TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

A Serialization Framework for XQuery

by

Roxana Zapata Gomez

in partial fulfillment of the requirements for the degree of
Master of Applied Computer Sciences

Supervisors:
Dipl.-Inf. Sebastian Bächle
M. Sc. Caetano Sauer
Prof. Dr.-Ing. Dr.h.c. Theo Härder

Submitted to:
Department of Computer Science
University of Kaiserslautern

August 31, 2012

Abstract

This thesis addresses the problem of management of large intermediate results in the context of XQuery, a language for processing XML data. This work proposes an XQuery serialization framework for enabling the management of large data during the execution of a query. Based on this framework, we present two applications. *BufferedSequence* manages large sequences which result from the evaluation of XQuery expressions. *TupleSort* is an implementation of an external sorting algorithm for large tuple streams.

Zuzammenfassung

Diese Masterarbeit beschäftigt sich mit dem Problem des Managements großer Zwischenresultate die bei XQuery, einer XML-Verarbeitungssprache, anfallen. In dieser Arbeit wird ein Framework für die Serialisierung von XQuery Resultaten vorgestellt, um die Verarbeitung großer Datenmengen bei der Ausführungung einer Anfrage zu ermöglichen. Aufbauend auf diesem Framework werden zwei Anwendungen vorgestellt: BufferedSequence verwaltet lange Sequenzen, die bei der Ausführung von XQuery-Ausdrücken enstehen. TupleSort ist die Implementierung eines externen Sortier-Operators, der lange Tupelströme unterstützt.

Ich versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema „A Serialization Framework for XQuery“ selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

Kaiserslautern, den 31.08.2012 Roxana Zapata Gomez

For my parents

Acknowledgements

I would like to thank Professor Theo Härder for giving me the opportunity of being part in his research group and for funding my master studies. I would like also to express my sincerest gratitude to Sebastian Bächle for his support and guidance during my research on Full Text Indexing in XML documents and on my master thesis. Furthermore, I would like to thank Caetano Sauer for supporting me and helping me in every detail of this work.

Most importantly, I would like to thank to my parents for their support and encouragement. I wish to thank my entire extended family for providing a loving environment for me. Finally, I would like to thank all friends who gave me an excellent time in Kaiserslautern and made me feel right at home.

Contents

Chapter 1

Introduction

1.1 Motivation

Effective and efficient management of large data volumes is necessary in all computer applications, from business data processing to library information retrieval systems, multimedia applications, computer-aided design and manufacturing, real time process control, and scientific computation. In the context of data management, “large” means that it cannot fit into main memory. Thus, it must reside on external storage and be brought into main memory selectively for processing. The management of large documents in XQuery–the query language for XML data–is more complex than in relational databases. Firstly, XQuery supports a nested data model which causes more difficulties for storing and retrieving it from external disk. Secondly, XQuery supports the processing of data from different sources such as database systems or documents in a file system, which requires a generic processing engine.

The Extensible Markup Language (XML) is by now the *de facto* standard format for exporting and exchanging data across systems, departments and network boundaries, due to its flexibility. First, it dissociates schema from data. In this way, data can exist without a schema, and data from legacy applications or archived can be processed. Secondly, it is able to represent a large spectrum of data, from totally unstructured, semi-structured, to totally structured data. As increasing amounts of information are stored, exchanged, and presented using XML, it becomes increasingly important to effectively and efficiently query XML data sources.

XQuery is a query language designed by the W3C [2] to address these needs. XQuery natively speaks XML, and it has convenient primitives to formulate queries using a syntax similar to SQL, in addition to support for XML-specific operations such as path navigation. Furthermore, it has been extended by a number of additional features which go beyond message transformation and XML query processing, for which XQuery was initially designed. XQuery Update Facility [11] and XQuery FullText [12] have been devised and have reached recommendation status by the W3C. Under development are still the XQuery Scripting Facilities [13]. With all these extensions, XQuery is much more than merely a query language; it has become an extremely powerful tool for developing almost any kind of data processing application.

The performance of XQuery in terms of effectiveness and efficiency relies on the XQuery processor architecture and the storage hierarchy context. The architecture of a processor is up to implementation decisions of its designers. On the other hand, the context of the storage hierarchy–a natural structure of computer architecture, given the set of technologies such as main memory and hard disk; and their access speed, capacity and cost characteristics–influences the management of large XML documents. Firstly, main memory access is much faster than access to data stored on disk. In fact, the relative difference in access time is at least 1000 times. Secondly, main memory capacities are much smaller than disk. Thus, the use of external disk is necessary. Thirdly, main memory is expensive in terms of price per megabytes, while disks are cheaper. Due to these differences, efficient algorithms for accessing and manipulating large XML documents are required to store intermediate results in external memory, while still providing acceptable performance.

1.2 Contribution

We propose a Serialization Framework which is integrated into an existing main-memory query processor, namely the Brackit Engine. The Serialization Framework addresses the issue of management of large XML documents between main memory and external disk in the XQuery context. Our framework is *compact* to make efficient use of the storage space, *fast* so the overhead of reading and writing megabytes of data is minimal, and *extensible* so we can transparently read and write data using different encoding schemes besides the one we propose. Furthermore, the framework is abstracted from the execution engine in which the implementation of our approach does not modify the manner how the execution engine works.

We use our framework in two different applications. In one application, it manages sequences that result from evaluating expressions, based on a component called *BufferedSequence*. It adds sequences to a buffer until a threshold, which is the maximum size assigned in main memory for holding sequences, is reached. Once it is reached, the buffered set of sequences is written into a temporary file using the Serialization Framework.

In the second application, our framework is used during the internal evaluation of FLWOR expressions using the special sort operator that implements external sorting. The sort operator has been extended to support external sorting using file-based sort and merge techniques. In addition, we have implemented optimization techniques to allow comparison operations directly on the encoded binary format.

The remainder of this thesis is organized as follows. Chapter 2 introduces the XQuery language focusing on its data model. Moreover, we present the Brackit XQuery Engine, which is used for the implementation of our approach. Chapter 3 presents the Serialization Framework. Chapter 4 presents our BufferedSequence component. Chapter 5 presents the external sort operator. In chapter 6 we empirically assess the efficiency of the framework with experiments. Finally, Chapter 7 concludes this thesis.

Chapter 2

XQuery Process Architecture

2.1 XQuery

XQuery is a query language for XML data sources and it is designed to meet the requirements of the W3C XML Query Working Group [16] and the use cases in [17]. XQuery provides three kinds of flexibility. First, it operates on a broad spectrum of XML information sources such as relational databases and documents. Second, XQuery allows to process completely untyped data, which may be progressively improved with schema information, in a "data first-schema later" (or as pay as you go) approach. Thirdly, it is able to operate in a large spectrum of data–from totally unstructured, semi-structured, to totally structured data.

Furthermore, XQuery has convenient primitives to formulate SQL-like queries, allowing typical query operations on XML data sources, including: selecting information based on specific criteria, filtering out unwanted information, searching for information within a document or set of documents, joining data from multiple documents or collections of documents, sorting, grouping and aggregating data, and so on.

There are as many reasons to query XML as there are reasons to use XML. XQuery language can be used for: extracting information from a relational database for use in a web service, generating reports on data stored in a database for presentation on the Web as XHTML, searching textual documents in a native XML database and presenting the result, pulling data from databases and transforming it for application integration and others.

The following subsections give an overview of the XQuery language main constructs, focusing primarily on its data model, and FLWOR expressions. Rather than a detailed explanation, we provide a sample query which illustrates the basic features of XQuery. For a proper introduction, we refer to W3C XQuery standard [2].

2.1.1 Data Model

XQuery is defined in terms of the *XQuery and XPath Data Model* [3], or shortly, XDM. Every input and output of a query is an instance of the data model. These are represented as an ordered *sequence*–an ordered list of zero, one or more *items*–and there is no distinction between a sequence of length one and the individual item it contains. This definition forbids nested sequences, i.e., sequences that contain another sequence as one of its items, which means that some operations must perform implicit unnesting. An item is a generic term that refers to either a node or an atomic value. The types of node and atomic values are defined by XML Schema [4], which is the standard that defines types in an XML documents.

Nodes are used to represent XML entities, and can be one of seven kinds: document, element, attribute, text, namespace, processing instruction, or comment. The first node in any document is the document node, which contains the entire document and contains its root node as child. The element nodes, comment nodes, and processing instructions nodes occur in the order in which they are found in the document. Element nodes occur before their children, i.e., the elements nodes, text nodes, comment nodes, and processing instructions they contain. Attributes are considered children of an element, but they have a defined position in the document order: they occur after the element in which they are found, before the children of the element. Text nodes contain plain character data of an element.

Every node has a unique node identity that distinguishes it from other nodes, even from other nodes that are otherwise identical. In addition to their identity, nodes have two kinds of values: string and typed. All nodes have a string value. The string value of an element is its character data content and all that of all its descendant elements concatenated together. The *string value* of an attribute node is simply the attribute value. Both element and attribute nodes have a typed value, too. They are taken into account if there is any. An element or attribute might have a particular type if it has been validated with a schema. If it is not declared in the schema, the type of the value is represented by the special type `xs:untypedAtomic`[4].

Atomic values are single values, with no markup, and no association with any particular element or attribute. An atomic value can have specific type, such as `xs:string` or `xs:integer`, or it can be untyped. Atomic values don´t have identity. It is not meaningful to ask whether 1 and 1 are the same integer or different integers; we can only ask whether they are equal.

Figure 2-1 illustrates a basic XML document taken from the XML Query Use Cases [5]. Its equivalent structure as an XDM instance is shown in Figure 2-2, where each node shape in the tree corresponds to an XDM node kind. The document represents bibliography data which contains a sequence of `book` elements.

```
<bib>
    <book year="1994">
       <title>TCP/IP Illustrated</title>

<author><last>Stevens</last><first>W.</first></author>
       <publisher>Addison-Wesley</publisher>
       <price>65.95</price>
    </book>
    <book year="1992">
       <title>Advanced Programming in the Unix
environment</title>

<author><last>Stevens</last><first>W.</first></author>
       <publisher>Addison-Wesley</publisher>
       <price>65.95</price>
    </book>
    <book year="2000">
       <title>Data on the Web</title>

<author><last>Abiteboul</last><first>Serge</first></author>

<author><last>Buneman</last><first>Peter</first></author>
       <author><last>Suciu</last><first>Dan</first></author>
       <publisher>Morgan Kaufmann Publishers</publisher>
       <price>39.95</price>
    </book>
    <book year="1999">
       <title>The Economics of Technology and Content for
     Digital
            TV</title>
       <editor>
            <last>Gerbarg</last><first>Darcy</first>
            <affiliation>CITI</affiliation>
       </editor>
       <publisher>Kluwer Academic Publishers</publisher>
       <price>129.95</price>
    </book>
</bib>
```

Figure 2-1: A sample XML document

2.1.2 Expressions

The expression is the basic unit of evaluation in the XQuery language. A query contains expressions that can be made up of a number of sub-expressions, which may themselves be composed from other sub-expressions. Every expression evaluates to a sequence, which may be a single atomic value, a single node, the empty sequence, or multiple atomic values and or/nodes. This section covers the most basic types of expressions, with a greater focus on FLWOR expressions.

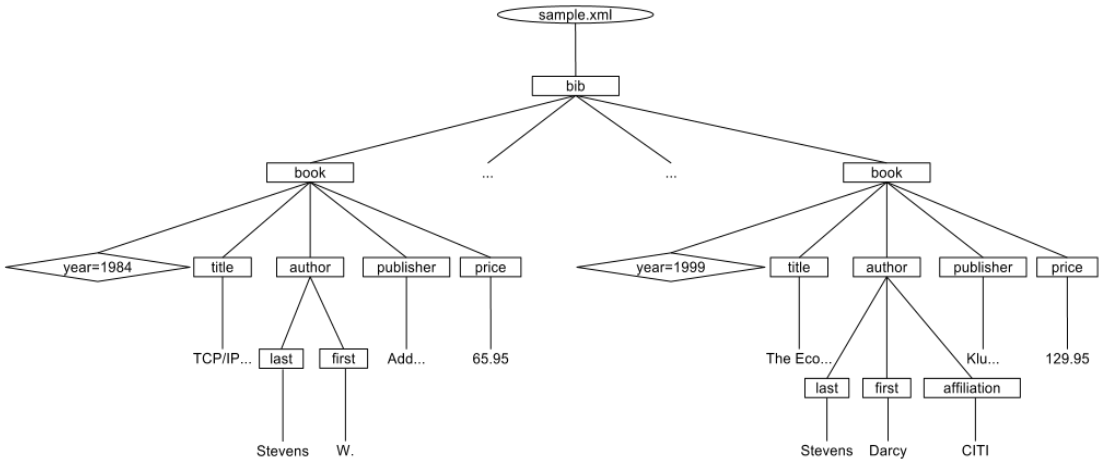


Figure 2-2: XDM instance for the sample document

2.1.2.1 FLWOR Expressions

FLWOR expressions are used to express data-intensive iterative computations, and hence may generate large amount of data during its evaluation. Therefore, they require great care for the data processing in main memory and external disk.

The acronym FLWOR stands for "`for`, `let`, `where`, `order by`, `return`," the *clauses* that are used in the expressions. The `group by` and `count` clauses were added later in the version 3.0 of the XQuery standard. In the scope of this work, only group by is considered. This kind of expression is often useful for computing joins between two or more documents and for restructuring data. A FLWOR expression starts with one or more `for` or `let` clauses in any order, followed by optional `where` clauses, optional `order by` clauses, optional `group by` clauses, and a required `return` clause. Figure 3-3 shows an example of FLWOR expression, which we use as basis for the following discussions.

The semantics of FLWOR expressions is given by a sequential evaluation of the clauses, where each clause consumes *tuples of bound variables* (which we refer to simply as *tuples*) as a context and produce one or more tuples as a context for next clause. Variables are denoted by the initial symbol $. At the beginning of the evaluation of a FLWOR expression, there is no variable bound, thus, the tuple is empty.

```
for $b in doc ('sample.xml')/bib/book
let $y := $b/@price
where $y > 50.00
group by $y
order by $y descending
return
     <result>
            <price>{$y}</price>
     <result>
```

Figure 2-3: Sample FLWOR expression

The `for` clause evaluates the expression given to its `in` parameter, where a tuple is generated for each item in the resulting sequence. The resulting tuple is then passed to the next clause and the process is repeated until all items in the sequence are consumed. In the example of Figure 2-3, the variable `$b` is bound to each book of the bibliography data, and the following clauses are evaluated once per each book.

The `let` clause binds a variable to the entire result of an expression. This bound variable is attached to the tuple. Note that the `let` clause does not perform any iteration. After the `for` and `let` clauses in the example of Figure 2-3 are evaluated, we have the following tuple stream: Here *b1*, …, *b4* represent the book elements in the document as XDM instances.

```
[$b = b1, $y =  "65.95"]

[$b = b2, $y =  "65.95"]

[$b = b3, $y =  "39.65"]

[$b = b4, $y = "129.95"]
```

The `where` clause filters out the tuples that are generated from the `for` and `let` clauses if the evaluation of a given expression returns the boolean value *False*. In our example, the following tuple stream remains after the where `clause`:

```
[$b = b1, $y =  "65.95"]

[$b = b2, $y =  "65.95"]

[$b = b4, $y = "129.95"]
```

The `group by` clause groups all tuples that have the same value in a given variable, grouping the other variables in a sequence. Its argument must be a variable reference. The `group by` clause also supports multiple variable references, which causes groups to be formed according to distinct values on every given variable. After the `group by` clause is evaluated, we have the following tuple stream:

```
[$b = (b1, b2), $y = "65.95"]

[$b = b4, $y = "129.95"]
```

The `order by` clause reorders the tuple stream according to the expression given, which must be an atomic value (also in `group by`). In our example, the two remaining tuples in the stream are swapped, so that the tuple with price "129.25" comes before the one with the price "65.95".

Finally, the `return` clause finishes the evaluation of the FLWOR expression and produces a final result by evaluating the given expressions for each tuple in the stream. Because of the unnesting semantics of XQuery, if the expression results in multiple-item sequences, these are concatenated to form a single sequence which is the result of the whole FLWOR expression. In our example, the return `clause` is given a *constructor* expression, which produces nodes as a result or creates nodes which do not necessarily originate from existing ones. The result of our example is shown in Figure 2-4.

```
<result>
        <price>129.95</price>
<result>
<result>
        <price>65.95</price>
<result>
```

Figure 2-4: Result of the sample query

Path expressions are used to navigate input documents to select nodes of interest. According to the XQuery semantics, they are normalized into FLWOR expressions. The return nodes of path expressions are in document order, defined in [3]. A path expression is made up of one more steps that are separated by a slash (/) or double slashes (//). A path expression is always evaluated to a particular *context* *item*, which serves as the starting point for the relative path. When a context item is a node, it is called the *context node*.

The context item change with each step. A step returns a sequence of zero, one or more nodes that serves as the context items for evaluating the next step. For example in `doc("sample.xml")/bib/book/title`, the `doc("sample.xml")` step returns a document node that serves as the context item when evaluating the `bib` step. The `bib` step is evaluated using the document node as the current context node, returning a sequence of one `bib` element child of the document node. This `bib` element then serves as the context node for evaluation of the `book` step, which returns the sequence of `book` children of `bib`. The final step, `title`, is evaluated in turn for each book child in this sequence.

Additionally, steps may contain predicates which filter the results to contain only nodes that meet specific criteria. Using predicates, we can select only elements that have a certain value for an attribute or child element, we can select only elements that have a particular attribute child element, or elements that occur in particular position within their parent. Figures 2-5 illustrates how a path expression is normalized into a FLWOR expression.

```
doc('sample.xml')/bib/book

           ⬇

for $b in doc('sample.xml')
return
  (
   for $c in child ($b)
   return
         child($c)
   )
```

Figure 2-5: Normalization of Path expression into FLWOR expression

In the scope of this thesis, we focus on FLWOR expressions that use the sort, group by, or join operators. These operators are known as blocking operators, meaning they consume the entire input before producing any output. As we mentioned in the introduction, the limitations that main memory has with respect to its capacity render that sorting, grouping or join must use external disk.

2.1.2.2 Other expressions

Primary expressions are the basic primitives of the language. They include literals, variable references, context item expressions, constructors, and function calls. A primary expression may also be created by enclosing any expression in parentheses, which is sometimes helpful in controlling the precedence of operators. Arithmetic expressions allow values to be added, subtracted, multiplied, and divided. Comparison expressions allow two values to be compared. XQuery provides three kinds of comparison expressions, called value comparisons, general comparisons, and node comparisons. A logical expression is either an `and-expression` or an `or-expression`. If a logical expression does not raise an error, its value is always one of the boolean values `true` or `false`. Construct expressions creates XML structures within a query. The ordered and unordered expressions set the ordering mode in the *static context* to ordered or unordered for a certain region in a query. The conditional expressions are based on the keywords `if`, `then`, and `else`. The processing of these expressions can be found in [2].

2.2 XQuery Processor Architecture

Brackit [6] is an implementation of XQuery written in Java. It uses a hybrid execution mechanism, as part of a database system (native XML database or XML-relational hybrid) and as standalone interpreter for small XML files. This hybrid solution provides efficiency comparable to the top performers in both usage scenarios.

In the scope of the thesis, we integrate our Serialization Framework into the main-memory query processor, the Brackit Engine. Our framework is abstracted from the execution engine. The following subsections present the compilation process and FLWOR pipelines in the Brackit Engine.

2.2.1 Compilation Process

Figures 2-6 gives an overview of the compilation process in Brackit. At the top, the Parser module generates an Abstract Syntax Tree, short AST, which is used during the compilation phases as a logical query representation. At the end of the compilation, the AST is passed to the Translator module, which generates a tree of executable physical operators. The Compiler module is constituted by the steps in between the Parser and Translator modules.

The first phase of the Compiler module is Analysis. Its goal is to perform static typing and annotate expressions with typing information, as well as to perform simple rewrites such as constant folding and introduction of let bindings that simplify optimization. The next phase is then Pipelining, which transform FLWOR expressions into pipelines- the internal, data-flow-oriented representation of

FLWORs, which we discuss in Section 2.1.2.1 Pipelines are subject to several optimization rules (such as push-downs, join recognition, unnesting, etc.), which are applied to the Optimization phase.

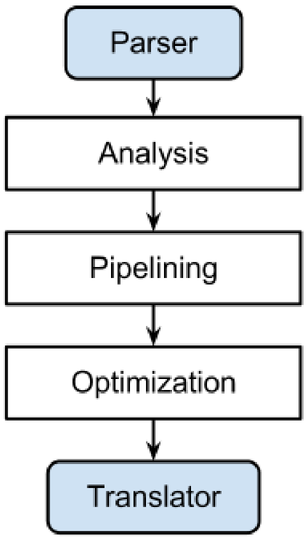


Figure 2-6: Overview of compilation process

2.2.2 FLWOR Pipeline

FLWOR pipelines are generated from a FLWOR expression tree, during the pipelining phase of the compilation process. The FLWOR expression tree is represented by the AST node `FlworExpr`. This node contains children that correspond to clauses that are present in the FLWOR expression. In the pipeline view, these clauses are translated as *operators*, where the operators are arranged in a top-down sequence. Figure 2-7 illustrates these two representations of the FLWOR expression.

To explain how FLWOR pipelines works, we make use of the example in 2-7. At the top, the `PipeExpr`, which represents the evaluation of the complete FLWOR expression, triggers the execution of the pipeline under it and receive the result from the last operator, which is always the `End` operator.

The execution of the pipeline starts with the `Start` operator, which creates an empty tuple that is fed to the next rightmost child operator in the pipeline. The following operators receive and produce tuples, in which the tuples are modified given the expressions they evaluate. The `End` operator is the exception in this process and represents the `return` clause. It does not produce tuples. It consumes the input tuples and generates a sequence as a result.

The operators in between the `Start` and `End` operator are `ForBind`, `LetBind`, `Select`, `GroupBy` and `Sort` operators which represent the `for`, `let`, `where`, `group by`, and `order by` clauses respectively. These operators consume and produce tuples as explained in Subsection 2.1.2.1. The pipeline representation resembles the physical plan, which follows the open-next-close model [18].

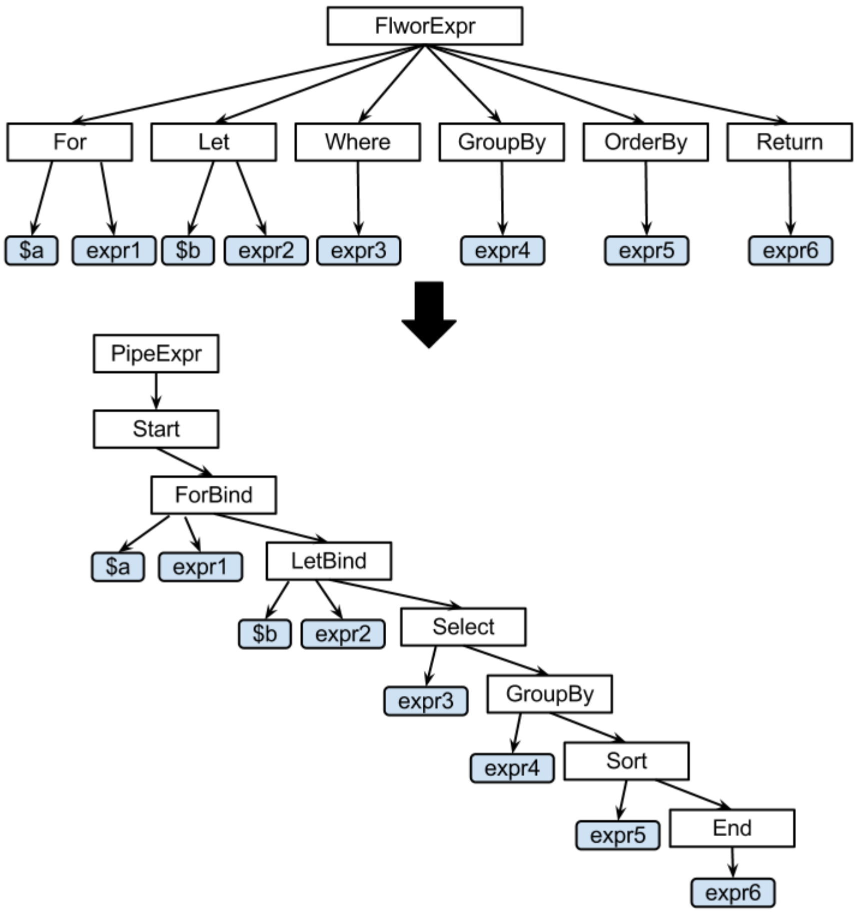


Figure 2-7: Translation of FLWOR expression tree into a pipeline

Chapter 3

XQuery Serialization Framework

3.1 Motivation

The processing of large data volumes requires writing intermediate results to external memory. The process of moving data from main memory to disk is called *serialization*, which transforms structured objects with pointers to memory addresses into a contiguous binary representation. The reverse process is called deserialization.

The manipulation of XDM instances between main memory and external disk for serialization and deserialization is supported by our XQuery Serialization Framework, shortly XSF. In this thesis, the assumed characteristics of XDM instances are as following. An item instance is associated with a type and a value. Values can be encoded with fixed-length or variable-length. Sequence instances are composed by zero or more items. The number of items that a sequence contains is not known in the beginning, because the expressions in our processor are evaluated in a lazy manner, computing individual items on demand. Tuples streams contain one or more sequences which are transferred between operators for further computation. The number of sequences that it contains is determined statically during the compilation process.

In the following section, we describe our encoding schemes in detail. We provide an encoding component for each structural level of XDM, namely Item, Sequence, and Tuple. Note that the higher-level components reutilize the lower-level ones to encode its nested values.

3.2 Item Encoding

The encoding schema for an item (see Figure 3-1) consists of a type identifier followed by its item type encoding. The type is either atomic or node. To encode the item´s value, we use *AtomicEncoding* when its item type belongs to atomic or *NodeEncoding* when its item type belongs to node.

The encoding structure for an atomic value is made up of a length field and the value payload. This encoding structure supports the storage of variable-length.

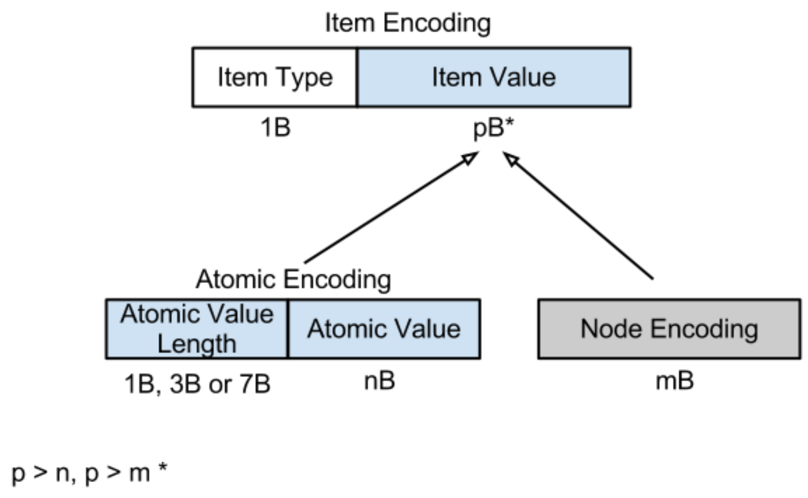


Figure 3-1: Item encoding schema

The string value of the atomic is encoded as an array of bytes. The size of the byte array–i.e., atomic value length–is encoded in additive growth manner. Algorithm 3-1 describes this encoding. If the size of the atomic value is less than 2^8-1 bytes, then the length field uses 1 byte (B_0) for its encoding. Otherwise, it uses 2 more bytes (B_1, B_2). In case the length of the field minus 2^8-1 is larger than 2^{16}-1, then we use four more bytes (B_3, B_4, B_5, B_6).

Algorithm 3-1 *getAtomicLength* of `AtomicEncoding`

```
1: function GETATOMICLENGTH (atomicValue)
2:     length ← atomicValue.length()
3:     if length < 255 then
4:         return 1-byte
5:     end if
6:     if length – 255< 65535 then
7:         1-byte ← 255
8:         2-bytes| 3-bytes ← length – 255
9:         return 1-byte| 2-bytes| 3-bytes
10:    end if
```

```
11:     if length – 25 – 65535>= 65535 then
12:             1-byte ← 255
13:             2-bytes| 3-bytes ← 65535
14:             return 4-bytes| 5-bytes| 4-bytes| 5-bytes ← length – 255 – 65535
15:     end if
16: end function
```

3.3 Sequence Encoding

The sequence encoding schema (see Figure 3-2) consists of a sequence type identifier followed by its sequence value. The sequence type identifier distinguishes a sequence of type *BufferedSequence*, a special sequence that we introduce in Chapter 4.

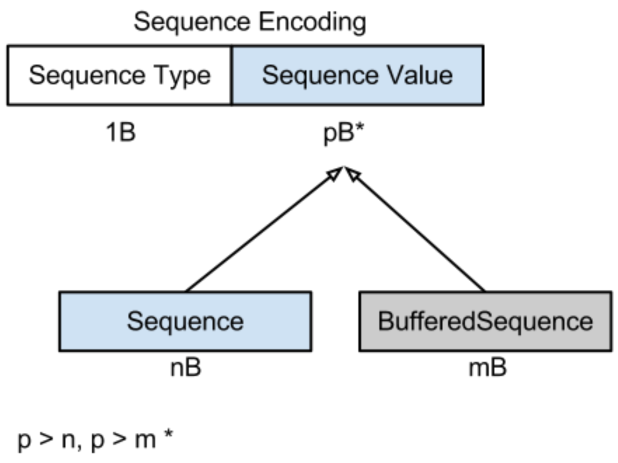


Figure 3-2: Sequence encoding schema

If the sequence is not a type of *BufferedSequence*, then the sequence value schema (see Figure 3-3) is encoded as following. It consists of one or more item length and item encoding followed by one sequence end. The item length fields are written as 1-byte, 3-bytes or 7-bytes integer(s), the setting of bytes follow the algorithm showed in Algorithm 4-1, which represents the length of the item encoded in bytes. The item fields follow the encoding structure explained in Section 3.2. The sequence-end field is represented by “0” (zero) which is written as a 1-byte integer. The purpose of having the sequence-end field is to determine the end of a sequence between a set of sequences that belong to a tuple when it is deserialized, a process explained in Section 3.4.

This encoding supports compact storage and flexibility. Compact storage space is achieved using efficient number of bytes for each field of the structure using the Algorithm 3-1. Flexibility is achieved as well because it allows the encoding of items as they are evaluated in a lazy manner.

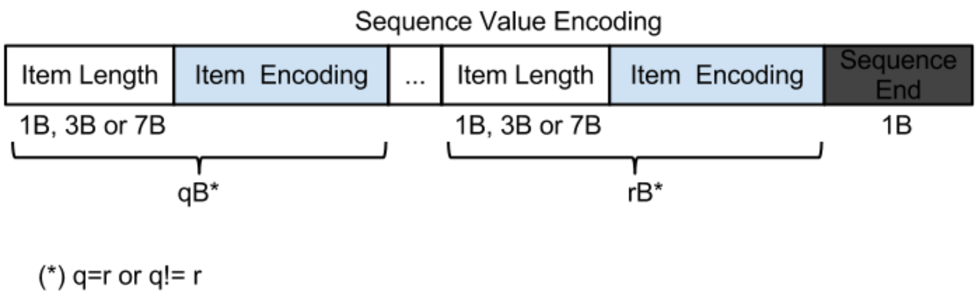


Figure 3-3: Sequence value encoding schema

If the sequence is a *BufferedSequence* instance, then it is encoded in a different manner. A *BufferedSequence* in our approach is associated with a file. Its encoding consists of the length of the file name followed by the file name. The length is written as 1-byte Integer, and the file name as n-byte String. Figure 3-4 illustrates this schema.

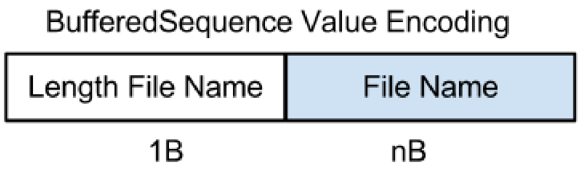


Figure 3-4: *BufferedSequence* value encoding schema

3.4 Tuple Encoding

The role that tuples play for transferring information across operators is really important, because tuple streams in blocking operators can become very large so that the use of external memory is necessary. Therefore, we need to provide compact and fast encoding of tuples. The tuple encoding schema (see Figure 3-5) consists of the number of sequences followed by one or more sequence encodings. The sequence encoding fields follow the encoding schema explained in Section 3.3.

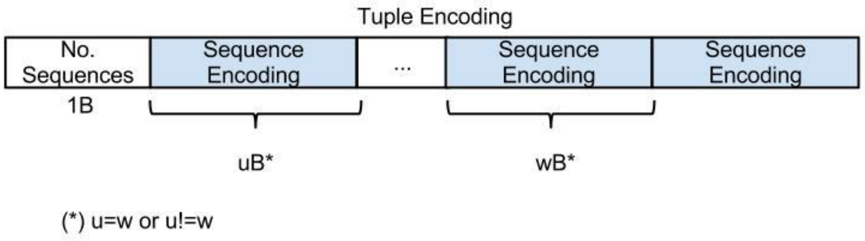
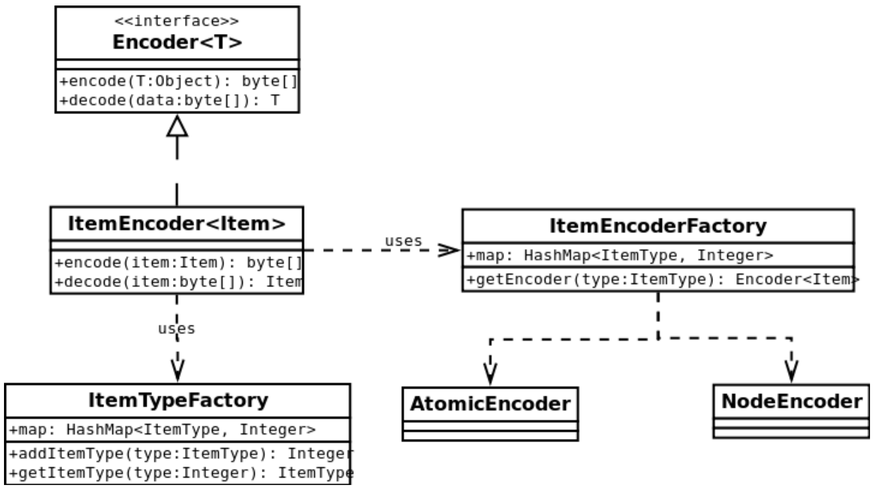


Figure 3-5: Tuple encoding schema

Figure 3-6 gives an overview of the proposed encoding schema. The interface `Encoder` has two functions called `encode` and `decode`. The `encode` function produces a byte array, while the `decode` function an object which is an abstract representation of the data. The interface explained above is implemented by the class `ItemEncoder`. It uses two other classes, `ItemTypeFactory` and `ItemEncoderFactory`. `ItemTypeFactory` maps an item type with an integer value, while the `ItemEncoderFactory` maps an encoding scheme that can be defined internally (our XEF) or externally to an integer value. When either item type or encoding type need to be referenced in Brackit, we use their integer value pair. `AtomicEncoder` and `NodeEncoder` are instantiated according to the type of the item to be encoded.



<<interface>>
Encoder<T>
+encode(T:Object): byte[]
+decode(data:byte[]): T
ItemEncoder<Item>
+encode(item:Item): byte[]
+decode(item:byte[]): Item
uses
ItemEncoderFactory
+map: HashMap<ItemType, Integer>
+getEncoder(type:ItemType): Encoder<Item>
uses
ItemTypeFactory
+map: HashMap<ItemType, Integer>
+addItemType(type:ItemType): Integer
+getItemType(type:Integer): ItemType
AtomicEncoder
NodeEncoder

Figures 3-6: Class Diagram of XEF

3.5 Implementation

As we mentioned in the introduction of this chapter, the XQuery Serialization Framework forms the basis for serializing and deserializing data generated from XQuery expression evaluations. The XSF is implemented using streams, which transports data from one point (e.g., main memory) to some other point (e.g., external memory). Java, the implementation language that we use to build XSF, has two fundamental components in the `java.io` package: `InputStream` class and `OutputStream`.

An `InputStream` is a reference to a source data sink (be it a file, network connection, etc), that we want to process as follows:

- We want to read the data as raw bytes [1] and write our own code encoding with the bytes.
- We want to read the data in a sequential order, that is, to get to the *n*th byte of data, we have to read all the preceding bytes first and we are not guaranteed to be able to jump back again once we have read them.

In order to support an abstract deserialization of data, we have defined three subclasses of the `InputStream` class: `ItemInputStream`, `SequenceInputStream` and `TupleInputStream`. The `ItemInputStream` class uses the item encoding schema for its deserialization. `SequenceInputStream` uses the sequence encoding schema. `TupleInputStream` uses the structure of the tuple encoding.

An `OutputStream` is a reference to a destination data sink (be it a file, network connection, etc), that we want to process as follows:

- We want to write the data as raw bytes.
- We want to write the data in a sequential order, one data appended after another data.

In order to support an abstract serialization of data as we did for the deserialization process, we have defined three subclasses of `OuputStream` class: `ItemOutputStream`, `SequenceOutputStream` and `TupleOutputStream`. The `ItemOutputStream` class uses the item encoding schema for its serialization. `SequenceInputStream` uses the sequence encoding schema. `TupleOutputStream` uses the tuple encoding schema.

[1] **Raw data** is a term for data collected from a source. Raw data have not been subjected to processing or any other manipulation, and are also referred to as primary data.

Chapter 4

Buffered Sequence

4.1 Management of large sequences

An XQuery program is constituted of a tree of expressions, which delivers a sequence when evaluated. The sequences can be constituted by zero or more items. To illustrate the data-flow of sequences across an expression tree, consider Figure 4-1(a). For the evaluation of the expression `expr1`, the arguments `expr2` and `expr3` must be evaluated. Expression `expr3` requires as an argument the delivered sequences from expression `expr4`. As showed in this example, in order to provide the result of the evaluation of a whole tree of expressions, there are many intermediate results that need to be managed by the XQuery processor.

There are two scenarios for these intermediate results. In the first one, the size of each intermediate result is smaller than the available main memory. For instance, the evaluation of the expression `1 + 1` is a single integer, which can be easily managed in main memory. In the second scenario, the size of each intermediate result is larger than the available main memory. Thus, we make use of our serialization framework to store those results in external memory.

Due to the diverse size of the intermediate results during the evaluation of expressions, we need an approach that is abstract from the whole evaluation of the tree of expressions. Figure 4-1 (b) illustrates where the approach need to be placed across the data-flow of sequences. In this scenario, when a parent expression calls the evaluation of a child expression, it receives an abstract Sequence object, which is iterated transparently.

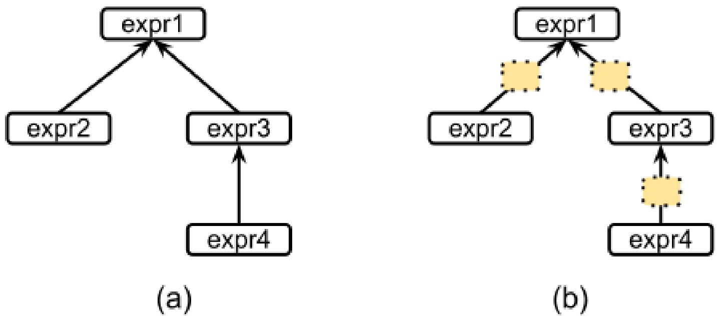


Figure 4-1: Tree of expressions

There are two approaches for the management of intermediate results in external memory. The first one is illustrated in Figure 4-2 (a). The expression `expr4` delivers items `item1`, `item2`, … , `itemn` when evaluated. These items are serialized immediately into an external disk one per one as they are delivered by the expression. When the expression `expr3` requires as argument the intermediate results from expression `expr4`, the intermediate result is obtained deserializing them from the external disk. This approach has the advantage of keeping a low memory footprint, but it has two severe problems. First, it generates too many small read operations, which are significantly slower than fewer large reads in a magnetic disk. Second, it underutilizes main memory by serializing even the smallest sequences, resulting in a drastic penalty for simple expressions that could be completely evaluated in main memory.

The second approach, which we propose, manages the intermediate results minimizing the number of I/O operations. This is achieved using a buffer, which is a temporary place in main memory which facilitates the exchange of data between main memory and disk. In this buffer, a set of items is kept in main memory up to a predefined cardinality threshold. Once it is reached, all items are written to disk in a single I/O operation, overcoming the first problem of the previous approach. The second problem is solved by using a predefining threshold in the buffer. If the threshold of the buffer is never reached, there is no serialization of the intermediate results. Therefore, there is no computation in main memory for encoding and decoding of sequences and there is no I/O operation. However, our approach has a disadvantage with respect to the advantage of the previous approach, it keeps larger memory footprint, but it can be adjusted by setting smaller thresholds in the buffer. Figure 4.2 (b) gives an overview of our approach.

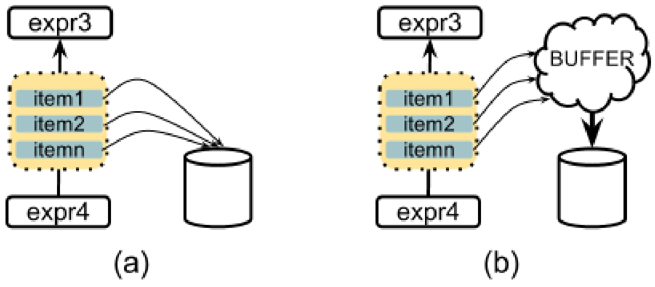


Figure 4-2: Management of intermediate result

Note that our approach is a generalization of the first one when its buffer size is set to zero. The buffer size can be defined as an array of bytes or as an array of items. In the former case, once it is created, it has a fixed size in main memory. Therefore, it is known how much of memory is allocated for the buffer. On the other hand, each item added to the buffer needs to be encoded to binary, which is an extra cost when the buffer does not need to be serialized. In contrast to this scenario, a buffer created as an array of items defines the number of items contained in main memory referenced by the buffer. However, there is no control of the memory overload. For instance, just one item can cause an overload the main memory. In this thesis, we have assumed that a single item always fit in main memory.

4.2 Implementation

The *BufferedSequence* component is the implementation of our approach. It is an extension of the *Sequence* Object. The *BufferedSequence* component manages the items delivered by any expression when evaluated. For each node in the expression tree, an instance of *BufferedSequence* is created. The instance of the BufferedSequence builds an initial empty buffer which is an array of items with a predefined threshold.

When the evaluated expression produces an item, it is delivered to the *BufferedSequence* instance. First, it is checked whether there is available space in the buffer for an item reference. If yes, the item is added into the buffer. Note that the items are added into the buffer in a sequential order as they are delivered by the expression. If the threshold of the buffer is never reached, the items referenced by the buffer are kept in main memory without need to use an external disk, avoiding IO operations. Figures 4-3 (a) illustrates this first phase of buffering of items.

When there is no available space in the buffer, meaning that the threshold has been reached, the *BufferedSequence* instance creates an empty *temporary file*. Then, the items referenced by the buffer are serialized into that temporary file using our serialization framework. The serialization of items referenced by the buffer consists of three steps. First, we iterate over the buffer and get a reference of an item. Second, we encode an item into a binary stream using the item encoding scheme proposed in our serialization framework. Third, we serialize the item binary stream into the temporary file. We repeat this process until there is no more item references in the buffer. Once all items from the buffer are serialized, we clean the buffer. Figure 4-3 (b) illustrates the second phase of buffering of items, which in some cases is never executed when the intermediate results are small.

If the *BufferedSequence* instance is fed with more items after the serialization process, the items are treated as in the first phase of the buffering. If the threshold of the buffer is reached again, the second phase of the buffering starts again, with the exception that the item binary streams are now appended to the existing temporary file created in the second phase of the buffering. This process represents the third phase of buffering, Figure 4-3 (c) illustrates it.

If the *BufferedSequence* instance is fed with more items after the serialization process and the threshold of the buffer is not reached, then the items referenced in the buffer are kept in main memory. Thus, our component avoids I/O operations and encoding computation cost for the last set of items referenced by the buffer.

In case, the buffered items must be serialized into the existing temporary file due to sorting purpose–we explain it in the next chapter–then, they are forced to be encoded as binary strings and they are appended to the existing temporary file. Figure 4-3 (d) illustrates the fourth phase of buffering.

Using our approach, we manage the data-flow of both small and large sequences across the tree of expressions. Moreover, the *BufferedSequence* component that implements this approach is abstract during the execution of the tree of expressions using the Brackit engine.

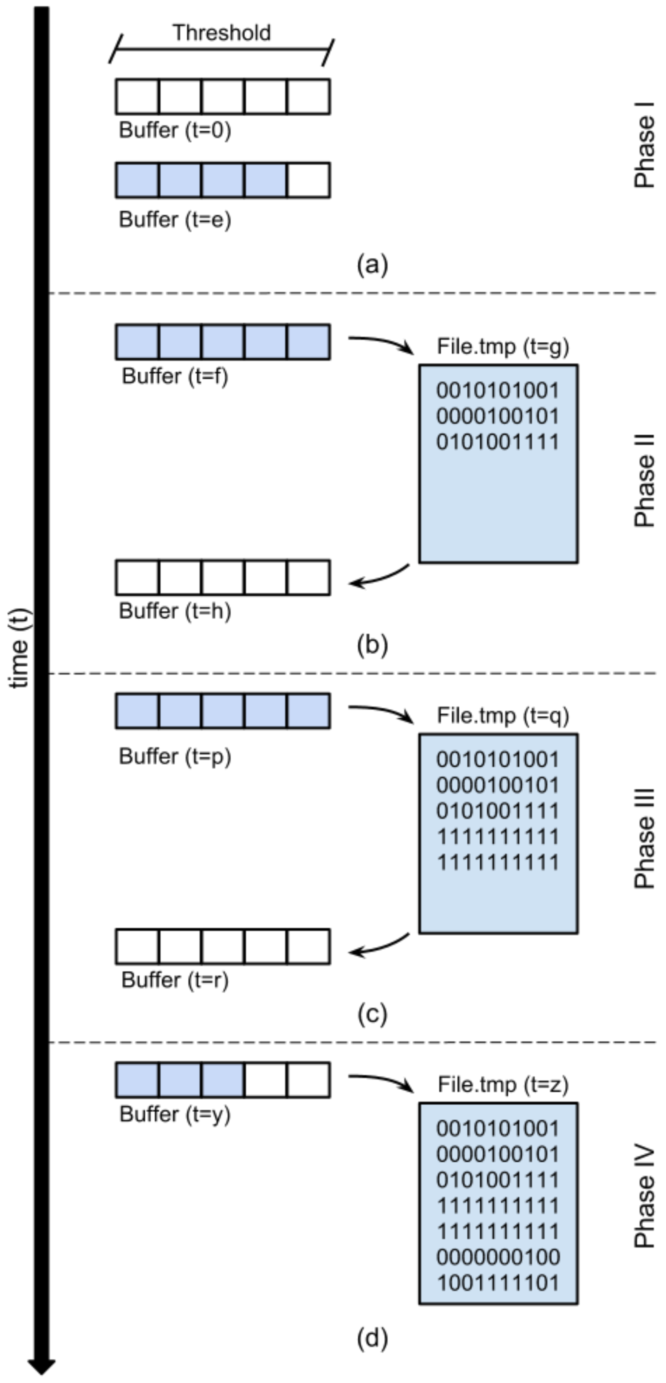


Figure 4-3: Management of intermediate results by BufferedSequence component

Chapter 5

External Sorting

5.1 Contextualization

In Brackit, a FLWOR expression is translated into a logical plan consisting of a top-down pipeline of operators, which follow the open-next-close model, where an output is requested from the final operator, which requests an input from the previous operator and so on.

In the case of the `Sort` operator, which is a blocking operator, when its output is requested with the *next* procedure, the `Sort` operator invokes the *next* procedure of its input operator repeatedly until all inputs are consumed. These inputs are sorted given the expression in the `order by` clause, and then delivered one by one as they are requested from the following operator. Figure 5-1 illustrates this process.

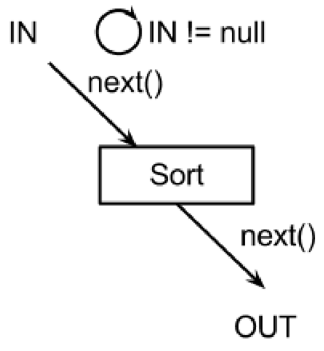


Figure 5-1: Execution of the `Sort` Operator

If the sort input is too large for an internal sort, then external sorting is needed. In this context, we need an approach that manages the sort input and the sort output following the open-next-close model as it is implemented in the operators. The sorting algorithm applied–whether internal or external–must be abstracted from the execution engine.

The approach proposed is similar to the previous approach in Chapter 4. In this scenario, it manages a buffer, which is a set of tuples, and a set of runs, which are temporary files that contain sorted tuples. If the whole input of the `Sort` operator is smaller than the threshold–which is a predefined maximum number of sequences that the buffer can contain–, the sort input is sorted internally in main memory. Otherwise, the sorted buffered tuples are serialized into runs using our serialization framework. Figure 5-2 illustrates our approach.

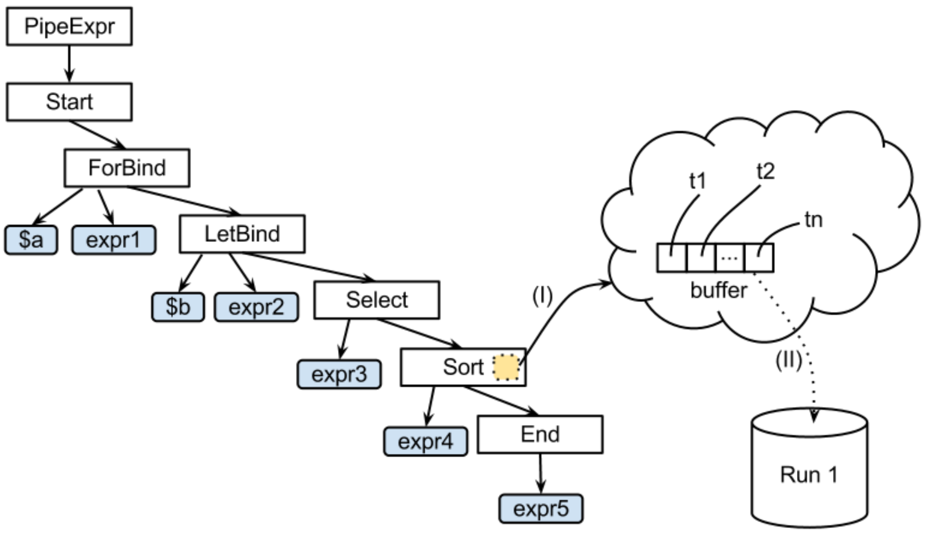


Figure 5-2: External sorting of stream of tuples.

The remainder of this chapter is organized as follows. In Section 5.2., we review the syntax of the `order by` clause. In Section 5.3, we present the implementation aspects of our approach. Finally, in Section 5.4, we introduce some optimization techniques for sorting.

5.2 Order by clause

FLWOR expressions have a natural ordering, which is determined by the bindings of the `for` variables (the order of the stream of tuples). However, the `order by` clause may be used to apply an explicit order. It takes a list of expressions, called *sort keys*.

The `order by` clause sorts the tuple stream according to each of the sort keys in order. The sort keys do not have to be returned in the result. Figure 5-3 shows an example of FLWOR expression using an `order by` clause, which we use as basis for the following discussions.

```
for $a in (1,2,3)

for $b in (3,2,1)

order by $b ascending, $a descending

return $a + $b
```

Figure 5-3: Sample FLWOR expression using the `order by` clause

During the execution of the FLWOR pipeline for evaluating the expression of the example in Figure 5-3, the Sort operator is fed with the following tuple stream:

```
[$a = 1, $b = 3]
[$a = 1, $b = 2]
[$a = 1, $b = 1]
[$a = 2, $b = 3]
[$a = 2, $b = 2]
[$a = 2, $b = 1]
[$a = 3, $b = 3]
[$a = 3, $b = 2]
[$a = 3, $b = 1]
```

Each sort key is first evaluated and atomized. A sort key must evaluate to the same type for every tuple in the tuple stream, and the type must be totally ordered. When the first sort key evaluated for two tuples results in a tie, then subsequent sort keys are used to break the tie. If all the keys tie, then the order of the tied tuples is implementation-defined unless the `stable` keyword is used in front of the `order by` clause. In that case, the tied tuples retain their original ordering relative to each other.

Each sort key expression may be followed by modifiers that control how that key affects the sort order. The most common modifiers are ascending and descending (ascending is the default) with the obvious effect. The other modifiers, `empty least` and `empty greatest`, control how the empty sequence sorts relative to all other values. In the scope of this work, only ascending and descending modifiers are considered.

The following tuple stream is the output of the `Sort` operator after reordering according to the given the given expression `$b ascending, $a descending` of the example in Figure 5-3:

```
[$a = 3, $b = 1]
[$a = 2, $b = 1]
[$a = 1, $b = 1]
[$a = 3, $b = 2]
[$a = 2, $b = 2]
[$a = 1, $b = 2]
[$a = 3, $b = 3]
[$a = 2, $b = 3]
[$a = 1, $b = 3]
```

The result of the example in Figure 5-3 after the reordering of tuples is the following sequence:

(4, 3, 2, 5, 4, 3, 6, 5, 4)

5.3 Implementation

Our approach is implemented by the *TupleSort* component. It is a combination of main-memory and external-merge sort. The implementation is I/O robust w.r.t. pre-sorted or small inputs.

The *TupleSort* component manages a buffer, which is an array of tuples, and runs, which are files. The size of the buffer is predefined by the variable *maxSize*, which is the maximum number of sequences that the buffer can contain. Once the *maxSize* is reached, the serialization of the buffered tuples occurs.

Given the stream of tuples (*st1*, *st2*, *st3*) from Figure 5-4, and defining *maxSize* = 3, the stream of tuples starts being serialized into runs at different positions of the space of the stream. The stream *st1* starts to serialize its buffered tuples at position 4, the stream *st2* at position 3, and stream *st3* at position 2. Note that the stream *st1* starts serializing its buffered tuples after its buffer has reached its maximum size. This scenario occurs just when a tuple is a single item. Otherwise, the buffered tuples are serialized when the buffer is not full.

```
1     [$a = 1]   [$a = 1, $b = 2]       [$a = 1, $b= (1,2)]
2     [$a = 2]   [$a = 2, $b = 3]       [$a = 2, $b= (2,3)]
3     [$a = 3]   [$a = 3, $b = 4]       [$a = 3, $b= (3,4)]
4     [$a = 4]   [$a = 4, $b = 5]       [$a = 4, $b= (4,5)]

        (st1)           (st2)                   (st3)
```

Figure 5-4: Stream of tuples

It is important to note that *maxSize* defined in terms of sequences–instead of tuples or items–manages the overloading of memory better than the two other alternatives. For instance, if the *maxSize* were defined as maximum number of tuples that a buffer can contain, the memory can be overloaded due to a single tuple being too large composed of many sequences, and a sequence may be composed of many items. On the other hand, if *maxSize* were defined as maximum number of items that the buffer can contain, the overloading of the memory is precisely controlled by this threshold. However, the serialization of buffered tuples could start when a sequence in a tuple is being evaluated. Thus, we would require to keep in main memory the current tuple being evaluated in order to not loose that information. For instance, if the *maxSize* (maximum number of items) is 5, the stream *st3* of the example in Figure 5-4 would be serialized when the item `3` in sequence `$b= (2,3)` is being evaluated in the tuple `[$a = 2, $b= (2,3)]`.

The threshold *maxSize* defined in terms of sequences manages both memory overloading and data movement. Using our previous approach *BufferedSequence* which manages large sequences, a large sequence is an instance of *BufferedSequence* which references the file where the sequence is serialized. When a tuple contains sequences of instance *BufferedSequence* and the tuple needs to be serialized, the BufferedSequence is serialized as file path name it references. In this approach, first we reduces the size of the runs where the tuples are serialized just using the file path name which its size is smaller than serializing the whole sequence. Second, the data movement for external sorting is smaller due to the encoding schema for tuples proposed in Chapter 3. Note that *sort keys* in the `order by` clause must be of type atomic. Thus, a *BufferedSequence* sequence is never compared to another sequence in other tuple in the *Sort* operator.

Figure 5-5 illustrates a stream of tuples as input for the *Sort* operator. Here *BS12,..., B32* refer to *BufferedSequence* instances. If *maxSize* is 6, then the serialization of the buffered tuples starts when the second input of the *Sort* operator is requested.

```
1     [$a = 1, $b= BS11, $c= BS12, $d= 2]
2     [$a = 2, $b= BS21, $c= BS22, $d= 3]
3     [$a = 4, $b= BS31, $c= BS32, $d= 4]
```

Figure 5-5: Tuple stream composed by one BufferedSequence

Algorithm 5-1 describes how each input of the *Sort* operator is managed by the *TupleSort* component. The variable *size*, which is the accumulated number of sequences that the buffer contains, and the variable *count*, which serves as a pointer in the buffer where the tuples is added, are initialized with zero. The *maxSize* variable is the predefined threshold for serializing the buffered tuples. The *getSize* function returns the number of sequences that the tuple contains.

Algorithm 5-1 *add* method

```
1: function ADD (Tuple item)
2:     itemSize ← getSize(item);

3:     if ((maxSize > 0) && (size + itemSize > maxSize)) then
4:         writeRun();
5:     end if

6:     buffer[count++] ← item;
7:     size += itemSize;

8: end function
```

5.3.1 Internal sorting

In the scenario that the input sort is smaller than *maxSize*, just the internal sorting is needed. The internal sorting takes place after all tuples from input *Sort* operator are consumed. Because the *maxSize* has not been reached, there is no generation of runs. Thus, there is no I/O operation.

The internal sorting is implemented using the method `java.util.Arrays.Sort(buffer,0,count,comparator)` provided by the implementation platform Java. This method sorts the specified range of the specified array of tuples–defined as a buffer–according to the order induced by the specified `comparator` defined in the `order by` clause. The range to be sorted extends from index `0`, inclusive, to index `count`, exclusive (if `count` is zero, the range to be sorted is empty). All tuples in the range must be mutually comparable by the specified comparator. This sort is guaranteed to be stable, where equal tuples will not be reordered as a result of the sort.

The sorting algorithm used in `Array.sort()` is a modified mergesort in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist. This algorithm offers guaranteed O(nlog(n)) performance.

Figure 5-6 illustrates the three phases for internal sorting. First, the sort input is added into the buffer using the algorithm 5-1. Once the whole sort input is added to the buffer defined in the *TupleSort* component, then the sorting phase takes place using the `Array.sort()` method. The sorted input–tuples in the buffer–is transformed as a stream of Tuples. Finally, when the output of the *Sort* operator is requested the stream is iterated delivering a tuple per request.

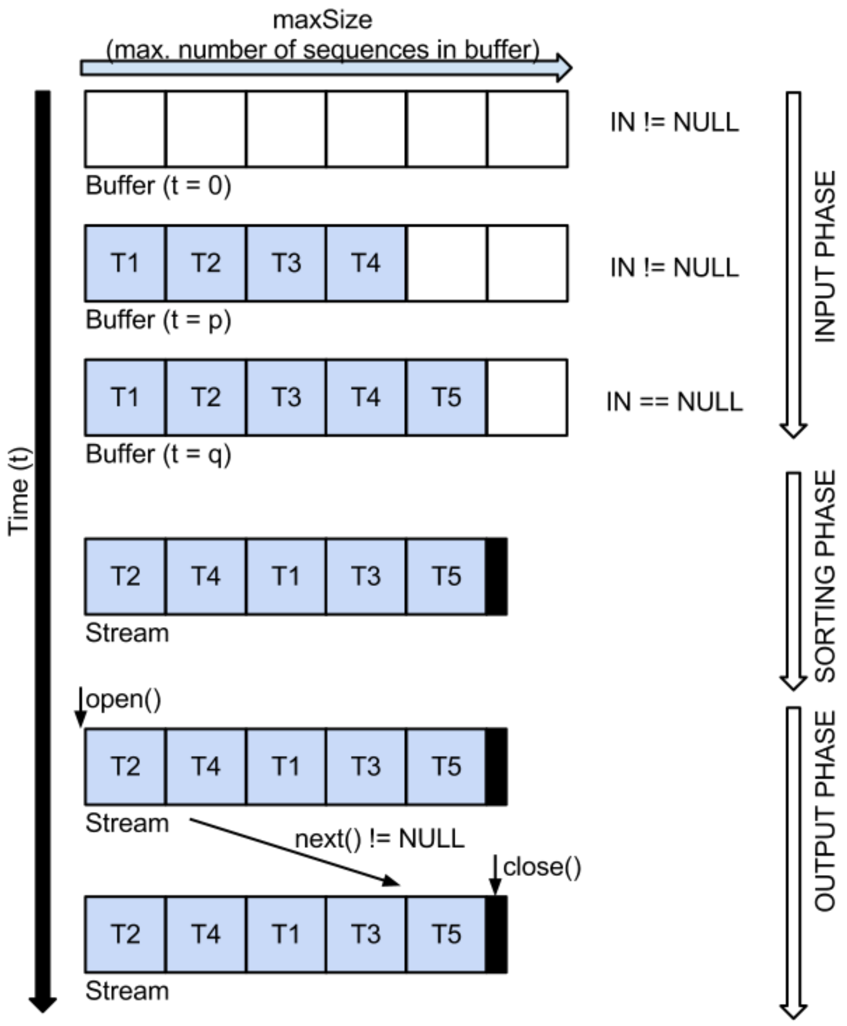


Figure 5-6: Sorting operator with internal sorting

5.3.2 External sorting

The external sorting uses the well-known merge-sort algorithm. The merge-sort algorithm–in our approach for the management of large input sort–is implemented using a buffer and runs which were previously introduced in Section 5.3.1. The sorting procedure in external sorting is split in two phases: generation of runs and then merging of runs. In the first one, a run is generated when the number of sequences that the buffer contains is greater than *maxSize* threshold defined in the *TupleSort* component, see Algorithm 5-1. The produced runs are then gradually merged in a single file during the second phase.

Phase I: Generation of runs

Whenever the *maxSize* threshold is reached, the buffered tuples referenced by the buffer are sorted using the internal sorting algorithm introduced in Section 5.3.1. Then a temporary file is created. The sorted buffer is then serialized in the temporary file, one tuple at a time, using our serialization framework. Once all tuples are serialized into the run, its location is registered in the *TupleSort* component. Once the run is written, the *size* and *count* global variables of the component are set to zero.

To exemplify the previous context, consider the example of Figure 5-3. If we assume that *maxSize* is 4, then the number of tuples that can be hold in main memory is 2, because each input tuple of the `Sort` operator contains two sequences. If the first two tuples of the input sort are sorted given the *sort keys* (`$b ascending, $a descending`) and then serialized in the run; the following run is generated:

```
Run0:
      [$a = 1, $b = 2]
      [$a = 1, $b = 3]
```

If the *maxSize* is over reached again, then a new run is created or it is appended the sorted buffered tuples to the previous run. The first scenario occurs when the first tuple of the current sorted buffer is ordered before than the last tuple in the run. Using the previous example, the last tuple in a run is `[$a = 1, $b = 3]` and first tuple of the current sorted buffer is `[$a = 1, $b = 1]`. As the first tuple of the sorted buffer is ordered before than the last tuple in a run, the new run generated as follows:

```
Run1:
      [$a = 1, $b = 1]
      [$a = 2, $b = 3]
```

Appending the current sorted buffer to the previous run will occur if the last tuple in run is ordered before than the first tuple in the sorted buffer. Using the previous example, this situation never occurs, therefore `Run2` and `Run3` are generated.

```
Run2:
      [$a = 2, $b = 1]
      [$a = 2, $b = 2]

Run3:
      [$a = 3, $b = 2]
      [$a = 3, $b = 3]
```

In case the *maxSize* is never reached again, but the buffer contains tuples in it, the tuples are kept in main memory. From the example in Figure 5-3, after all inputs are consumed the buffer contains just one tuple in main memory:

```
Sorted Buffer:
     [$a = 3, $b = 1]
```

The Algorithm 5-2 describes how the different scenarios for writing a run are managed in our approach. Fisrt, the *sortBuffer*() function orders the tuples in the buffer. Then, it checks whether the sorted buffer can be appended to a previous runs. Note that appendding a buffer into a run renders two situations in the sorting phase. First, we can have an unbalanced tree of runs, meaning that a run can contain different number of tuples per run. For instance a run `run1` can contains 10 times the number of tuples in run `run2`. Second, the number of run-merges is smaller due to the smaller number of generated runs when the append run condition is satisfied. In case this condition is not satisfied, the current run is close. Then, it registers the location of a file where the buffered tuples are serialized in it. Once the run is written, *size* and *count* variables is set to zero for the next buffering of the sort input.

Algorithm 5-2 *writeRun* method

```
1: function WRITERUN ()
2:     sortBuffer()

3:     if ((lastInRun != null) && (lastInRun<=buffer[0])) then
4:         appendToRun()
5:         return
6:     end if

7:     if (currentRun !=null) then
8:         currentRun.close()
9:     end if

10:    run ← createTempFile()

11:    currentRun ←new BufferedOutputStream(new FileOutputStream(run));

12:    for (int i = 0; i < count; i++) do
13:        lastInRun = buffer[i];
14:        writeItem(currentRun, lastInRun);
15:    end for

16:    runs[runCount++] ← run;
```

```
17:     size ← 0;
18:     count ← 0;
19:     initialRuns++;

20: end function
```

Phase II: Merging

In this phase, the sorted runs created during the first phase are merged into larger runs of sorted tuples. The merge continues until all tuples of the sort input are in one large run. The output of the merge phase is the output of the *Sort* operator.

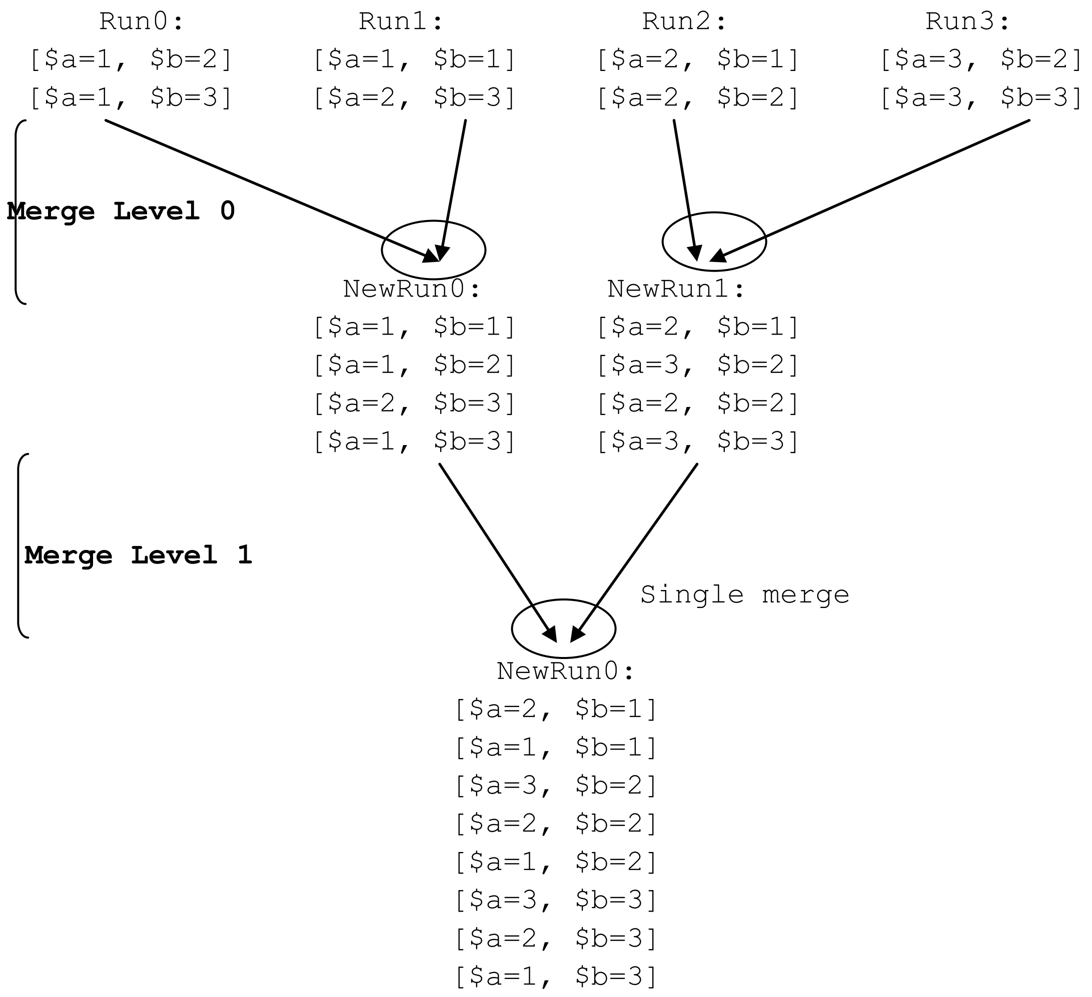
The merging phase consists of zero or more merge level(s) until one run is delivered. Each merge level consumes runs and produces new runs. The number of new runs is smaller than the consumed runs at each merge level. The size of each new run is equal to the size of the consumed runs. Algorithm 5-3 describes a strategy to merge the runs in pairs. The number of merges at each level is calculated dividing the number of runs by 2. If the numbers of merges is not even, then the number of new runs is the same as the number of merges. The merge pairs can be sorted forward or backward. The first one is described in the algorithm. In case the number of runs is even, the number of new runs is equal to the number of merges plus one, where the single run is used as output for the next merge level.

Algorithm 5-3 *mergeRuns* method

```
1: function MERGERUNS (File[] runs)
2:      newRuns ← null;
3:      mergeLevel ← 0;
4:      runCount ← runs.length();

5:      while (runCount >1) then
6:              merges ← runCount/2
7:              singleRun ← runCount mod 2
8:              newRunCount ← merges + singleRun
9:              newRuns← new File[newRunCount]

10:             pos ← 0;

11:             if (singleRun) then
12:                     for (int i = newRunCount - 1; i > 0; i--) do
13:                             newRuns[pos++] ← merge(runs[2 * i - 1], runs[2 * i])
14:                     end for
```

```
15:                 newRuns[newRunCount - 1] ← runs[0]
16:         else
17:                 for (int i = newRunCount - 1; i >= 0; i--) do
18:                         newRuns[pos++] ← merge(runs[2 * i], runs[2 * i + 1]
19:                 end for
20:         end if

21:         runs ← newRuns
22:         mergeLevel ← mergeLevel +1

23:     end while
24: end function
```

Using the example of Figure 5.3, the runs generated were `Run0`, `Run1`, `Run2` and `Run3`. If we merge them using the algorithm introduced previously, we obtain the following merge level:



If there is no buffered tuple in main memory, the output of the sort operator is iterated from the final run merge. It delivers one tuple at a time when the *next()* procedure is requested of the following operator. If it is not the case, the sorted buffer in main memory and the final run merge are iterated delivering each one a tuple which are compared providing as an output the one ordered before. The other tuple is kept in memory for comparison with the next tuple of final merge run or the next tuple of the buffer. The tuples are compared in pair from different sources.

From the previous example, there is a buffer in main memory containing the tuple `[$a = 3, $b = 1]` which is merged with the final run producing as a output the following stream of tuples as they are requested:

```
 SortedOutput:
[$a=3, $b=1]
[$a=2, $b=1]
[$a=1, $b=1]
[$a=3, $b=2]
[$a=2, $b=2]
[$a=1, $b=2]
[$a=3, $b=3]
[$a=2, $b=3]
[$a=1, $b=3]
```

During internal and external sorting, there is a significant cost of in-memory computation. It is dominated by two operations: key comparison and data movement. In order to provide good performance for sorting operations, we have implemented some techniques to speed up key comparison. In the next section, we introduce the techniques implemented in the Brackit engine.

5.4 Optimization sorting techniques

The cost of in-memory comparison is an issue that can be quite complex due to multiple keys to be compared within each tuple, each with its own type, length, sort direction (ascending or descending), and so on. Given that each tuple participates in many comparisons, it seems worthwhile to reformat each tuple before and after sorting if the alternative format speeds-up the multiple operations in between.

The format that is most advantageous for fast comparisons is a simple binary string. Thus the entire complexity can be reduced to comparing binary strings, and the sorted output tuples can be recovered from binary string. Since comparing two binary strings takes only tens of instructions, it makes sense to use normalized keys for sorting. Needless to say, hardware support is much easier to exploit if key comparisons are reduced to comparisons of binary strings [18].

In the Brackit engine, we have implemented some techniques to speed up the comparison of tuples as follows. Fist, the tuple is transformed by adding the *key sorts* items defined in the `order by` operator at the beginning of the tuple. Second, we encode the transformed tuple using our tuple encoding schema defined in Section 3.4. Then, they are serialized into runs. When the merging phase takes place between runs, the tuples are compared as binary string. Then, when the merge phase has finished and the tuples need to be delivered, the transformed tuple is restored to the original format. Figure 5-5 illustrates this process.

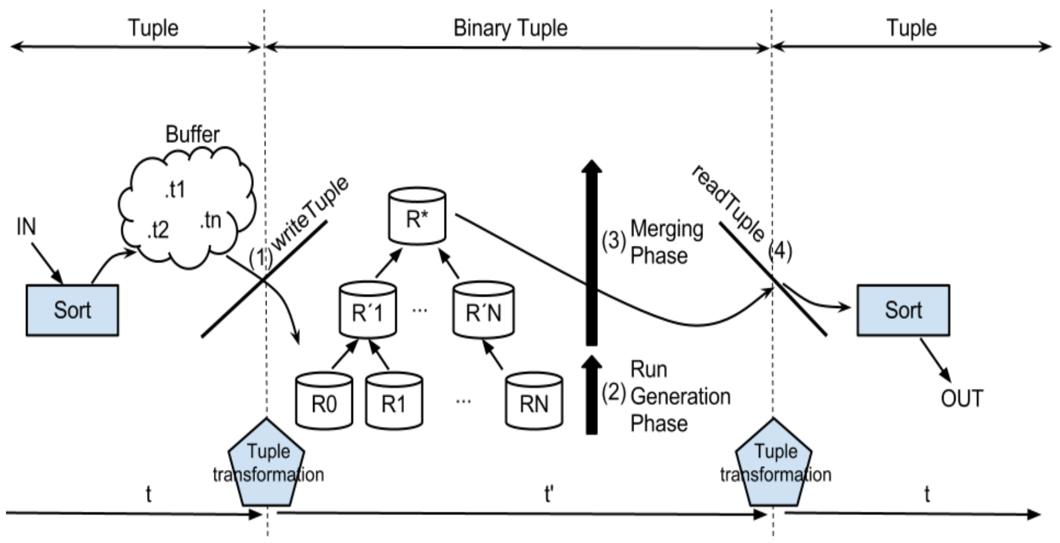


Figure 5-5: External Sorting with optimization

The following run exemplifies how the tuples are written into a run with and without transformation:

```
                         Run0:

       Tuple          Tuple with transformation
  [$a=1, $b=2]     [$b=2, $a=1, $a=1, $b=2]
  [$a=1, $b=3]     [$b=3, $a=1, $a=1, $b=3]
```

Transformed tuples may be substantially larger than the original ones, thus increasing the requirements for both memory and disk, space and bandwidth. However, the proposed optimization technique is beneficial, as most comparison will be decided by the first bytes alone. In the next Chapter, we assess the efficiency of our implementation.

Chapter 6

Experiments

6.1 XSF

In this section we evaluate the encoding framework in terms of the size of the encoded sequence w.r.t. size of the original sequences and time to encode and decode sequences.

For this experiment, we have generated sequences of different sizes (1 KB, 1MB, 10MB) containing atomic items of different types (Integer, Float, Double, String). The values of the atomic items are randomly generated.

6.1.1 Sequence size vs. Encoded sequence size

Figures 6-1 illustrates the relative size of the encoded sequence with respect to the original size of the sequence. The encoded sequence sizes are up to 3.5 times larger than the original sequence. For Integer atomic items, the encoded sequence size is in average 3.5 times the original one. For Float atomic items, its encoded size is 3.2 times the original one. For Double atomic items, its encoded size is in average 2.8 times the original one. The size of String atomic items are encoded closely as its original size, it is in average 1.1 times.

The encoding schema used for our XQuery Serialization Framework is not compact w.r.t. the original XDM instances. This is because the encoded XDM instance contains metadata like type and length fields which are used for deserialization purpose. The proposed encoding schema can be extended with compression techniques to reduce the size of the encoded data.

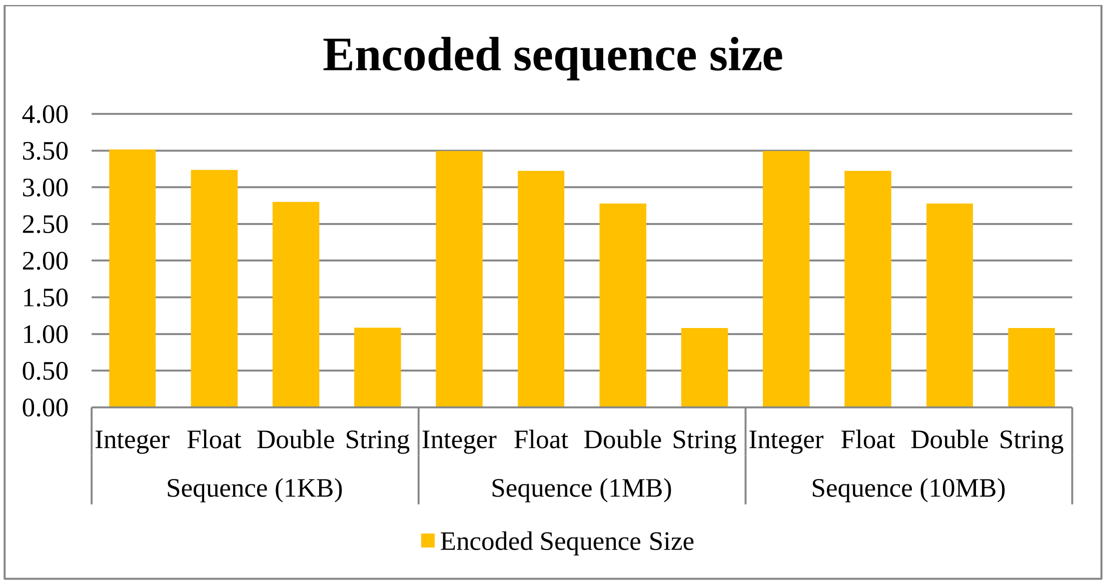


Figure 6-1: Relative encoded sequence size

6.1.2 Encoding time vs. Decoding Time

The time for encoding and decoding a sequence of 1KB, 1MB or 10 MB differs with respect to its size and the atomic item type they contain (see Figure 6-2, Figure 6-3 and Figure 6-4).

When the sequence size is 1KB (see Figure 6-2), we obtain the following measures. If the type is Integer, it takes 12 ms for encoding the sequence. If the type is Float, it takes 36 ms for encoding the sequence. If the type is Double, it takes 20 ms for encoding the sequence. If the type is String, it takes 1 ms for encoding the sequence. The time for decoding sequences is less than the maximum time for encoding the different sequences.

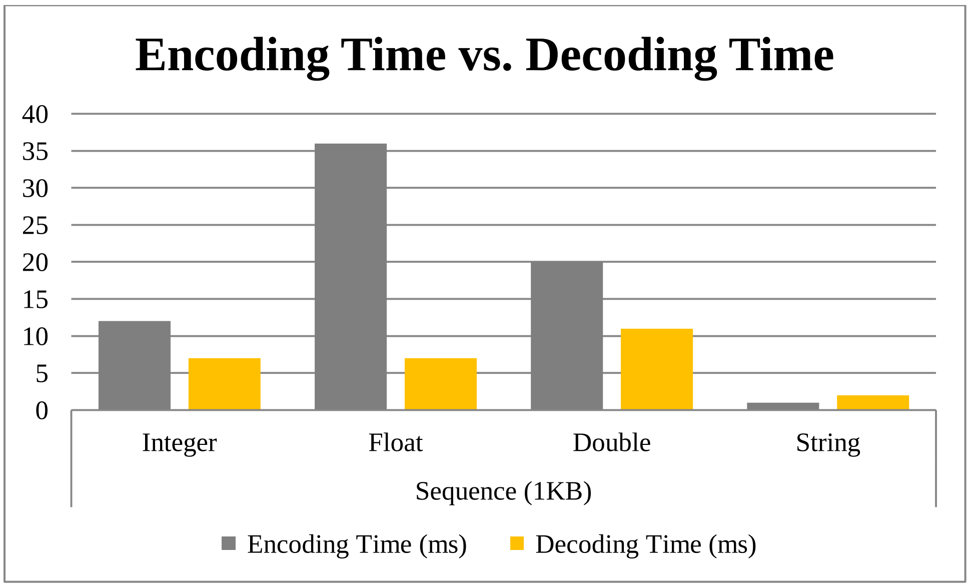


Figure 6-2: Encoding Time vs. Decoding Time (Sequence 1KB)

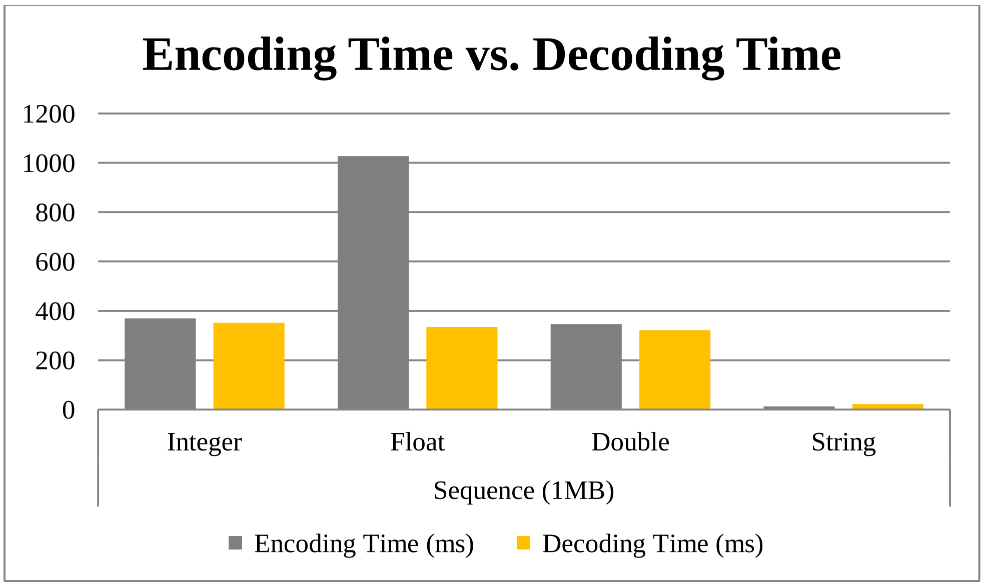


Figure 6-3: Encoding Time vs. Decoding Time (Sequence 1MB)

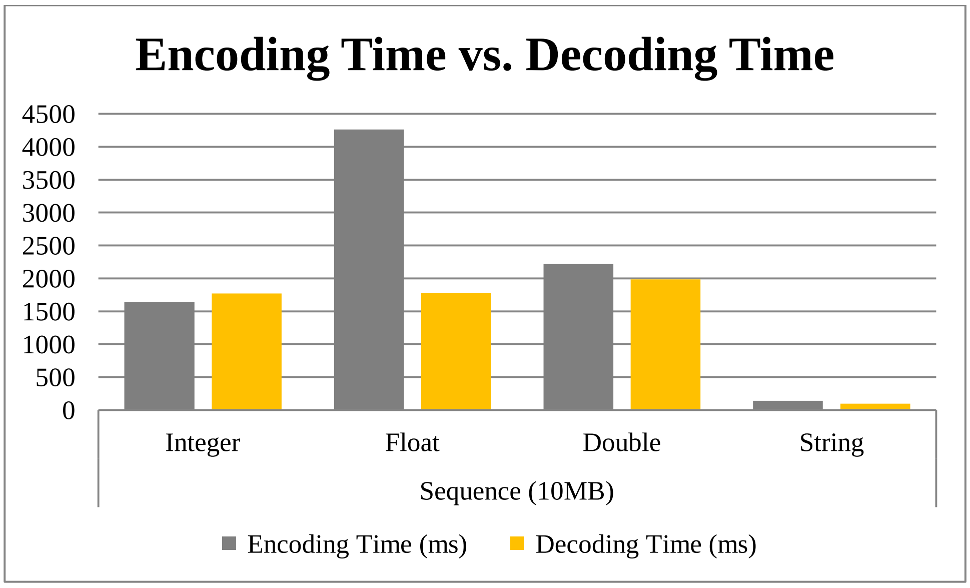


Figure 6-4: Encoding Time vs. Decoding Time (Sequence 10MB)

As you can see in Figure 6-2, Figure 6-3 and Figure 6.4, the encoding and decoding times of a sequence composed by String atomic items are smaller than other types of atomic items. This is because each character of the string is interpreted as binary. Thus there is no need for encoding or decoding it. For a sequence of 10MB size composed of string atomic items, it takes 138 ms and 97 ms for encoding and decoding respectively.

The measures obtained in this section–encoding size, and execution time for encoding and decoding of XDM instances–need to be considered when we run the experiments for *BufferedSequence* and External Sorting.

6.2 Buffered Sequence

In this section we measure the execution time for evaluating a FLWOR expression which processes large sequences. First, we measure their execution time without using our *BufferedSequence* approach. Second, we measure it using our approach. In this second scenario, we measure it using different buffer sizes.

6.2.1 Execution Time

Figure 6-5 shows the query used for this experiment. It is a FLWOR expression with another nested FLWOR expression. The nested expression generates large intermediate results, namely one million integers. These intermediate results are used twice by other expressions in the query for generation aggregations.

```
for $x in (1 to 10)
let $y:= ( for $a in ($1 to 1000000)
           return $a)
let $f:= (sum($y)mod $x)
group by $f
return avg($y)
```

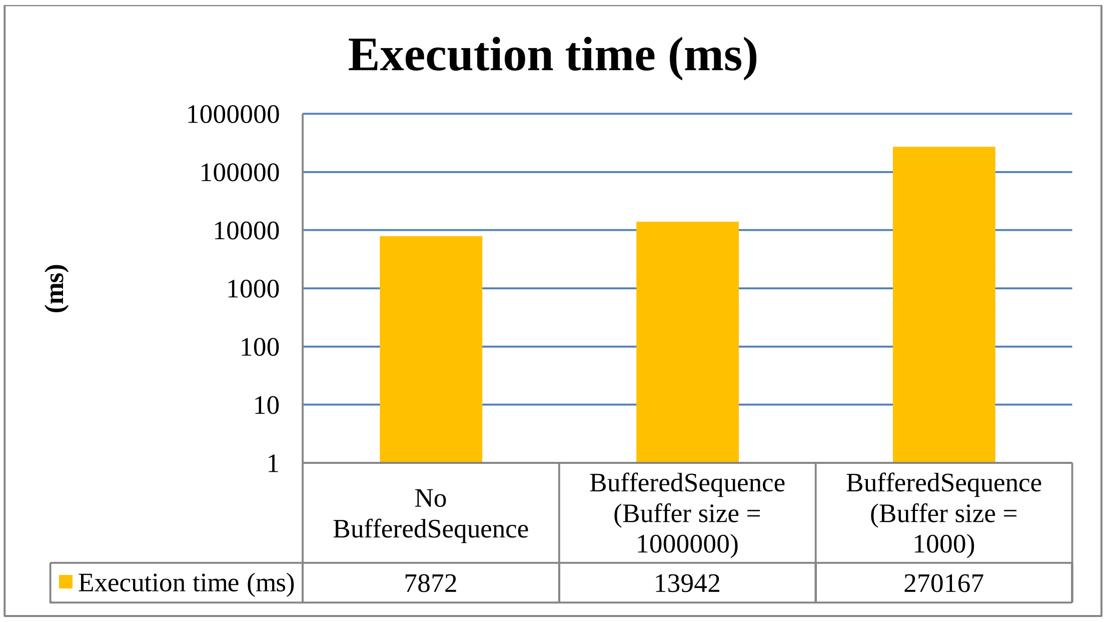
Figure 6-5: Nested FLOWR expression

When we integrate our approach *BufferedSequence* in the Brackit engine, the evaluation of the query transforms the nested FLWOR expression into an instance of *BufferedSequence*.

Figure 6-6 shows the execution time of the query under different scenarios. If no *BufferedSequence* component is integrated to the Brackit engine, then the execution of the query takes 7872 ms. If we integrate our component to the *Brackit* engine, then the execution time of the query is larger than without using it.

If we use our *BufferedSequence* with a buffer size of 1000000–meaning that all items in the sequence are added to the buffer and they are never written to disk–then the execution of the query takes 13942 ms. This execution time is almost the double of the execution time without using the *BufferedSequence*. It is expected to have the double time because we iterate every sequence twice when creating the *BufferedSequence*. The first iteration takes place when we add the items of the sequence to the buffer. The second iteration takes places when iterate over the buffer for evaluating the sequence.

If we use our *BufferedSequence* with buffer size of 1000–meaning that most of its items will be serialized–then the execution of the query takes 270167ms. This execution time is 34 times the execution time without using the *BufferedSequence* and it is 19 times the execution time using the BufferedSequence with buffer size of 1000000. It is expected to have larger time because the *BufferedSequence* generates many I/O operations.



	No BufferedSequence	BufferedSequence (Buffer size = 1000000)	BufferedSequence (Buffer size = 1000)
🟨 Execution time (ms)	7872	13942	270167

Figure 6-6: Execution time of nested FLWOR expression

6.3 External Sorting

In this section we measure the execution time for evaluating a FLWOR expression which processes large sorting operations using our external sorting algorithm implemented for the *Sort* operator. First, we measure its execution time without using binary comparison. Second, we measure it using binary comparison. In both scenarios, we increase the size of the buffer to determine how the performance of the query varies.

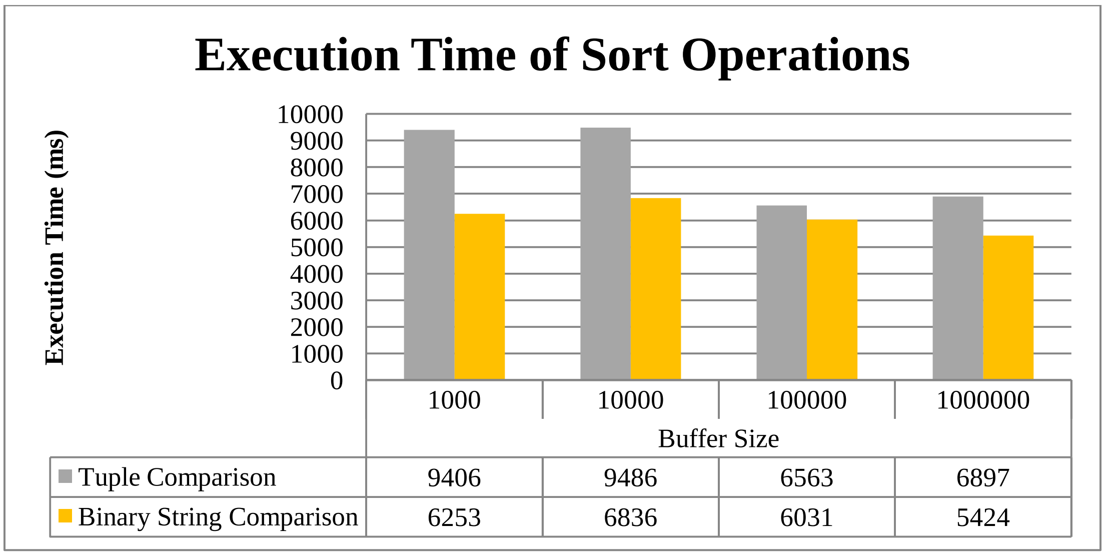
6.3.1 Execution Time

Figure 6-7 shows the query used for this experiment. It is a FLWOR expression which requires the sorting of large stream of tuples, namely one hundred million tuples generated by the two `for` clauses in the expression. The stream of tuples are sorted according the *sort key* `$b`.

As you can see in Figure 6-8, the execution time of the query using tuple object comparison takes more than time than when the tuples are compared by their binary representation using the XSF for all different buffer sizes (1000, 10000, 100000, 1000000). Comparing tuples during external sorting by its binary representation spare up to 33% of execution time. It is expected to have such a better performance due to the binary representation does not require to be transformed as tuple in the implementation language, therefore we spare this time of encoding and decoding the tuple.

```
for $x in (1 to 1000000)

for $b in (100 to 1)

order by $b

return ($a, $b)
```

Figure 6-7: FLWOR expression



	1000	10000	100000	1000000
■ Tuple Comparison	9406	9486	6563	6897
■ Binary String Comparison	6253	6836	6031	5424

Figure 6-8: Execution time of Sort Operations

Chapter 7

Conclusion

We presented an XQuery serialization framework for enabling the management of large intermediate results during query evaluation. We used our framework in two different applications. The first one, the *BufferedSequence* component, manages large sequences. The second one, the *Sort* operator, supports external sorting for managing large tuple streams.

The XQuery Serialization Framework proposed in this work defines an encoding schema for items, sequences and tuples. The encoding schema of each data item includes its metadata like type information and value length, which is obtained during the execution of the query. Thus, as it is showed in the experiment in Section 6.1, the size of the encoded data is larger than its original size because the metadata is added for each item.

We could improve the compactness of our encoding schema by previously analyzing the expression tree to determine type information, then adding this information as schema of the file where the data is stored. This analysis, known as static typing, could avoid us encoding metadata for each item. Thus, we could reduce the size of the encoded data and reduce the execution time for encoding and decoding.

Our *BufferedSequence* component enables the management of large sequences, but the performance of a query evaluation using it decreases due to an eager loading of items in the buffer when the sequence is constructed. Furthermore, the *BufferedSequence* component is statically integrated into the `PipeExpr` of the Brackit engine. This leads two problems: First, we use *BufferedSequence* for only `PipeExpr` expressions. Thus, other expressions which may as well result in large sequences are not supported. Second, we use *BufferedSequence* for every instance of

`PipeExpr`. In most expressions, the contents of a sequence are simply streamed, which means that there is no need for buffering even very large sequences. This situation could be improved by previously analyzing the expression tree to decide whether to use a *BufferedSequence* or not.

The external sorting algorithm for the *Sort* operator enables the sorting of large tuple streams using external sorting. Moreover, the formatting of the tuples as binary string for comparisons paid off showing better performance in the experiments in Section 6.3 than without using it. However, its performance could be improved if the encoding and decoding of the tuples were smaller than the used one, because a binary tuple is encoded by duplicating the *sort keys* of the `order by` clause in the beginning. Instead of that, we could just move and not append the *sort keys* of a tuple in the beginning without increasing the size of the tuple.

Besides the merge-sort strategy implemented for the external sorting, other merging strategies could be implemented, and later on we could determine which strategy performs better or worse than the current one. As we have implemented an external algorithm for the Sort operator, we could implement external algorithms for other blocking operators like join and group by.

In this thesis, the XQuery Serialization Framework, the *BufferedSequence* component and the *Sort* operator are integrated into the execution engine of Brackit. However, the performance of our approaches in terms of space consumption and execution time could be improved if the compilation phase would be considered for the analysis of metadata of XDM instances and estimation of the numbers delivered results.

The use of our framework and the two proposed components enables the execution of arbitrary XQuery over very large datasets, whose size is limited only by the available external memory.

Bibliography

[1] Zhen Hua Liu and Ravi Murthy. A Decade of XML Data Management: An Industrial Experience Report from Oracle. IEEE International Conference on Data Engineering, 1351-1362, 2009.

[2] W3C. XQuery 3.0: An XML Query Language.
http://www.w3.org/TR/xquery-30/, 2011.

[3] W3C. XQuery and XPath Data Model 3.0
http://www.w3.org/TR/xpath-datamodel-30/, 2011.

[4] W3C. XML Schema Definition Language (SDL) 1.1 Part 1: Structures
http://www.w3.org/TR/xmlschema-11-1/, 2011.
XML Query Uses Case

[5] W3C. XML Query Uses Case
http://www.w3.org/TR/xquery-use-cases/, 2007

[6] Sebastian Bächle and Caetano Sauer. Unleashing XQuery for Data-independent Programming. Submitted.

[7] Roger Bamford, Vinayak R. orkar, Mathias Brantner, Peter M. Fischer, Daniela Florescu, David A. Graf, Donald Kossmann, Tim Kraska, Dan Muresan, Soring Nasoi, and Markos Zacharioudaki. XQuery Reloaded. PVLDB, 2 (2): 1342-1353, 2009.

[8] Hennessy, Jhon L.
Computer Architecture: a quantitative approach/ John L. Hennessy, David A. Patterson; with contributions by Andrea C. Arpaci-Dusseau… [et al.]. – 4th ed.

[9] E. I. Cohen, G. M. King, and J. T. Brady. Storage Hierarchies. IBM Systems Journal, Vol. 28. No 1: 62-76, 1989.

[10] Gray, Jim, 1944-

Transaction processing: concepts and techniques/ Jim Gray and Andreas Reuter.

[11] W3C. XQuery Update Facility
http://www.w3.org/TR/xquery-update-10/, 2011.

[12] W3C. XQuery and XPath Full Text
http://www.w3.org/TR/xpath-full-text-10/, 2011.

[13] W3C. XQuery Scripting Extension
http://www.w3.org/TR/xquery-sx-10/, 2010

[14] Caetano Sauer. XQuery Processing in the MapReduce Framework.
Master thesis, University of Kaiserslautern, Kaiserslautern, 2012.

[15] Introduction to Java Input Streams
http://www.javamex.com/tutorials/io/input_stream.shtml

[16] XQuery 3.0 Requirements
http://www.w3.org/TR/xquery-30-requirements/

[17] XQuery Query Use Cases
http://www.w3.org/TR/xquery-use-cases/

[18] Goetz Graefe. Query Evaluation Techniques for Large Databases. ACM Comput. Surv., 25(2): 73-170, 1993

[19] Goetz Graefe. Implementing Sorting in Database Systems. ACM Comput. Surv., 38(3): 1-37, 2006