one large collection as input.

Hence, we aim at a simple parallel infrastructure that should solves the problem of distributing predicate queries on top of the *Non-Blocking Hammock* pattern, introduced in section 3.2.2.

Long-Term Development

All design choices made for our first system were taken to guarantee a seamless evolution of our infrastructure in order to support all kinds of FLWOR expressions in future releases.

Ideally, we would like to be able to execute any XQuery code, including Scripting [32] and Update Facility [31]. We are convinced that the architecture described so far could be used to execute in parallel any XQuery 3.0 and Update Facility queries. However, since scripting has side effects, running XQuery Scripting expressions would require further investigation. More details about future enhancements of our system are given in chapter 7.

5.3 Asynchronous Queuing System

In order to run parallel queries, we first needed a mechanism that would allow us to schedule jobs asynchronously. We therefore started the development of our infrastructure by implementing a queuing system, namely the *Job Scheduler*. The scheduler is a 28.io project that has been integrated into 28msec's Web portal such that it can be seen by every user application and 28.io component. A system instance can address a request to the scheduler by using its base URI.

The role of the job scheduler is basically to handle job objects inside a queue. Its functionality can be summarized as follows: by addressing HTTP requests to the scheduler, it is possible to create, retrieve and delete jobs as well as to retrieve and update job properties.

5.3.1 RESTful API

The job scheduler should present a uniform interface to allow clients to easily use its services.

Since REST [28] is a predominant model to design Web services, we decided to implement the scheduler API as such. REST defines a clear way to access and manage resources stored inside a collection. The idea behind a RESTful Web API is that individual resources contained in the collection are referenced using URIs which act as identifiers. Resource representations are exchanged between system components and the scheduler using HTTP. In our case, we use JSON as media type for these representations. Also, HTTP methods are utilized to defined the set of operations supported by the scheduler.

The scheduler collection contains only job objects and, hence, all data items inside the queue have the same structure. Every object is identified by a UUID and contains several properties such as the job's creation time or status. The UUID is therefore used inside URIs to identify a particular job. And now, since the URI points to a unique job object, sending a request to this URI will return a JSON representation of that job. Job properties can be accessed as well if the property name is appended to the URI, just after the job UUID as shown below.

```
/scheduler/jobs
/scheduler/jobs/<uuid>
/scheduler/jobs/<uuid>/<property>
```

The above relative URIs represents the three possible types of URIs that can be called on the scheduler. Invoking the scheduler with any other URI structure will result in a 404 "Not Found" HTTP response.

At the scheduler, the URI is segmented to extract parameters contained in the requested path. For example, if the URI points to a specific job property, two parameters are extracted, namely the job UUID and property name. Afterwards, the scheduler inspects the HTTP method used in the request to trigger the appropriate logic.

The scheduler only implements four methods: GET, POST, PUT and DELETE. GET is used to retrieve resources (jobs) from the collection, POST inserts new jobs in the queue, PUT is used to modify some job properties (such as "accessed"), and DELETE removes completed jobs from the queue. If a client used another methods, e.g. TRACE, the scheduler returns status code 405 "Method Not Allowed".

Table 5.2 shows the actions performed by the scheduler, as well as the data and status code returned to clients for each combination of implemented HTTP method and valid URI. The GET method can be called with any valid URI, i.e. on the entire collection, individual job or job property. The status code returned by a GET method is 200 "OK" for any non-empty response, 204 "No Content" if the collection is empty, or 404 "Not Found" if the URI points to a non-existing resource. The POST method can only be invoked with the base URI, directly on the collection. If successful, a job is inserted in the queue and status code 201 "Created" is returned. A POST request on an individual resource result in a 405 "Method Not Allowed". PUT is only used on job property and returns either 204 "No Content" to tell that the update was done successfully or 404 "Not Found" if no resource exists. Other URIs with PUT method are not allowed. Finally, DELETE can be invoked on both the collection and individual jobs, in both cases a 204 "No Content" is sent back; otherwise, if the request points to a nonexistent job, 404 "Not Found is returned". Deleting a job property is not allowed since it will put the resource in an inconsistent state. Again, in this case, a 405 "Method Not Allowed" response is sent back to the client.

5.3.2 Job Objects

Job objects are stored in JSON format inside a materialized collection that represents the queue. A real example of job object is shown in Appendix B.1. We gives here a slightly simplified job structure to illustrate its key fields:

```
{
    "uuid": "dfa42886-0a4f-49c6-91ce-5959d1059a60",
    "input": {
      "collection": "products",
```

	GET	POST	PUT	DELETE
Queue Collection URI /jobs	Returns an array of job URIs (*JSON*) 200 "OK" 204 "No Content", if result is empty.	Creates a new job and returns new UUID (*JSON*) 201 "Created"	Not used. (Replaces the entire collection by another) 405 "Method Not Allowed"	Deletes all jobs contained in collection, if permission granted 204 "No Content"
Job Resource URI /jobs/<uuid>	Retrieves a job based on its identifier (*JSON*) 200 "OK" 404 "Not Found", if job does not exist.	Not used. (Treats URI as collection) 405 "Method Not Allowed"	Not used. (Replaces a whole job by another) 405 "Method Not Allowed"	Deletes the specified job, if allowed 204 "No Content" 404 "Not Found", if job does not exist.
Property Resource URI /jobs/<uuid>/<property>	Retrieves a property value (*JSON*) 200 "OK" 404 "Not Found", if job/field not exist.	Not used. (Treats URI as collection) 405 "Method Not Allowed"	Updates a property value (used for simulation only) 204 "No Content" 404 "Not Found", if job or field does not exist.	Not used. (Deletes the property) "405 Method Not Allowed"

Table 5.2: Scheduler's RESTful API

```
      "chunk": {
          "id": "5139f6722abe5bf9a110a88d"
      }
    },
    "output": {
      "collection": "results"
    },
    "application" : {
        "name": "my-application"
    },
    "function" : {
        "name": "Q{http://www.28msec.com/repository/}user-query",
    },
    "properties" : {
        "created": "2013-01-14T14:31:31.32",
        "accessed": "2013-01-14T14:32:03.42",
        "lock": "daemon-1",
        "lock-acquired": "2013-01-14T14:31:54.92",
        "status": "in-progress",
    }
}
```

Job objects play a central role in our parallelization framework, as they carry relevant information needed to run a query in parallel. Besides, in addition to purpose-specific data, a job contains fields that are required to implement asynchronous communication. UUID, function, application name as well as job properties are mandatory for proper functioning of

the asynchronous queuing system. These properties consist of the following fields:

- *created*: automatically set by the scheduler, it stores the time at which the job was created.
- *accessed*: initially empty (null), contains a timestamp value that is either updated when the job is accessed by a daemon or periodically modified by beacon messages if the query is currently being processed.
- *lock*: initially empty, it contains the daemon name which is handling the job.
- *lock-acquired*: initially empty, encloses the time at which the daemon acquired the lock on the current job object.
- *status*: indicates the job status; a job can have three normal states, "pending", "in-progress" and "completed", as well as two error states, "timed-out" and "failed".

These different job states have the following meanings:

- *pending*: means that the job still need to be processed and that no daemon is yet assigned for this task. This is the default status that is given to each job when inserted in the queue.
- *in-progress*: denotes that a job is assigned to a daemon and that the job is being processed.
- *completed*: simply means that a daemon successfully completed the job.
- *timed-out*: is set when a daemon process does not send a beacon message (see section 5.5.2), meaning that the process itself is blocked or has failed for some reason.
- *failed*: signifies that the job could not be executed by the daemon process for some unknown reason.

Other fields such as input and output collections are specific to the parallelization framework and are, hence, optional, i.e. they could be removed without altering the behavior of our asynchronous system. We will explain more about these specific fields in the next section.

5.4 User Application and 28.io Modules

In this section, we present several aspects of the implementation. We start by showing how a end-user can easily use the parallelization module to execute a query on a large distributed collection. A simple example is provided for illustration purposes. Afterwards, we explain in details how the parallelization module works in the background and what are the configuration options offered to users. This section is then concluded by the description of the job module.

5.4.1 User Application

The current version of our parallel infrastructure does not include automatic parallelism. 28.io cannot detect (yet) whether or not a query should be executed in parallel on a large distributed collection. Therefore, users still have to write some additional code to explicitly tell 28.io what queries to parallelize across multiple machines.

First of all, users need to wrap their queries inside a function, which must respect a specific signature: it must take a sequence of items as input and returns another sequence of items as output. The following code shows an abstract FLWOR expression wrapped inside such a function.

```
(: Function to be executed in parallel by worker nodes :)
declare function local:query($sequence as item()*)
  as item()*
{
    for $item in $sequence
    where predicate( $item )
    return result( $item )
};
```

The idea behind wrapping FLWOR expressions into functions is to be able to invoke queries remotely from anywhere in the system and to avoid on-the-fly query rewriting. In fact, when a parallel query is broadcast to several processes, each dealing with a specific data chunk, the FLWOR expression contained in the function should be modify such that it will run on the right underlying data chunk and not on the entire collection. Wrapping the queries inside a general-signature function allows to dynamically change its input and output collections without relying on automatic query rewriting. Besides, this spares us the charge of making any important changes to 28.io or Zorba.

Once users have wrapped their query and imported the parallelization module, the parallel query execution can be initialized by calling the `map()` method. What happens next in our parallel infrastructure is hidden from the user.

Concrete Example

Before delving into the details of the parallelization module, we give a concrete example of how users can invoke the module to run a query in parallel from their applications. This example shows in particular how to initialize the parallel infrastructure, pull job status and get the results back once all jobs are completed. We use the simple FLWOR expression from section 4.1, which takes a big "products" collection as input.

```
module namespace local = "http://www.28msec.com/example";
import module namespace parallel =
  "http://www.28msec.com/modules/parallelization";

(: Big Collection :)
declare variable $local:collection as xs:QName :=
  xs:QName("local:products");

declare collection local:products as nodes()*;

(: Wrapped Query :)
declare variable $local:function as xs:QName :=
  xs:QName("local:query");

declare function local:query($input as item()*) as item()* {
    for $p in $input
    let $price := xs:integer($p/price)
    where $price lt 100
```

```
19      let $name := $p/name/text()
20      for $c in $p//country/text()
21      return <product>
22               <name>{ $name }</name>
23               <origin>{ $c }</origin>
24             </product>
25  }
26
27  (: User Program :)
28  declare function local:app() {
29      (: Initialize framework and schedule jobs :)
30      variable $id := parallel:map(
31          $local:collection,
32          $local:function
33      );
34
35      (: Wait for all jobs to complete :)
36      variable $count := fn:count(db:collection($id));
37      while (parallel:status($id)("completed") lt $count) { }
38
39      (: Retrieve and output results :)
40      parallel:results($id)
41  }
```

We can notice that the modifications applied to the original FLWOR expression are minimal, i.e. users only need to replace the name of the large input collection by the parameter variable. In this example, only the initial clause was modified by replacing the whole product collection (`db:collection("products")`) by the `$input` parameter variable.

The `map()` function call on line 30 is non-blocking, meaning that the user application will not stop or wait for the whole query execution to complete. Instead, the function returns a collection identifier immediately after having initialized the query. This identifier is then used to retrieve job status and final results using functions `status()` and `results()` respectively.

The code fragment on lines 35-37 shows how a user can wait for all jobs to complete. However, in a later release, a new function will be available to wait for the whole parallel query to complete without requiring users to iterate over job status, e.g. `parallel:wait-for-completion($id)`.

5.4.2 Parallelization Module

The controller endorses two main roles: initializing the parallel framework (e.g. data chunks, jobs, etc.) and handling the communication with the job scheduler. This is the reason why we decided to split the controller logic into two separate modules, namely the *Parallelization Module* and *Job Module*. We wanted to have a clear separation between the ability to schedule jobs asynchronously and the ability to actually parallelize queries.

In addition of offering a simple interface to end-users, the main functionality of the parallelization module is to initialize the entire parallel framework and allow users to easily retrieve query results.

Framework Initialization

Calling the `map()` function initializes the data partitions by either getting information on operative data chunks, using the store module, or creating new collections dynamically. If new data collections are created, input data items are hashed into those collections. The job module is then invoked multiple times (once per partition) to schedule a set of asynchronous jobs representing the parallel query. Finally, job UUIDs are collected and stored inside another collection whose name is returned to the user application. The collected job UUIDs act as tickets which are used to retrieve individual job status or results. In other words, this function *maps* a FLWOR expression on a group of independent jobs and partitions the whole input data on a number of collection chunks.

The `map()` function takes two mandatory parameters: the function EQName containing the parallel query and the collection name on which the query must be evaluated. Additionally, this function has two optional arguments: an output collection name, in which resulting items are stored, and a JSON object containing one or more parameters for configuration purposes. If no parameter is given, the back-end infrastructure is initialized using default parameter values. In the current version of the `map()` function, we distinguish three possible parameters: the chunk size (i.e. number of items per chunk), the number of chunks (i.e. degree of parallelism), and a user-defined hash function.

In general, the big input collection is already partitioned because data items are stored inside a MongoDB collection "sharded" across multiple machines. To exploit sharding, information about data shards is collected (using the store module) and passed to the job module together with the collection name. Otherwise, if sharding is not enabled on the input collection, names of dynamic collections are given to the job module instead.

We also implemented a variant of the mapping function, `map-on-sequence()`, which takes a possibly large sequence of items rather than a reference to an input collection, such that these data items then needs to be partitioned into new collections. In this case, a chunk size and hash function can be set by the user to configure the system in its own way. Note that this function was mainly implemented for testing purposes and might eventually be removed in a later release.

Query Status

The collection identifier returned by `map()` is enough for users to get some insight about the current status of their parallel query as well as to retrieve all query results.

As shown in the previous example, a user can call `status()` with the collection identifier to obtain information about the overall query processing. This function returns a JSON object containing the job status together with their respective number of jobs. As an example, a call on this function could return the following object:

```
{
  "pending": 6,
  "in-progress": 4,
  "completed": 10
}
```

This implies here that the parallel query is divided into 20 jobs, form which ten are completed, four are being executed, and six are still waiting to be processed.

Query Results

Once parallel query processing is completed, users can invoke the `result()` function, again, with the collection identifier to gather query results. The parallelization module then loops on all jobs affiliated to the user application, reads individually every data chunk and returns the data items contained in them.

Note that the user does not necessarily need to wait for all jobs to complete before calling this function. Typically, if one is only interested in a subset of the answer, the user could, for instance, call `result()` right after the first job completed. In this case, all data items produced by this job will be sent back to the user application together with some partial results from in-progress jobs.

5.4.3 Job Module

The job module offers an interface to easily schedule one asynchronous job at a time. The main role of this module is to communicate with the job scheduler in order to avoid other modules to re-implement HTTP request/response parsing and handling and thus to address the job scheduler directly.

Separating the functionality of the job module from the parallelization module decouples the asynchronous architecture from the rest of our system and, therefore, allows the use of asynchronous communication for tasks other than parallel query processing. For instance, users could typically invoke the job module directly to built their own asynchronous application. Moreover, at the same time, this empowers 28.io with asynchronous communication means that can potentially replace synchronous calls. Asynchronous calls could become handy, for instance, when 28.io pushes user files into the compilation queue.

Job Scheduling

The job module offers three different ways of scheduling a job depending on the type of input. A job can be carried out on either a chunk, a dynamic collection, or a sequence of structured items. Therefore to create one job, user modules must call one of the following functions together with their appropriate parameters:

- `schedule-job-on-collection()` takes the name of a dynamic collection and the function QName containing the user query.
- `schedule-job-on-chunk()` takes the same input parameters as the function on collection, plus an additional argument containing information about a MongoDB shard.
- `schedule-job-on-sequence()` takes a sequence of items instead of a collection name and the function EQName.

All these functions also take an output collection name as optional argument. Depending on the function invoked and the given parameters, the module may trigger different kinds of scheduling process.

First, if no output collection is given by the caller, one is automatically generated. Next, if the function called is `schedule-job-on-sequence()`, the given sequence of items is stored inside a new input collection. This way, the problem of processing a sequence in parallel is reduced to the problem of processing an already existing input collection. Once these pre-processing operations are done, the job object can be created as follows using given the input parameters and sent to the scheduler with an HTTP request.

```
{
  "input": {
      "collection": $input-collection,
      "chunk": $chunk-info
  },
  "output": {
      "collection": $output-collection
  },
  "application" : {
      "name": $project-name
  },
  "function" : {
      "name": $function
  }
}
```

Note that the chunk information is only present inside the JSON object if the function called is `schedule-job-on-chunk()`. Otherwise, the "input" object consists only of one field, namely "collection". One can also notice that, according to the job object structure presented in section 5.3, the above JSON object does not contain all required fields. It is the role of the scheduler to insert the missing job properties and not the job module. Once the object is built, it is sent to the scheduler using a POST request on the collection URI.

Other Functions

In addition to the above scheduling function, the module interface exposes two more functions to users: `status()` and `result()`. The `status()` function takes a job UUID as input, sends a GET request to the scheduler and returns the corresponding job status.

The `result()` method also takes a job UUID as input, reads the output collection content using a GET request and returns the produced data items back to the caller.

Additionally, the job module contains two private functions. One to create collection on the fly if the function parameters did not specify any, and another that handles HTTP communication with the scheduler. The latter function does in particular prepare the request body, set the content type, parse the response, and handle potential errors.

5.5 Job-Processing Daemon

The Job-Processing Daemon assumes the role of a *worker* or processing node. It is an executable that runs on a single machine and that can launch several parallel processes to handle pending jobs.

In this context, the conceptual *pool of workers*, presented in section 3.3.5, is actually implemented as a cluster of machines. Therefore, if we want to add processing power to our parallel infrastructure, we can do this simply by starting up new machines, each with a running daemon, and including them in the cluster.

As opposed to other infrastructure components, the daemon is not implemented in XQuery but in C++ for efficiency reasons. This way, instead of working on jobs using the scheduler API, the daemon directly accesses the queue collection using the C++ MongoDB driver.

All active daemons are pulling the same shared collection representing the scheduler queue. Since this queue is accessed in parallel by several processes, we needed a locking mechanism

```
while (true) {
    // fetches pending job from scheduler queue
    uuid = fetchPendingJob()

    if (uuid) {
        pid = fork() // creates child process

        switch (pid) {
          case 0: /* Child Process */
            // extracts query from job object
            query = retrieveQueryFromJob(uuid)

            // periodically sends beacon and runs query
            keepaliveSignal(uuid)
            execute(query)

            // marks job as completed
            editJobProperty(uuid, "status", "completed")

            exit(0) // stops child process

          default: /* Parent Process */
            childPIDs.add(pid)
            // eventually kills child process on timeout
            watchForTimeout(uuid, pid, time)
        }
    }
    while(childPIDs.size() >= MAX_CHILD_PROCESSES) {
        // parent process waits for child processes to complete
        completedPID = wait()
        childPIDs.remove(completedPID)
        sleep(1)
    }
}
```

Figure 5.3: Daemon Pseudo Code

for concurrency control. To this extent, we decided to use the locking protocol of MongoDB to allows multiple daemons to access the queue concurrently while preventing them from modifying the same job object at the same time and, thus, keeping job objects consistent.

In the remainder of this section, we explain in more details how the job-processing daemon is actually working with the help of the pseudo code presented in Figure 5.3.

5.5.1 Parent Process

The daemon starts with a main process that directly enters an infinite "while" loop. In each iteration, the main process inspects the scheduler queue for new pending jobs. If a job is found, the daemon fetches it; otherwise, the main process reiterates and tries again.

Fetching Jobs

Once the daemon finds a dependent job, i.e. a job that should run on the data residing in the underlying machine, the daemon retrieves this job and updates three of its properties: the daemon name is written in the "lock" field, and both "lock-acquired" and "accessed" are updated with the current system time.

Forking Child Processes

Once a job has been fetched by the daemon, the main process forks, meaning that it creates a copy of itself, called child process, that inherits most of its attributes. The parent process then transmits the job to its new child process, starts a timeout watcher for this same child process, and looks again for pending jobs.

Timeout Watcher

Just after forking, the parent process launches a parallel thread that will watch over the child process. We call this thread a *Timeout Watcher*, because it keeps child processes under surveillance by watching over the timeout value of in-progress jobs. As soon as the timeout becomes too old according to the beacon time interval, the timeout watcher sets the jobs status to "timed-out" and sends a SIGKILL signal that immediately terminates the child process.

Waiting for Child Processes

While child processes are running, their parent process continues to search for pending jobs in the scheduler queue. If one new job is found, the parent process forks again, creating a new child process to handle the job. This action is repeated as long as the number of running child processes do not reached a certain threshold. When the maximum number of child processes is reached, the parent process pauses its iteration and waits for one or more child processes to complete. We define the maximum number of child processes as being the number of cores from the machine (on which the daemon is running) plus one: $\#cores + 1$, e.g. 5 or 9 for quad-core and eight-core respectively. When at least one of its child processes returns, the main process is restarted and searches again for new pending jobs.

5.5.2 Child Process

The task of a child process is to execute the function attached with the job. Based on other job parameters, the child process builds a special query that will be used to execute the user-defined function on local data. To this purpose, the input collection name, output collection name, and function EQName are extracted from the job object. Additionally, the function namespace is detached from the function EQName in order to import the user code and execute it locally on the data. All these parameters are then used to construct the following query:

```
1  import module namespace ns = functionNamespace
2
3  let $input-data  := db:collection(inputCollectionName)
4  let $output-data := functionEQName($input-data)
5  return db:insert(outputCollectionName,  $output-data)
```

Note that variables *functionNamespace*, *inputCollectionName*, *functionEQName*, and *outputCollectionName* are replaced at runtime by the child process itself.

Keep-Alive Signal

While the child process is processing the query, it sends beacon messages periodically, i.e. a keep-alive signal, to the scheduler. This task is attributed to a thread that works in parallel to the child process. A beacon message is used to tell other system components that the job is currently being executed by a child process. The beacon simply updates the "accessed" job property with a new timestamp value. After sending a beacon message, the thread is put to sleep for a specific amount of time, which gives the beacon time interval.

As soon as a timestamp value becomes older than the beacon time interval would allow, one can conclude that the child process has failed or is blocked for some reason. The job is then marked as "timed-out" by the timeout watch such that one can eventually reschedule the job and reassign it to a new process automatically.

Completing Jobs

As soon as a query has been run on the input data, the child process sets the current job status as "completed". The child process then stops, frees its resources, and exits with a non-negative value returned to the parent process.

Chapter 6

Experiments

In this chapter, we present the experiments carried out to validate both the overall approach and its implementation. We start by highlighting the precise goals and performance expectations of these experiments. Afterwards, the hardware environment together with the database benchmark used to measure the system performance are presented. At the end of this chapter, the performance measurements are compared to those of serial processing, before being related to the actual expectations.

6.1 Goals and Expectations

We decided to test our overall approach and the first version of the parallel infrastructure by performing two experiments:

1. *data bulk-loading* and
2. *point-query processing*.

Both experiments are run in turn using different sizes of input data. The idea is to first load a variable quantity of data on 28msec's servers during the bulkload experiment and then to use those loaded data for the point-query experiment.

The proof-of-concept experiments presented in this chapter aim at validating our overall approach. Their primary goals is therefore to prove that the Tree-Hammock topology together with the system architecture can be used to parallelize data-intensive queries and to demonstrate clear performance improvements compared to serial processing. Moreover, these experiments should allow us to get some insights about the behavior of our framework and to determine which factors have the most impact on it. Finally, since the parallelization framework pushed the 28.io platform to new limits, these experiments have also as objective to highlight where lie the performance bottlenecks and what are the possible enhancements to remove them.

Before presenting the results, it must be pointed out that the volumes of input data used in these experiments do not match the petabyte scale and are thus relatively smaller than Big Data. As explained above, we would like to first prove that the framework fulfills our exceptions for small datasets before testing it with much larger collections. In particular, this will allow us to make some improvements on the software, both in the infrastructure and in 28.io platform, before scaling the experimental data to a higher level.

6.2 Environment

In this section, we describe both the hardware and software environment used to deploy our framework and run the experiments. In order to evaluate the complete infrastructure, we use Amazon Web Services (AWS) [49] together with an XML database benchmark, called TPoX [46, 47].

6.2.1 Hardware

During the experiments, our parallel infrastructure was deployed on the Amazon Elastic Compute Cloud (Amazon EC2). Amazon EC2 is a web service that provides resizable compute capacity for Web-scale computing [49].

Different types of instances are proposed by Amazon EC2. 28.io runs on a cluster of M1-Large instances. Each M1-Large machine is a 64-bit platform that provides four EC2 compute units (i.e. two dual-core CPU), 7.5 GB main memory, and 850 GB instance storage. All experiments were performed using ten M1-Large machines, given a total of 40 processing units.

6.2.2 Software

We use the 28.io platform (Zorba and MongoDB) together with the "Transaction Processing over XML" (TPoX) benchmark [46, 47] to measure the performances of our parallel infrastructure.

The TPoX benchmark is an application-oriented and domain-specific benchmark, developed by IBM, which exercises all aspects of XML databases including storage, indexing, logging, transaction processing, and concurrency control. TPoX is based on the analysis of real XML applications and simulates a financial multi-user workload with XML data conforming to the FIXML standard [48].

During the experiments presented in this chapter, we only use a subset of TPoX. Among the three different types of XML documents offered by TPoX (*Order*, *Security*, and *CustAcc*), we only used *Order* documents, each having a size of 1-2 KB. Furthermore, we did not use the TPoX workload because, even if it represents a real system utilization, it is composed of a set of query, insert, update, and delete operations that are not yet supported by our framework. This is why we decided to restrict the workload to simple (predicate) queries and, therefore, to only use query *Q1: get_order* as point query for the second experiment.

Factors

We distinguish five factors that may be changed to influence parallel processing of the framework:

1. *collection size*: size of the input dataset on which the experiment is run;
2. *number of data chunks* or *chunk size*: volume of data per MongoDB shard;
3. *number of jobs* or *job size*: volume of data that is handled by a process;
4. *number of daemons*: number of machines taking part in the parallel processing;
5. *maximum number of child processes* per daemon: degree of parallelism on each individual machine.

For the sake of simplicity, we decided to fix all factors except the collection size in both experiments. For varying the size of the input dataset, the notion of *batch* is used as unit of measurement. We define a batch as a group of exactly 48'000 XML documents having a size of 70 MB on average (46-94 MB). In both bulkload and point-query experiments, we vary the number of batches from 1 to 10, given a collection size ranging from approximately 50 MB up to almost 1 GB. This is illustrated in more detail in Table 6.1.

If we consider that a processing node can take exactly one jobs and one input chunk at a time, then one can safely assume that the chunk size and the job size are equal. Nevertheless, it is possible to have different chunk and job sizes, where a job defines a certain amount of data rather than a single chunk. In this case, we assume that the amount of data per job can be equal or grater than a chunk but not lower, since the chunk determines the granularity of the parallel processing. This gives the following rule of thumb: *chunk-size* $\leq$ *job-size*, or equivalently *#chunks* $\geq$ *#jobs*.

#Batches	#Documents	Avg. Size [MB]	Range Size [MB]
1	48'000	70	47-94
2	96'000	141	94-188
3	144'000	211	140-281
4	192'000	281	188-375
5	240'000	352	234-467
6	288'000	422	281-563
7	336'000	492	328-656
8	384'000	563	275-750
9	432'000	633	422-844
10	480'000	703	469-938

Table 6.1: Variable Factors of the Experiments

Finally, the number of daemons corresponds to the number of machines that participate in the parallel processing. Since the cluster is composed of ten M1-large computers, the number of daemons is set to ten as well, each daemon then running on a different machine (*#daemons=10*). As for the maximum number of child processes, it is defined as in section 5.5: *#cores + 1*. Because there are four computing units per machines, the number of processes is thus set to five (*#child-processes=5*).

Time Overhead (Jobs vs. Slots)

When increasing the input collection size, an important relationship to observe is the number of jobs compared to the number of available *slots*, where a slot corresponds to a processing node or child process.

As long as the number of jobs is equal to or lower than the number of processing nodes, there will be no overhead during parallel processing. However, as soon as the number of jobs exceeds that of the processing nodes (*#jobs* $>$ *#slots*), the elapsed time required to complete the parallel query is slightly increased. This is due to the additional jobs that need to wait because all available slots/nodes are already processing a first level of jobs. Furthermore, once there is more than twice as many jobs as slots (*#jobs* $>$ *2*#slots*), another overhead occurs as well. This elapsed time overhead appends again, each time the number of jobs exceeds the number of available slots by another factor (*#jobs* $>$ *N*#slots*).

For these reasons, we do not wait for a constant performance, but rather expect the elapsed time to increase with the number batches following a stairs-like shape, where each

	Bulkload (Exp.1)			Point Query (Exp.2)		
#Batches	Parallel	Serial	#Jobs	Parallel	Serial	#Jobs
1	00:52	03:26	8	00:13	00:30	48
2	01:19	07:28	16	00:15	01:22	96
3	01:22	09:02	24	00:22	01:37	144
4	02:01	13:37	32	00:27	02:03	192
5	01:56	17:54	40	00:52	02:22	240
6	01:59	20:51	48	00:49	03:39	288
7	03:25	28:17	56	01:01	04:41	336
8	02:52	32:00	64	00:57	04:32	384
9	02:53	39:23	72	01:14	06:27	432
10	03:35	44:15	80	01:07	06:02	480

Table 6.2: Experiment Results (elapsed time in [mm:ss])

step corresponds to the number of jobs exceeding the number of slots by another multiplicative factor.

Additionally, executing a query in parallel comes with a cost. Indeed, creating and scheduling all job objects adds a penalty to the overall processing. The time overhead is therefore proportional to the number of jobs used to execute a query in parallel and, thus, to the collection size.

6.3 Results

This section presents both the bulkload and point-query experiments and draws some conclusion based on their respective results. Results are shown in Table 6.2 together with the number of jobs/chunks used in each experiment.

6.3.1 Experiment 1: Bulk-loading

As explained at the beginning of this chapter, the first experiment is to populate a database using pre-generated XML documents. Prior to running the experiment, ten XML files are generated and uploaded on Amazon EC2. Each file represents a batch made of TPoX *Order* documents.

Depending on the scaling factor, i.e. the number of batches, the database is populated using either one, some, or all XML files. Every batch/file is split into height ranges that represent the data chunks, each containing 6'000 documents.

After determining the partitioning, data bulk-loading is initialized by calling the `map()` function with four parameters: the input collection (*order-chunks*), the worker function (*order:load*), the output collection (*orders*), and the chunk size per job (set to 1). The reason why the chunk size is set to one is because we want each processing node to handle one single chunk, i.e. to fetch one object from the "order-chunks" collection containing an input file and a reference to the first document (of the data partition).

```
1  declare function exp:parallel-bulkload($scale as xs:integer)
2  {
3     for $s in 1 to $scale-factor
4     for $i in 0 to 7
```

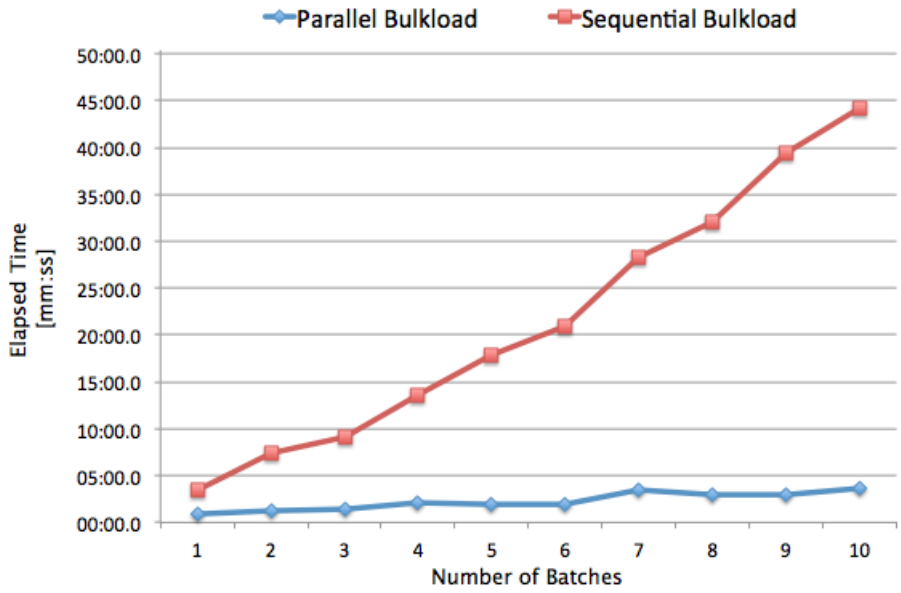


Figure 6.1: Bulkload Experiment

```
    return
      db:insert("orders-chunks",
       {
         "file": "http://s3.aws.com/28msec/batch-" || $s || ".xml",
         "start": 6000 * $i + 1
       }
      );

  parallel:map(
     "orders-chunks",
     xs:QName("order:load"),
     "orders",
     {"chunk-size": 1}
  )
};
```

The parallel infrastructure then schedules a total of height jobs per batch. Each job is assigned to a daemon process that simply connects to MongoDB (also running on the Amazon EC2 instances) and uploads part of the documents contained in one of the XML files into a new MongoDB collection (*orders*).

```
declare function order:load($items)
{
  for $item in $items
  let $file := $item("file")
  let $start := $item("start")
  let $stream := fetch:content($file)
  return
    parse:parse($stream)[position() ge $start][position() le 6000]
};
```

Note that, in the case of serial bulk-loading, only one job is scheduled by the parallel infrastructure, such that a single daemon process handles all data chunks.

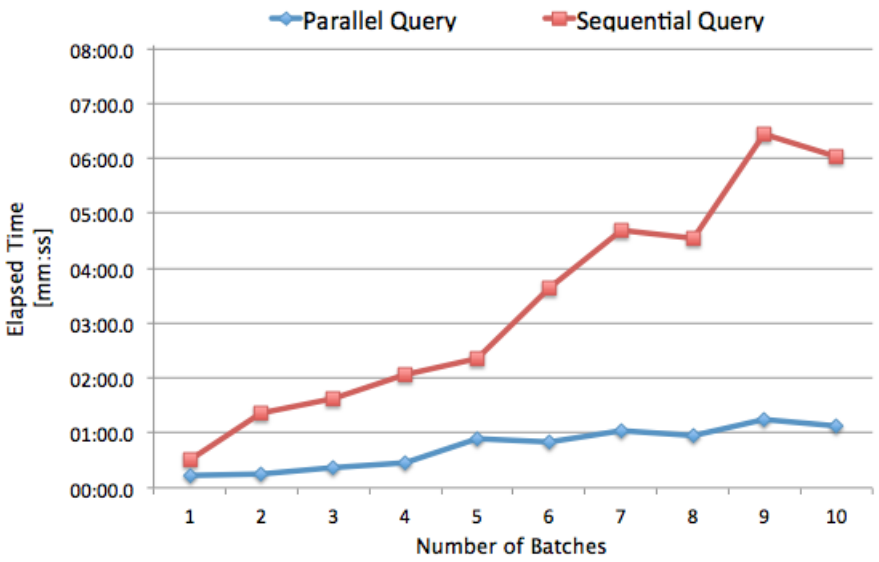


Figure 6.2: Point Query Experiment

Figure 6.1 shows the experiment results for both serial and parallel bulk-loading of XML documents. As we can see the parallel bulkload outperforms serial processing by an order of magnitude. If we look more carefully at the results from Table 6.2, one can see that parallel bukloading is more than ten time faster than its serial counterpart for a number of batches superior to five. Parallel bulk-loading using our framework clearly shows performance improvements compared to sequential bulk-loading. As we will see in the next section, this trend can also be observed for the point query.

6.3.2 Experiment 2: Point Query

The second experiment consists of a full table-scan to retrieve a single document from a distributed database collection. This experiment queries the pre-generated XML documents loaded during the previous experiment, which now reside inside a MongoDB collection.

The parallel query processing is initialized by calling the same `map()` function taking as parameters the input collection containing the *Order* documents and the chunk/job size. The chunk size is set to 1'000 documents (1-2 MB), given a total of 48 chunks per batch.

```
1  declare function exp:parallel-point-query()
2  {
3      parallel:map(
4          "orders",
5          xs:QName("order:get"),
6          {"chunk-size": 1000}
7      )
8  };
```

Notice that, for serial query processing, only one job is scheduled by the parallel infrastructure. This is done by directly invoking the job module's `schedule-job-on-collection()` function (rather than the parallelization module). Hence, only one job is created and only one processing node is responsible to query to whole input collection.

As point query, we use the TPoX Q1 query *get_order* which returns a full *Order* document without the FIXML root element. The daemon returns the document if it is part of the input data; otherwise, no data items are returned.

```
1  (: Query1: Retrieve an order with the specific ID :)
2  declare function order:get($items)
3  {
4      $items/fixml:Order[xs:int(@ID) eq xs:int(103282)]
5  };
```

Results for this second experiment are illustrated in Figure 6.2. Even if the difference in elapsed time between parallel and sequential processing is not as strong as in Experiment 1, their performances still differ by an order of magnitude.

Because the measurements were run once for each size of the input collection, one can remark some fluctuations in the graph. However, these fluctuations are negligible since they do not bias the overall performance trend.

All in all, these experiments have confirmed our expectations by demonstrating clear performance improvements from our parallel infrastructure compared to serial processing. Both experiments, i.e. bulk-loading of pre-generated data items and point query using full table scan, have shown that our framework outperforms serial processing by an order of magnitude. These results therefore validate the overall approach, namely that our system architecture can be used to parallelize data-intensive XQuery expressions on datasets having a reasonable size. More experiments are naturally required to confirm that our approach can also cope with Big Data. However, based on the current results, we are convinced that the parallelization framework can already be used to query larger amounts of data.

Chapter 7

Future Work

In addition to all possible improvements already mentioned in this thesis, this chapter highlights most probable or valuable enhancements that can be implemented in future releases of our parallelization framework.

7.1 Supporting more FLWOR Expressions

An obvious next step for our infrastructure is to allow more types of queries to be supported by our infrastructure. Remember that for the moment our system only deals with simple FLWOR expressions as described in section 4.1.

The next parallel queries that we plan to implement are ordering queries. As we have shown in chapter 4, this type of queries requires an additional pass on the input data, such that an additional layer of special workers can shuffle data items around. This extension requires two key enhancements of our infrastructure. First, we need to implement shuffling of a large collection. Since the context of a daemon process is currently limited to one input collection and one output collection, we will therefore need to extend the daemon such that it will be able to write results into several output collections. Second, we need to introduce the notion of dependency between jobs. To be run in parallel, an ordering query must be split into two distinct sets of jobs, where the first implements shuffling and the second is used for partial ordering. These two sets of jobs are therefore dependent on each other: the first set must be completed before the second can be scheduled.

Having multiple processing steps for the same query requires extra planning. Hence, we need another component (on top of the parallelization module) to handle the control flow of jobs in order to schedule the different sets of jobs separately, one at a time. Note that another solution would be to add a priority field into job objects to tell daemons which to execute first.

Moreover, further changes in the daemon process would also be necessary to execute these queries. In fact, it should be possible for a daemon to accept not only one but two or more input collections to either aggregate or join intermediate results.

Once all these key enhancements are included in our parallel framework, it will be possible to support other types of FLWOR expressions, such as aggregation and join queries, and to even emulate the state-of-the-art frameworks. The following code shows how a future version of our own framework could be used to emulate the MapReduce programming model.

```
(: Function to be executed in parallel by worker nodes :)
declare function local:query($input as item()*)
  as item()*
{
    for $item in $input
    let $pair := map( $item )
    group by $key := $pair("key")
    return {
      $key, reduce( $pair )
    }
};
```

7.2 Materialization of Intermediate Results

As soon as our infrastructure will allow multiple levels of jobs for the same parallel query, it will have to deal with intermediate results in some way. So far, we simply materialize all resulting items into new collections, because the infrastructure does only support simple FLWOR expressions, which require one single layer of processes. This means that data items produced by these processes already represent the final query result and must be physically stored in order to be later retrieved by users. However, as soon as multiple passes on the data are necessary, intermediate results will be materialized as well. Copying entire items back and forth from collection to collection and forwarding them over the network between processes are costly operations that may lead to serious processing overheads.

The idea is then to restrict copying and sending relevant data only and use references (e.g. UUID) pointing to the original items. This approach will considerably reduce both the communication costs and I/O storage costs for materializing intermediate results. Nevertheless, further investigations will be need regarding the use of remote references, i.e. pointers referencing non-local resources.

7.3 Automatic Parallelism

We have seen in chapter 5 that the user needs to adapt his/her application code in order to run queries in parallel.

First, the user needs to wrap its queries inside functions having a specific signature. Then, after importing the parallelization module, the user must invoke methods from the module API in order to initialize the infrastructure, launch parallel query processing, and retrieve query results. Furthermore, to obtain better performances, a user may configure the parallel infrastructure by passing additional parameters to the module (together with the wrapped query and the input collection). Obtaining the best performance for a given query is a non-trivial task since it highly depends on the query itself, its input data, as well as the current system state including workload, available resources, queue size, etc.

All together these tasks represent a significant work overhead for end-users. As a consequence, by introducing automatic parallelism, the usability of our framework will be considerably improved. Then, instead of asking the user to manually parallelize its queries, FLWOR expressions could be automatically run in parallel.

We plan to develop a new system component that will (1) detect when a query should be run in parallel, (2) examine what query plan and system configuration would best suit that query, and (3) automate parallel processing on top of our infrastructure.

7.4 Instruction-Level Parallelism

In this thesis, we only investigated the hammock part from our tree-hammock topology and focused exclusively on data-intensive computing. Obviously, efforts are still required to investigate the tree topology of our approach and, thus, to exploit parallel processing at the instruction level. Moreover, we will also need to examine how this could be implemented in symbiosis with our hammock pattern.

The reasons why we did not further investigate the tree pattern in this thesis are, firstly, because parallelism occurs at a deeper level in the query processor and would have required modification of Zorba to safely use multi-threading and, secondly, because this is not of primary interest when dealing with large amounts of distributed data. Nonetheless, we are convinced that the use of parallel instruction processing represents a high potential to boost query execution, not only in the context of our infrastructure, but for any code or query complied and run with Zorba.

7.5 Fault Tolerance and Robustness

Up to now we only covered process failures in the job-processing daemon, where the parent process watches over its child processes and eventually marks their jobs as timed out if they are blocked or failed for some reason. Unfortunately, failures are not limited to processes and can be more general. A machine together with all its processes could, for example, crash or an entire cluster could abruptly be disconnected from the network and become (temporary or permanently) unavailable.

At a certain point, we will thus need more robustness and finer error handling to recover from those failures. In particular, we would like to be able to automatically reschedule failed and timed-out jobs. Besides, we plan to introduce other failure-tolerant mechanisms to prevent (rather than handle) errors. For instance, an interesting extension that will probably be brought later into our system is the use of *backup tasks* as presented in [21]. Scheduling backup executions of in-progress jobs when the query execution is near completion will allow us to deal with blocker nodes, called *Stragglers*, i.e. slow machines that stall the whole query processing.

Additionally, more investigation will be necessary regarding the failure of controllers, i.e. master nodes. This may not be a major problem in our parallel infrastructure, however, because all jobs and tickets exchanged between system components are systematically persisted into stable storage.

7.6 Advanced Experiments

We will need to run other experiments in addition to those presented in chapter 6 in order to further validate our approach and get more insights about both the parallel infrastructure and the asynchronous queuing system. In particular, we would like to analyze the cost overhead required to initialize the complete infrastructure, as well as to study the impacts of a full scheduler queue on the system.

Parallelizing query on to of our framework is not a free-lunch operation and comes with a cost that is due to dynamically creating collections, hashing data items, and scheduling jobs. Therefore, this first experiment about the initial cost overhead aims at determining which and when queries would benefit from parallelism, or whether the time needed for initialization is compensated by the time gained in running a given query in parallel. Note that the knowledge acquired from this experiment could typically be used for automatic parallelism later on.

On the other hand, the second experiment will allow us to establish what happens if the scheduler queue becomes full, for example, in case of an overloaded system or if jobs are not removed from the queue once completed. We hope that the results of this experiment will provide some insights about when and at which frequency to apply garbage collection to remove completed (or failed) jobs.

Chapter 8

Conclusion

In this thesis, we presented the Tree-Hammock topology, a dual model that provides both data-level and instruction-level parallelism for XQuery expressions. In parallel, we designed a scalable system architecture that leverages the Tree-Hammock model and uses data-intensive computing to orchestrate parallel query execution over large volumes of distributed data. We also demonstrated that all our presented use cases can be instantiated on top of this system architecture and can therefore benefit from parallel processing.

Moreover, we successfully developed a simple, yet powerful infrastructure which allows predicate queries to be run in parallel while offering some parallelization capabilities for XQuery programmers to tune their queries. The experiments carried out at the end of the thesis validated our approach, as our parallel infrastructure clearly outperforms serial processing by an order of magnitude. The first set of results were therefore convincing and motivated us to undertake further experiments in order to accurately sketch the boundaries of our infrastructure.

The next development step is now to enable parallel execution of more complex queries. In particular, we are confident that it will soon be possible to emulate the MapReduce programming model with the help of our framework. Although further investigations are still needed, we are convinced that this is a very promising approach which opens new perspectives for querying Big Data in the Cloud.

Appendix A

XQuery Expressions

In this appendix, we explain and classify all types of XQuery expressions in addition to those already presented in chapter 3. This chapter aims therefore at completing this chapter to provide the reader with a full picture of the XQuery language.

As mentioned in section 3.1, XQuery expressions can be classified in two different types: those which contain variable bindings as operands and those which do not. We call them *complex expressions* and *simple expressions*, respectively. These two categories can in turn be subdivided into other categories.

A.1 Simple Expression

The vast majority of XQuery expressions are *simple* in the sense that they logically evaluate their operands only once and do not bind any variable before evaluating any operand. For example, an additive expression, like `1+2`, is evaluated as follows. First, both its left-hand-side operand and right-and-side operand are evaluated exactly once. Then, results of these operands are then added together to produce the final result item.

Simple expressions can themselves be subdivided into several distinct categories based on their arity, i.e. the number of operands that are taken as input. Figure A.1 presents these different categories with some examples.

A.1.1 0-ary Expression

This is the most simple type of expression. It has no operand and just produces one item.

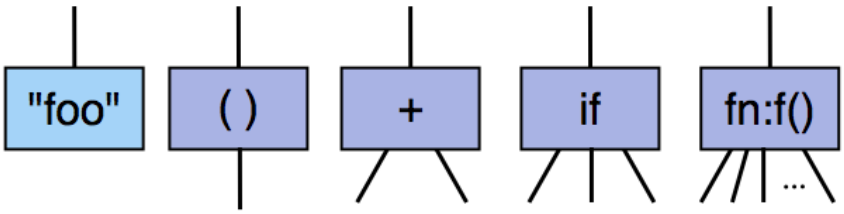


Figure A.1: Simple Expressions

Literal

Number and string literals belongs to this category (e.g. `1`, `"foo"`). For instance, the hello-world program

```
"Hello, World"
```

returns a sequence of one item of type xs:string, i.e. a string literal.

Named Function Reference

A named function reference denotes a function defined in the static context for the query that is uniquely identify by both its name (as a extended QName) and its arity. For example, `fn:abs#1` is a named function reference on the (built-in) absolute function that has exactly one parameter.

A.1.2 Unary Expression

Unary expressions have a single operand. Among all unary expressions, the most encountered is probably the parenthesized expression.

Parenthesized Expression

A typical expression having only one operand is the parenthesized expression *"("* *Expr* *")"* that is used to enforce a certain evaluation order. This kind of primary expression can be seen as a simple forwarding operation that takes an operand as input, evaluates it, and outputs its result. For example, `(1)` contains a number that is evaluated as described in the previous section and then output by the parenthesized expression as a sequence of one item.

Ordered and Unordered Expressions

Ordered and unordered expressions set the ordering mode for a certain region in a query that is enclosed by curly braces:

```
ordered {1,2,3}
```

Here, the ordered expression contains three literal expressions separated by commas ",".

Unary Arithmetic Operators

Addition and subtraction operators both come with their unary and a binary variants. The unary operator changes the sign of the given operand as follows: `-1`. Binary arithmetic operators are presented in the next section.

Expressions on Sequence Types

Among this king of expressions, four of them are unary: instance-of, treat, castable, and cast. These expressions have actually two operands (a type and an expression), but we treated them as unary expressions since only one of their operands can be substituted by another expression. An instance-of expression checks if the first operand match the given sequence type and returns a boolean value. For example,

```
0.5 instance of xs:integer
```

returns `false`, because `0.5` is actually a number of type xs:decimal. Castable tests whether a given value is castable into a given target type.

```
"foo" castable as xs:integer
```

will return `false` since `"foo"` is a string made of characters and can therefore not be cast into a integer. Trying to cast "foo" as an integer will result in a dynamic error. Cast expression creates a new value of the specified type based on the existing value. Thus,

```
1.5 cast as xs:integer
```

returns `1` of type xs:integer. Treat expressions can modify the static type of its operand:

```
1 treat as xs:decimal
```

returns value `1` of type xs:decimal. Note that unlike cast expression it does not change the dynamic type or value of its operand.

Validate Expressions

A validate expression is used to validate a given document node or an element node with respect to the in-scope schema definition (XML Schema).

```
1 import schema namespace ns = "http://www.zorba-xquery.com/schemas/
    pul";
2 validate {<ns:pending-update-list/>}
```

returns the element unchanged if it is valid according to the imported XML Schema; otherwise the above expression returns a dynamic error.

Extension Expressions

Extension expression is a special type of unary expressions that allows programmers to extend XQuery with new own expressions.

```
1 module namespace ext = "http://www.example.com/xquery-impl";
2 (# ext:expr #){ 0 }
```

The above example shows a valid extension expressions that simply contains one number in its body.

A.1.3 Binary Expression

As its name suggests, a binary expression has two operands that are both evaluated before combining them according to the semantics of a certain binary operator. There are plenty of binary expressions in XQuery 3.0 such as (binary) arithmetic operators, string concatenation, comparison expressions, logical expressions and sequence expressions.

It is important to highlight here that some binary expressions presented in this section are associative, such as sequence expression, string concatenation, logical expressions, as well as addition and multiplication operation. As a consequence, operands of associative expressions do not require to be evaluated in a specific order, and can therefore be considered as N-ary expressions once multiple operators are used together.

Sequence Expression

In XQuery 3.0, it is possible to construct, filter, and combine sequences of items (using operands ",", **union**, **intersect**, **except**). A sequence can be constructed by using a comma operator, which evaluates each of its operands and concatenates the results, in order, into a single result sequence. In the same manner, two sequences can be combined with a comma operator. Again, every operand of each sequence is first evaluated before the resulting sequences to be combined. For example, an expressions such as

```
(1,(2,3)), (1,"foo"), "bar"
```

will result in the following sequence `(1,2,3,1,"foo","bar")`. Note that a sequence can contain duplicated items, but can never contain nested sequences. Nested sequences are automatically flattened. Another binary expression that can be used to construct a sequence of consecutive integers is the *range expression*. For example, `(1 to 5)` is evaluated to `(1,2,3,4,5)`.

Node sequences can be combined using set operators such as union (**union** or "|") that combines two node sequences, intersection (**intersect**) that returns a sequence containing nodes occurring in both sequence operands, and difference (**except**) that returns sequence with nodes occurring in left-hand-sided operand but not in right-hand-sided operand.

Binary Arithmetic Operators

In addition to unary (+/-) operators, XQuery also provides binary addition, subtraction, multiplication, division and modulus operators (+, -, *, div, idiv, mod). Arithmetic expressions are grouped from left to right. Therefore, expressions from the left-hand side are first evaluated before combining their results with the right-hand sided operands. For example, an operation such as `3*7 mod 5` returns `1` because the multiplication is evaluated before the modulus.

String Concatenation

String concatenation expressions (||) allow two string values to be concatenated together as follows.

```
"Hello "|| "World."
```

Comparison Expressions

These expressions allow two values to be compared. There are three kind of comparisons, which all returning a boolean result. Value comparisons (**eq**, **ne**, **lt**, **le**, **gt**, **ge**) are used for comparing single values, e.g. `"foo"eq "bar"`; general comparisons (=, !=, <, <=, >, >=) are existentially qualified and may be applied to sequences of any length, e.g. `(1,2)=(2,3,4)`; node comparisons are used to compare two nodes based on their identity (**is**, <<, >>), e.g. `<a/> is <a/>`.

Logical Expressions

A logical expression is either an and-expression or an or-expression (**and**, **or**). Like comparison expressions, logical expressions also returns a boolean value as final result. In order to evaluate a logical expression, the boolean value of its operands must first be computed. For example,

```
1=(1,2) or ()
```

in the above expression, the effective values of `1=(1,2)` and `()` are `true` and `false` respectively. The resulting boolean value of this or-expressions is therefore `true`.

A.1.4 Ternary Expression

There is only one type of expressions having exactly three operands: conditional expressions are based on the combination of keywords **if**, **then**, and **else**. For example:

```
if (1<2) then "foo" else "bar"
```

Conditional expressions evaluate all three operands, and output the result either of the *then* or *else* clause according to the boolean value computed in the *if* clause. In the above example, `"foo"` is returned because `(1<2)` is always `true`.

A.1.5 N-ary Expression

This kind of expression has a certain number of operands, often greater than three. These expressions evaluate all operands and concatenates their values, while preserving spatial ordering.

Static Function Calls

All static (build-in or user-defined) function calls such as `fn:concat("a","b","c")` are considered as N-ary expressions, where N is the number of parameters that can be passed to the function. Such expressions have as many operands as the function's arity indicates. It evaluates all operands, calls the function and outputs the resulting sequence of items.

Inline Function Expression

An inline function expression creates an anonymous function that is coded directly inside the query. An inline function returns a reference to the function body (its implementation) as well as its signature, parameter names, and non-local variable bindings. The reference is usually bound the a variable in order to be called later in the query, as shown in the following example.

```
$f := function ($x) {$x + $y}
```

Inline functions have an important property: within the function body, in addition to the function parameters, one can access any variable which was in-scope at the time the inline function was created. This is commonly called a closure in functional languages, which allows the function body to access variables outside its immediate lexical scope.

The body expression is not directly considered as an operand. Instead, only descendant (non-parameter) free variable expressions are considered operands. It evaluates the free variable operands and builds a function using the operand results as non-local variable bindings. In the example, `$y` is such operand: it is bound in the query before the inline function is created and can already be evaluated as opposed to `$x` which is a function parameter that has not yet be bound to any value. As a result, `$x` can neither be evaluated, nor be considered as operand.

Inline functions do not necessarily include variable bindings in their body and, therefore, are considered as complex expression by default. For instance, `function ($x){$x + 1}` is an inline function expression without free variables.

Partial Function Applications

Partial function application is a feature that comes from functional programming. A partial function application looks like a normal function call using a question mark (`?`) as placeholder in one or more of its parameters. They allow us to fill in parts of a function call, leaving some parameters open for a later invocation in the query.

For example, `fn:add(1, ?)` returns a new (anonymous) function together with its parameter names, signature, implementation and non-local variable bindings, plus a binding of the argument value (e.g. `1`) for each of the corresponding parameter name. Note that the result of a partial function application is somehow similar to an inline function expression. This new function can then be called using a dynamic function call (see section A.2), which will increment the value of its parameter by one.

As for static function calls, the arity corresponds to the number of parameters that do not contain placeholders.

Try/Catch Expression

This expression provides error handling for dynamic errors and type errors raised during dynamic evaluation. A try/catch expression evaluates the expression contained inside the try clause. If no error occurs during evaluation, its result item is returned; otherwise an error is thrown and a eventually handled by a catch clause if error codes match.

```
declare namespace err = "http://www.w3.org/2005/xqt-errors";
try {
  3 + 2 > "30"
} catch err:XQDY0004 {
  "Invalid type: xs:string compared to xs:string."
} catch * {
  concat("Caught ", $err:code)
}
```

In this example, the try clause generates an error that is then caught by the first catch clause, which then returns a string message.

Switch Expression

A switch expression chooses one of several expressions to evaluate based on input values evaluated in the switch operand and other case operands. Each switch expression may contain one or more case clauses (each having one case operand and one single expression) and a default clause (containing a single expression).

```
switch ("Cat")
  case "Cow" return "Moo"
  case "Cat" return "Meow"
  case "Duck" return "Quack"
  default return "What's that odd noise?"
```

It returns the single expression's value that corresponds to the first case operand, called effective case, matching the switch operand. The switch expression in the above example will thus return `"Meow"`.

Typeswitch Expression

Typeswitch expression belongs to expressions on sequence types. It behaves as the switch expression presented in the previous section except that it chooses one of several expressions to evaluate based on the dynamic type of the input data. Here is an example of typeswitch:

```
typeswitch (<a/>)
  case $a as xs:integer return "integer"
  case $a as xs:string return "string"
  case $a as node() return "node"
  default return "unknown"
```

In this case, the above typeswitch expression returns "node".

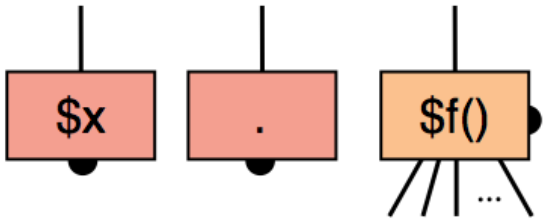


Figure A.2: Complex Expressions

Constructors

Constructors are used to create XML structures within a query. There are constructors for all seven kinds of XML nodes: element, attribute, document, text, comment, namespace, and processing instruction. Two kinds of constructors are provided: *direct constructors*, which use an XML-like notation, and *computed constructors*, which use a notation based on enclosed expressions. The following code show an example direct and computed constructors, both creating the same example.

```
<!-- A great book -->
<book isbn="$isbn">
  <title>Julius Caesar</title>
  <author>William Shakespeare</author>
</book>
```

```
comment { "A great book" },
element book {
  attribute isbn { $isbn },
  element title { "Julius Caesar" },
  element author { "William Shakespeare" }
}
```

Direct constructors are classified as N-ary expressions because they can contain any number of operands, enclosed by curly braces `{ }` which delimit enclosed expressions, distinguishing them from literal text. Enclosed expressions are evaluated and replaced by their value dynamically to produce XML structures on the fly. Computed constructors on the other hand could be subdivided more accurately into unary and binary expressions: document, text, and comment node constructors are actually unary expressions, whereas element, attribute, namespace, and processing-instruction node constructors are binary expressions.

A.2 Complex expression

A complex expression is an expression that has bindable expressions as operands (or accesses a bindable expression after getting a reference thereto from an input) and evaluates them by binding values dynamically. Such expressions are typically used when the values of bindable expressions is changing at run-time, for example, in a loop.

Figure A.2 illustrates the different complex expressions. As opposed to simple expressions, we do not take arity into account to sub-classify complex expressions. Instead, we only distinguish between primitive bindable expressions (having exactly one free variable) and expressions using or consisting of bindable expressions (having one or more free variables and possibly other simple input expressions). Bindable expressions are represented by semicircles.

A.2.1 Primitive Free-Variable Expression

A primitive bindable expression is an expression that requires a single variable binding to occur before it can be evaluated.

Variable Expression

Variable expressions such as `$x` can only be evaluated if the variable `$x` has been bound to a value. If no value has being assigned when evaluated, an error occurs.

Context Item Expression

Context item expressions `.` and `..` represent the current item and its parent respectively. Their reference (bound item) is likely to change at run-time. As for variable expressions, the context item `.` can only be evaluated if the context item has been bound to an item. If no item is bound to the expression, an error is returned.

A.2.2 Expressions with Free-variable Expressions

Potentially all simple expressions can become complex expressions as soon as one replaces one of their operands by a free variable. However, there are some complex expressions that do not have a simple counterpart, i.e. these expressions cannot be used without free-variable expressions. Qualified, path, as well as FLWOR expressions are such kind of complex expressions.

Dynamic Function Calls

In XQuery 3.0, a dynamic function call consists of a base expression that returns the reference to a function and a parenthesized list of zero or more arguments, both being its operands. Once invoked, it is evaluated as a static function call. The non-local variable bindings returned by the base expression as well as the parameter results are fed into the function body (referenced by the function). The function body expression is then evaluated and the result item forwarded.

```
1  let $f := math:add#2
2  return $f(1,2)
```

In this case the function reference is obtained from variable `$f`, but any primary expression could be used instead. Thus, evaluating `$f(1,2)` gives `3`.

Filter Expressions

A filter expression consists of a base expression followed by a predicate enclosed by square brackets. The items returned by the base expression are filtered by applying the predicate to each item in turn. The ordering of the items returned by a filter expression is preserved. For example, the following expression returns all integers from 1 to 10 that are divisible by 2:

```
(1 to 10)[( . mod 2) eq 0]
```

The predicate is evaluated ten times, once for each item. Every time the context item (which can be seen as a special kind of variable) is bound to one integer produced by the base expression.

Any filter expression can be reduced to a FLWOR expression. Hence, the above example have the following FLWOR counterpart.

```
for $x in 1 to 10
where ($x mod 2) eq 0
return $x
```

Note that the difference between these two expressions lies in the (implicit or explicit) binding of each item. In the FLWOR expression, the context item is replaced by variable `$x`.

Path Expressions

A path expression can be used to locate nodes within trees. It consists of a series of one or more steps separated by a path operator ("/" or "//") with an initial step that may begin with a path operator itself. Each step is either a Postfix expression (i.e. filter expression or dynamic function call) or an axis step. Actually, every relative (non-initial) path expression is binary operator on step expressions.

This sequence of steps is then evaluated from left to right. Hence, when evaluating *Expr1/Expr2*, each item produced by *Expr1* is used as context item to evaluate *Expr2*. If this expression had a third step, items returned by *Expr2* would have been used to evaluate the next expression.

In the following example,

```
(<a/>, <b/>)/self::a
```

the operand `self::a` is evaluated twice (once for each element returned by the first step): each time the context item is bound to one of `<a/>`, `<b/>`, and since only `<a/>` satisfies the axis step, this element is returned.

Simple Map Operator

The simple map operator "!" behaves mostly like the path operator "/", but with some key differences. A path operator can only be applied to a sequence of nodes, which means that once a path step returns simple values, no path operator can be further applied. Moreover, path operators does additional processing implicitly: the result is sorted into document order regardless of the input order and all duplicate nodes are removed.

The simple map operator does not have these limitations. Both its left-hand side operand and its right-hand-side operand can be any sequence of items, and the original order is preserved. The simple map operator can be seen as a binary operator taking two expressions as its operands. It behaves like a for clause in FLWOR expression: the right-hand side is evaluated once for every item in the sequence returned by the left-hand side.

Any expression using simple map operators can be rewritten as a FLWOR expression. For example, a simple map operation multiplying every number between 1 and 100 by 2 and concatenating each result item with a sharp character

```
(1 to 100) ! (. * 2) ! ("#" || .)
```

can be reduced to the following FLWOR expression:

```
for $x in (1 to 100)
let $y := $x * 2
return "#" || $y
```

Of course, FLWOR expressions (presented later in this section) are still necessary for various reasons such as sorting, local variable bindings, or joins.

Quantified Expressions

Quantified expressions are used to test if any or all input items match a certain condition. The value returned by quantified expressions is always true or false. Quantified expressions support existential and universal quantification ($\exists$ and $\forall$ respectively):

- Existential quantifier (**some**): Given two item sequences, if any item in the first sequence has a match in the second sequence, the expression returns true. Otherwise, if no item satisfies the comparison, the result is false.
- Universal quantifier (**every**): Given two item sequences, if every item in the first sequence has a match in the second sequence, the returned value is true. Otherwise, if only one item does not satisfy the comparison, the result is false.

For example, in this expression

```
some $x in (1, 2) satisfies $x le 2
```

the operand `$x le 2` is evaluated twice, each time the variable `$x` is bound to one of (1, 2). The result returned by the above expression is true.

A quantified expression has a format very similar to a FLWOR expression and can also be rewritten as such using the built-in function `fn:exists`:

```
1 fn:exists(
2   for $x in (1, 2)
3   where $x le 2
4   return $x
5 )
```

FLWOR Expressions

The FLWOR expression is a flexible, multipurpose expression that is composed of multiple clauses. It can be used for different tasks such as iterating over sequence, joining multiple documents, or grouping and aggregating some query results. FLWOR expressions are analogous to SQL statements (select, from, where, etc.) and provide join-like functionality to XML documents in the same way as SQL can be used to join SQL tables. *FLWOR* is an acronym for its clauses (*for*, *let*, *where*, *order by*, and *return*) available in XQuery 1.0. However, in XQuery 3.0, three new clauses were added, namely *window*, *count*, and *group by*.

The evaluation of FLWOR expressions is different from other XQuery expressions. Its semantics is based on the concept of *tuple stream*, i.e. an ordered sequence of many tuples, where a *tuple* is defined in the XQuery specification [29] as being a set of multiple named variables, each bound to a XDM instance. An individual tuple steam is uniform, meaning that all its tuples contain variables with the same names and static types. Every FLWOR expression generates its own tuple stream, which is then successively modified by its clauses. Therefore, each clause consumes an input tuple stream produced by the previous clause and produces a new output tuple stream. Only the first clause does not take any tuple stream as input, but simply creates a new one.

A FLWOR expression starts with an initial clause, followed by an indefinite number of intermediate clauses and ends with a final (return) clause. An initial clause is the first expression of a FLWOR expression, it is either a for, window or let clause. The initial clause generates a sequence of bound variables, i.e. a tuple stream, that will be used as input by the next clauses. An intermediate clause can be any possible clause (except *return*). There can be zero or more intermediate clauses inside a FLWOR expression. The final clause is always a return clause. A detailed description about how each clause behaves is given in the remaining of this section.

For: The *for* clause is used for iterating over an input sequence. Each variable in a for clause is bound in turn to each item present in the sequence. If the clause contains more than one variable bindings, it has the same semantics as if multiple for clauses were used instead. In this clause, a tuple represents a single item from the sequence.

```
for $x at $i in (2,4,6,8,10), $y at $j in (1 to $x)
```

Let: The *let* clause declares one or more variables and assign (bind) a value to each of them.

```
let $a := "foo", $b := "bar"
```

Window: The *window* clause (as the for clause) iterates over its binding sequence and generate a sequence of tuples. However, a tuple now represents a whole *window*, where a window is a sequence of consecutive items draw from the binding sequence. A window may be defined by one to nine bound variable together with a start and end condition. There are two kinds of windows: *trumbling windows* that do not overlap (i.e. an item can occur only once in exactly one window), and *sliding windows* that may overlap (i.e. an item can appear in diffeent windows).

```
for tumbling window $w in (1 to 20)
  start $s at $spos previous $sprev next $snext
    when $s gt 5
  only end $e at $epos previous $eprev next $enext
    when $e - $s eq 2
```

Where: The *where* clause acts as a filter on tuples from previous clauses. It tests the tuples against a constraint (evaluated once for each of them). If a tuple satisfies the condition, it is retained in the output tuple stream; otherwise, it is discarded.

```
where ($i mod 2 = 0) and ($j mod 5 = 0)
```

Count: The *count* clause simply enhances the tuple stream with a new variable that is bound to the original position of each tuple.

```
count $counter
```

Group-By: The *group-by* clause modifies the tuple stream in which each tuple now represents a group of tuples (from the input stream) that have the same grouping keys. A group-by clause can have multiple grouping keys wich are then applied to the tuple stream from left the right.

```
group by $category, $language
```

Order-By: The *order-by* clause is used to enforce a certain ordering on the tuple stream. Thus, the input and output streams contain the same tuples, but eventually in a different order. Multiple sort keys (separated by commas) can be specified in either ascending or descending order.

```
order by $prize descending
```

Return: The *return* clause is required at the end of every FLWOR expression and occurs only once. This clause is evaluated once for each input tuple and the results are concatenated to produce an XDM instance, i.e. a sequence of items, which is the final result of the FLWOR expression.

```
return <item>{ $x }</item>
```

To conclude this section, we give an example of how FLWOR expression can be used:

```
1 for $x in (1 to 9)
2 let $m := $x mod 3
3 group by $m
4 return <numbers>{ $x }</numbers>
```

In the above example, the operand `$x` is evaluated nine times, each time the variable `$x` is bound to an integer between 0 and 9. Next, the let clause computes the modulo of base 3 of `$x`. The group by clause then groups the numbers based on their remainder `$m`. Finally, the return clause produces an element for each of these groups. This gives the following result sequence:

```
1 <numbers>3 6 9</numbers>
2 <numbers>1 4 7</numbers>
3 <numbers>2 5 8</numbers>
```

Appendix B

Implementation

B.1 Concrete Job Object

In this section, we present a real job object that was retrieve from the scheduler queue during the bulkload experiment.

In addition to the fields presented in chapter 5, there are two other fields that are automatically added by MongoDB, namely a primary key ("_id") and an index build on "uuid" ("_28"). These fields are not used by the parallel infrastructure, but one could typically replace the "uuid" by "_id" in a future release and, thus, remove the entire index.

```
1  {
2    (: UUID automatically created by MongoDB :)
3    "_id" : ObjectId("5139f6722abe5bf9a110a88d"),
4    (: Index on "uuid" :)
5    "_28" : {
6        "i0012" : [
7            { "k00" : "263b96fd-9dc6-46a2-ba0e-efca1e1eed74",
8              "k00_o" : "263b96fd-9dc6-46a2-ba0e-efca1e1eed74" }
9          ]
10     },
11
12     (: UUID used by the framework :)
13     "uuid" : "263b96fd-9dc6-46a2-ba0e-efca1e1eed74",
14     (: Name of the input collection,
15        may contain some chunk information :)
16     "input" : {
17         "collection" : "e3434650-0bb3-43fe-84b9-8f2f10ec0dee"
18     },
19     (: Name of the output collection :)
20     "output" : {
21         "collection" : "orders"
22     },
23     (: Name of the user app. that initiates the parallel query :)
24     "application" : {
25         "name" : "tpox"
26     },
27     (: Function containing the user query,
28        to be run by daemon processes :)
29     "function" : {
```

```
30          "name" : "Q{http://www.28msec.com/benchmark/tpox/workers/
               bulkload}load"
31      },
32
33      (: Job properties used for asyn. communication and locking :)
34      "properties" : {
35          (: Last time a daemon "touched" the job object :)
36          "accessed" : ISODate("2013-03-08T14:33:58Z"),
37          (: Time when job was created :)
38          "creation" : ISODate("2013-03-08T14:32:18.885Z"),
39          (: Daemon currently holding the lock :)
40          "lock" : "daemon1",
41          (: Time when the lock was acquired :)
42          "lock-acquired" : ISODate("2013-03-08T14:32:18.885Z"),
43          (: Current job status :)
44          "status" : "in-progress"
45      },
46  }
```

Note that the comments inside the job object are just here for illustration purposes and are not part of the physical object.

List of Figures

List of Tables

Bibliography

[1] D.J. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *Comm. ACM*, New York, USA, 1992.

[2] H. Pirahesh, C. Mohan, J. Cheng, T.S. Liu, and P. Selinger, "Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches", *Proc. Second Symposium Databases in Parallel and Distributed Systems*, New York, USA, 1990.

[3] J.P. Richardson, H. Lu, K. Mikkilineni, "Design and Evaluation of Parallel Pipelined Join Algorithms", *Proc. ACM SIGMOD*, New York, USA, 1987.

[4] K.A. Hua, C. Lee, and J.K. Peir "A High Performance Hybrid Architecture for Concurrent Query Execution", *Proc. 2nd IEEE Symp. Parallel and Distributed Processing*, Washington, USA, 1990.

[5] J. Menon, "A Study of Sort Algorithms for Multiprocessor Database Machines", *Proc. 12th VLDB Conf.*, San Francisco, USA, 1986.

[6] D.J. DeWitt, J.F. Naughton, and D.A. Schneider, "Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting", *Proc. First Int. Conf. Parallel and Distributed Information System*, Madison, USA, 1991.

[7] R.A. Lorie, H. C. Young, "A Low Communication Sort Algorithm for a Parallel Database Machine", *Proc. 15th VLDB Conf.*, San Francisco, USA, 1989.

[8] P. Valduriez and G. Gardarin, "Join and Semi-Join Algorithms for a Multiprocessor Database Machine", *ACM. Trans. Database Systems*, 1984.

[9] M.S. Lakshmi and P.S. Yu, "Effectiveness of Parallel Joins", *IEEE Trans. Knowledge and Data Eng.*, Yorktown Heights, USA, 1990.

[10] K.A. Hua and C. Lee, "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning", *Proc. 17th VLDB Conf.*, San Francisco, USA, 1991.

[11] H. Lu and K.-L. Tan, "Dynamic and Load-Balanced Task-Oriented Database Query Processing in Parallel Systems", *Proc. 3rd EDBT Conf.*, London, UK, 1992.

[12] G. von Bültzingsloewen, "Optimizing SQL Queries for Parallel Execution", *ACM SIGMOD Record*, New York, USA, 1989.

[13] M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Architecture and Performance of Relational Algebra Machine GRACE", *Proc. Int. Conf. Parallel Processing*, Chicago, USA, 1984.

[14] D. Kossmann, "The State of the Art in Distributed Query Processing", *ACM Computing Surveys*, New York, USA, 2000.

[15] M. Yui , J. Miyazaki, S. Uemura, and Hirokazu Kato, "XBird/D: Distributed and Parallel XQuery Processing using Remote Proxy", *Symp. Applied Computing*, Fortaleza, Brazil, 2008.

[16] Y. Zhang and P. Boncz, "XRPC: Interoperable and Efficient Distributed XQuery", *VLDB*, Vienna, Austria, 2007.

[17] M. Fernandez, T. Jim, and N. Onose, "Highly Distributed XQuery with DXQ", *SIGMOD*, Beijing, China, 2007.

[18] M. Fernandez, N. Onose, T. Jim, J.Simeon, and K. Morton, "DXQ: A Distributed XQuery Scripting Language", *4th Int. Workshop XQuery Impl.*, Beijing, China, 2007.

[19] S. Khatchadourian, M.P. Consens, and J Simeon, "Having a ChuQL at XML on the Cloud", *Proc. Conf. CASCON*, Riverton, USA, 2011.

[20] L. Fegaras, C. Li, U. Gupta, and J.J. Philip, "XML Query Optimization in Map-Reduce" *14th Int. Workshop Web Databases*, Athens, Greece, 2011

[21] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", *Procc. OSDI*, San Francisco, USA, 2004.

[22] A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis "Dremel: Interactive Analysis of Web-Scale Datasets", Proceedings of the VLDB Endowment, Singapore, 2010.

[23] S. Alsubaiee, A. Behm, R. Grover, R. V. V. Borkar, M. J. Carey, C. Li, "ASTERIX: Scalable Warehouse-Style Web Data Integration", *IIWeb*, Scottsdale, USA, 2012.

[24] Clustrix Inc., "Driving the New Wave of High Performance Applications", http://www.clustrix.com, San Francisco, USA, 2012.

[25] Alexander S. Szalay, Jose A. Blakeley, "Gray's Laws: Database-centric Computing in Science", *The Fourth Paradigm (Part 1)*, Microsoft Research, Washington, October 2009.

[26] Duncan Mackenzie, "Architectural Options for Asynchronous Workflow", Microsoft Developer Network, December 2001.

[27] Russ Miller, Laurence Boxer, "Algorithms Sequential and Parallel: A Unified Approach", Chapter 9 (Partition Sort), Second Edition, Course Technology PTR, August 2005.

[28] R.T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures" (Chapter 5), PhD Thesis, University of California, USA, 2000.

[29] XQuery 3.0: An XML Query Language Specification, http://www.w3.org/TR/xquery-30/, Candidate Recommendation as of January 2013.

[30] XPath and XQuery Functions and Operators 3.0 Specification, http://www.w3.org/TR/xpath-functions-30/, Candidate Recommendation as of January 2013.

[31] XQuery Update Facility 1.0, http://www.w3.org/TR/xquery-update-10/, W3C Recommendation as of March 2011.

[32] XQuery Scripting Extension 1.0, http://www.w3.org/TR/xquery-sx-10/, W3C Working Draft as of April 2010.

[33] Xadoop, http://www.xadoop.org/, December 2012.

[34] Google BigQuery, http://developers.google.com/bigquery/, October 2012.

[35] Hadoop, "The Hadoop Distributed File System: Architecture and Design", http://hadoop.apache.org/docs, December 2012.

[36] 28msec Inc., 28.io Platform, http://www.28msec.com/, September 2012.

[37] 28msec Inc., 28.io Platform, http://www.28msec.com/documentation/, October 2012.

[38] Zorba, http://www.zorba-xquery.com/, September 2012.

[39] XQuery Scripting Extension Proposal, http://www.zorba-xquery.com/html/scripting_spec.html, January 2013.

[40] J. Robie, G. Fourny, M. Brantner, D. Florescu, T. Westmann, and M. Zaharioudakis, "JSONiq Specification", version 0.4, http://www.jsoniq.org/, January 2013.

[41] J. Robie, M. Brantner, D. Florescu, G. Fourny, and T. Westmann "JSONiq: XQuery for JSON, JSON for XQuery", *XMLPrague Conf. Proc.* Prague, Czech Republic, 2012.

[42] MongoDB, http.//mongodb.org/, October 2012.

[43] MongoDB, Aggregation Framework, http://docs.mongodb.org/manual/applications/aggregation/, October 2012.

[44] MongoDB, Sharding, http://docs.mongodb.org/manual/sharding/, January 2013.

[45] BSON Specification, http://bsonspec.org/, December 2012.

[46] M. Nicola, I. Kogan, and B. Schiefer, "An XML Transaction Processing Benchmark", *SIGMOD*, Beijing, China, 2007.

[47] M. Nicola, A. Gonzales, K. Xie, and B. Schiefer, "XML Database Benchmark: Transaction Processing over XML (TPoX)", http://tpox.sourceforge.net/, March 2013.

[48] Financial Information eXchange (FIX) Protocol, http://www.fixprotocol.org, March 2013.

[49] Amazon Web Services (AWS), http://aws.amazon.com/en/ec2/, March 2013.