

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHELOR OF COMPUTER SCIENCE

AUGUSTO BARCELLOS BERND

**Unnesting Queries Using Algebraic Equivalences
in an XQuery Engine**

Graduation Thesis

M. Sc. Caetano Sauer
Advisor

Prof. Dr. Renata Galante
Coadvisor

Porto Alegre, July 2015

CIP – CATALOGIN-IN-PUBLICATION

Bernd, Augusto Barcellos

Unnesting Queries Using Algebraic Equivalences
in an XQuery Engine / Augusto Barcellos Bernd. – Porto
Alegre: Graduação em Ciência da Computação da UFRGS,
2015

52 f.: il.

Graduation Thesis – Universidade Federal Do Rio Grande do
Sul. BACHELOR OF COMPUTER SCIENCE, Porto Alegre,
BR–RS, 2015. Advisor: Caetano Sauer; Coadvisor: Renata
Galante

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling Franco

Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb

Coordenador do Curso de Ciência da Computação: Prof. Raul Fernando Weber

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ACKNOWLEDGEMENTS

I would like to thank my family, because they are the base of my life, giving me all the love, patience, physical and emotional support to complete my graduation, including the time I was living in Germany, I always felt I could stand by them, even far away.

I would like to thank Prof. Theo Härder for providing me the opportunity to study at Technische Universität Kaiserslautern and join his research team at DBIS, where beyond the chance to develop this project, I could have an amazing academic and life experience. I also have to thank Caetano Sauer for also being key part of this opportunity and giving me every kind of support I needed while I was there, with much patience.

I would also like to thank Universidade Federal do Rio Grande do Sul (UFRGS) and Instituto de Informática for affording a great academic environment, and all the professors. In special, I thank Prof. Taisy Weber, for coordinating the exchange program to Kaiserslautern, and Prof. Renata Galante for helping me to develop my Thesis with important advices.

And last, but not least, I would like all my friends from graduation along these 5 years, including my friends from Kaiserslautern, for helping me on my studies and providing me great moments in this challenging step of my life. And of course, I thank my friends from Xis (which will last forever) because they were very important all this time long.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS	7
LIST OF FIGURES.....	9
LIST OF TABLES	11
ABSTRACT.....	13
1 INTRODUCTION.....	15
2 BACKGROUND	17
2.1 XQuery	17
2.1.1 XQuery Data Model	17
2.1.2 Relevant XQuery Expressions.....	19
2.2 Brackit Pipeline	22
2.3 Brackit's Optimizer	23
2.3.1 Pipeline Optimization.....	24
2.4 Chapter Remarks	26
3 UNNESTING QUERIES	27
3.1 Unnesting Strategy	27
3.2 Importing Ideas to Brackit	29
3.3 Algebraic Rewriter	30
3.3.1 Normalization	31
3.3.2 Algebraic Pattern Recognition	32
3.3.3 Decision Tree.....	32
3.3.4 AST Rewriting	35
3.4 Chapter Remarks	38
4 EXPERIMENTAL EVALUATION	41
4.1 Experiment Environment	41
3.2 Evaluated Queries	41
4.3 Experiments Results.....	44
4.4 Results Analysis	47
5 CONCLUSION	49
REFERENCES.....	51

LIST OF ABBREVIATIONS AND ACRONYMS

SQL	Structured Query Language
XML	eXtensible Markup Language
DBMS	Database Management System
AST	Abstract Syntax Tree
XDM	XQuery Data Model
FLWOR	For Let Where Order by Return
CNF	Conjunctive Normal Form
IDE	Integrated Development Environment
CPU	Central Processing Unit

LIST OF FIGURES

Figure 1.1:	SQL Query with a nested query and the unnested version.	16
Figure 2.1:	XDM structure based on sequences of items.	19
Figure 2.2:	FLWOR expression example, with expression tree and conceptual analysis.	20
Figure 2.3:	Semi-join example based on path and filter expressions.	21
Figure 2.4:	High-level view of Brackit Pipeline.	22
Figure 2.5:	Converted operator tree after pipelining.	24
Figure 2.6:	XQuery and the respective AST, after optimization steps until 4-way left join generation.	25
Figure 3.1:	Example of a query with existential quantifier nesting.	28
Figure 3.2:	Example of a query with universal quantifier nesting.	29
Figure 3.3:	Example of a query with implicit grouping.	29
Figure 3.4:	Algebraic rewriter overview: query unnesting process.	31
Figure 3.5:	Converting predicate to CNF.	32
Figure 3.6:	Decision tree to unnest existential quantifiers.	33
Figure 3.7:	AST Rewriting using Equivalence 1, adding Count operator.	35
Figure 3.8:	AST Rewriting using Equivalence 2, based on Cartesian product.	36
Figure 3.9:	AST Rewriting using Equivalence 3, based on Semi-join.	36
Figure 3.10:	AST Rewriting using Equivalence 4, based on the comparison to the result of the aggregation function.	37
Figure 3.11:	AST Rewriting using Equivalence 5, based on Theta-join.	38
Figure 4.1:	Data model of Shakespeare's plays.	42
Figure 4.2:	Data model of publication catalogs.	42
Figure 4.3:	Response time comparison of the two versions of Query 1.	45
Figure 4.4:	Response time comparison of the two versions of Query 2.	46
Figure 4.5:	Response time comparison of the two versions of Query 3.	46
Figure 4.6:	Response time comparison of the two versions of Query 4.	46
Figure 4.7:	Response time comparison of the two versions of Query 5.	47
Figure 4.8:	Nested and unnested queries' running time comparison.	47
Figure 4.9:	Percentage performance gains.	48

LIST OF TABLES

Table 2.1:	Overview of expressions in XQuery 3.0.	18
Table 3.1:	Algebraic Patterns and examples in XQuery.	27
Table 3.2:	Defining aggregation function to use and the aggregation value in empty case, according to θ	38

ABSTRACT

Since the beginning of the 80's, during the rise of relational databases, it has been developed many strategies to deal with problems of executing queries in a nested way. Most of these strategies are based on classification into generic query types, followed by an unnest technique for each type. The two main approaches to unnest are: source level and algebraic level. The latter has some advantages, as expressiveness. These publications were important to the success of relational database architectures. However, all this knowledge is not only useful for relational databases, but also for queries from non-relational databases, e.g. XML databases. Brackit is an open-source XQuery compilation engine, developed at Technische Universität Kaiserslautern. The compilation pipeline in this engine includes an optimization stage where we could develop unnesting algorithms. Aiming optimize the query evaluation in Brackit, we present how we applied algebraic equivalences to unnest queries. The contribution of this work is the implementation of an efficient and high level unnesting technique, easy to understand and to improve. This implementation reduces the heavy code legacy of the current optimizer version. The basis of these equivalences application was the good correspondence between the algebra and Brackit's AST nodes. The optimization, thus, was based on AST manipulation. Here, we are going to call this manipulation as rewriting. For experimental analysis, it was performed simulations throw XQuery to prove the gain of the unnesting strategy.

Keywords: XQuery, unnesting, XML, database, subquery, optimization, algebraic, equivalence, rewrite

1 INTRODUCTION

The first publications on how to optimize query evaluation in DBMSs was focused on SQL queries (Kim 1982, Ganski 1987, Muralikrishna 1992). The reason for that was the increase of relational databases popularity, which brought up the need of strategies of how to decrease the time response of the queries since the data to deal with was getting bigger. Thenceforth, it has been pointed a main villain of query efficiency: the straightforward nested evaluation.

One problem of the nested query evaluation is the overhead of initializing the inner query and loading the inner table(s) multiple times, the data in such way that the physical access to these data increases quadratically or worse, in case of more than one subquery. There is also the issue of not exploring the join selectivity, beyond other improvements that the optimizer could do since there are fewer query levels, which could be used to decrease the amount of disk access. To illustrate the idea of nested query, Figure 1.1 gives a simple example in SQL, with an equivalent unnested version.

The first of these publications (Kim 1982) came up with a source level approach, classifying queries according to the class of the inner query and the relation between the inner query and the outer one. Then, for each class, it was specified a generic algorithm to unnest the query. Later, it appeared many other publications, with improvements on these ideas and bugs avoiding (Ganski 1987, Dayal 1987, Muralikrishna 1992).

All these publications were very important to the improvement of SQL itself because from these ideas came the importance of adapting the SQL with operators aimed to unnested query evaluation. Whereas the rise of relational DBMSs has brought an increase of data to manipulate, these improvements became indispensable for SQL popularity keep growing since it got more expressive and delivered users better response times. However, these concepts were not only useful for relational databases, but also for queries from non-relational databases, e.g. XML databases. Hence, those studies could be extended to unnesting queries in XQuery, a data programming language developed to manipulate XML-based data.

A good example of study at this line was developed at an algebraic level, whose some strategies were applied in this work. That study presented the Natix Algebra (May et al. 2006), which the XQueries would be translated to, and which the unnesting equivalence rules would be applied on, in order to obtain an optimized query.

There are advantages of using an algebraic view for applying unnesting techniques instead of a language level approach, which was the case of the first publications. Probably the main one is the generic structure of the results because they can be applied directly to any other query language translatable into the underlying algebra. Another advantage is the possibility of integration into a cost-based plan generation.

In summary, the unnesting strategies presented in that work were based on a translation to that algebra (which included a normalization step), followed by the application of algebraic equivalences, according to one of three decision trees. These decision trees were defined by the nesting type: existential quantifying, universal quantifying and implicit grouping.

Then, it has come the idea to bring these strategies into Brackit (Sauer e Bächle 2011), a data-independent XQuery compiler. Its compilation pipeline is divided into four steps: parsing (AST creation), analysis, optimization and translation (plan generation). Hence, our work goes into optimization step.

The importance of this work is that the current unnest strategies were using operators with an unnecessary level of complexity, what used to make it hard to work on the following pipeline steps, beyond the difficulty of improving the optimization itself. Then, it would be important to restructure the optimizer, where one of the optimizations would be focused on algebraic query unnesting. For the development of this new unnesting in Brackit, it is possible to point another fundamental advantage from the algebraic level within the strategies shown at that study: the good correspondence between the algebra and Brackit's AST structure.

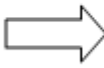
Therefore, the contribution of this work is the implementation of an efficient and higher level unnest technique for Brackit's optimizer, with algorithms easier to understand and to improve. Furthermore, this implementation reduces the heavy code legacy caused by the complex operators. To get that, it was adapted general strategies for unnesting queries for XML databases into the Brackit's system of AST rewrites, which is how most part of the query compilation process works in this engine. This adaptation consists on the implementation of the query normalization, the algebraic pattern recognition, the algebraic equivalence definition with a decision tree, and, finally, the AST rewrite matching the defined equivalence. To examine the performance improvement provided by this strategy, it was necessary to simulate the algebraic rewrite application. The obtained results give a preview of how useful these strategies can be, stimulating the continuation of this implementation in Brackit.

This work continues with the following structure: Chapter 2 introduces XQuery and describes the structure of the engine, detailing the compilation steps, and presents the current unnest technique existent in Brackit (Bächle 2013). Chapter 3 describes the proposal of this work that is the implementation of unnesting strategies on our engine, creating the algebraic rewriter. Chapter 4 reports the experiment method, explaining how the simulated evaluation was performed and the importance of the results. The final statements and the opening of possible future works are made in Chapter 5.

```

SELECT id
FROM books
WHERE author_id in ( SELECT id
                     FROM authors
                     WHERE age >50)

```



```

SELECT b.id
FROM books b, authors a
WHERE author_id = a.id
      AND a.age >50

```

Figure 1.1: SQL Query with a nested query and the unnested version.

2 BACKGROUND

The purpose of this chapter is to present Brackit (Sauer e Bächle 2011), the XQuery engine compiler, where the unnest techniques that will be detailed in this thesis were implemented. Here, we also come up with the current strategy for optimizing nested queries, focusing the pipeline optimization, which includes rewritings and new operators on the AST.

But before that, we begin it introducing the XQuery itself, presenting the fundamentals of XQuery, defining the XQuery Data Model (XDM) structure and describing the most relevant expressions for this thesis.

2.1 XQuery

XQuery is a powerful functional programming and querying language, created to manipulate XML-based data (Katz et al. 2003). Initially created to implement declarative XML processing as designed by a sequence of efforts (W3C 1998, Robie et al. 2000), but then several revisions were done, resulting a versatile functional programming language, with significant strength in XML bulk processing.

For the data manipulation, XQuery offers a large set of expressions, which includes the FLOWR expressions (query basis), path expressions, quantified expressions and many programming ones, such as comparisons, conditionals, arithmetic, references and function calls. Table 2.1 shows a complete overview of XQuery 3.0 expressions (W3C 2010b). XQuery is defined as a transformation on the XQuery Data Model, also called XDM (W3C 2010a). XDM follows some XML standards (W3C 2004, W3C 2006). In XQuery, a query is defined as a tree of expressions evaluated to a sequence of items.

2.1.1 XQuery Data Model

The concepts of sequence and item are the basis of XDM, as illustrated by Figure 2.1. A sequence is an ordered list of zero or more items. Even single data are treated as a sequence, i.e., values such as the integer 2 and the string “hello” are equivalent to the sequences (2) and (“hello”), respectively. Null values are defined by () – empty sequence – given that it is also used to represent missing data.

The items in XDM are classified into three classes: atomic types, nodes, and functions. Atomic types are related to primitive types, such as integers, floats, strings, etc, including the “untyped” type. Nodes are the standard abstraction for representing structured and semi-structured XML data, according to properties defined in the XML Infoset specification (W3C 2004). Those properties include the seven node kinds

(Document, Element, Attribute, Comment, Text, Instruction, and Namespace) and several kind-specific properties.

Expression Type	Examples
FLWOR Expression	for \$b in doc("products.xml") let \$limit := 100 where \$b.price lt \$limit order by \$b.price return \$b.id
Quantified Expression	some \$x in \$y satisfies \$x eq \$z (every)
Path Expression (XPath)	\$ec/person-group[@person-group-type='author']/name
Literal	123
Filter Expression	(4, 9, 2, 6)[.>5]
Variable Reference	\$total
Parenthesized Expression	(...)
Sequence Expression	"abc", \$x*2, foo(2)
Arithmetic Expression	12 + 3 (-, *, /, mod, div, idiv)
String Concatenation	"conca" "tenation"
Static Function Call	fn:foo("param")
Dynamic Function Call	\$fun("foo","bar")
Named Function Reference	fn:foo#1
Inline Function Expression	function(\$a as xs:double, \$b as xs:double) as xs:double { \$a * \$b }
Logical Expression	1 eq 1 and true (or)
Conditional Expression	if (\$x>9000) then "It's over 9000" else "ok"
Switch Expression	switch(\$x) case "a" return 1 case "b" return 2
Context Item Expression	.
Node Sequence Combination	\$seq1 union \$seq2 (intersect, except)
Node Constructor	<count-books> { count(\$book) } <\count-books>
Node Comparison	<a>5 is <a>5 (>>, <<)
Value Comparison	\$i/price eq 25 (ne, gt, ge, lt, le)
General Comparison	\$i/name = "John" (!=, <, <=, >, >=)
(Un-)ordered Expression	unordered {...} ordered {...}
Try/Catch Expression	try { foo() } catch * { "Some error" }
Range Expression	0 to 10
Instance Of Expression	0.7 instance of xs:float
Typeswitch Expression	typeswitch (\$team) case \$n as element(*, franchise) return \$n/owner case \$n as element(*, club) return \$n/president
Cast Expression	"0.7" castable as xs:double
Castable Expression	"0.7" castable as xs:double
Constructor Function	xs:boolean("true")
Treat Expression	treat \$nums as xs:integer+
Validate Expression	validate strict { doc('hamlet.xml') }

Table 2.1: Overview of expressions in XQuery 3.0.

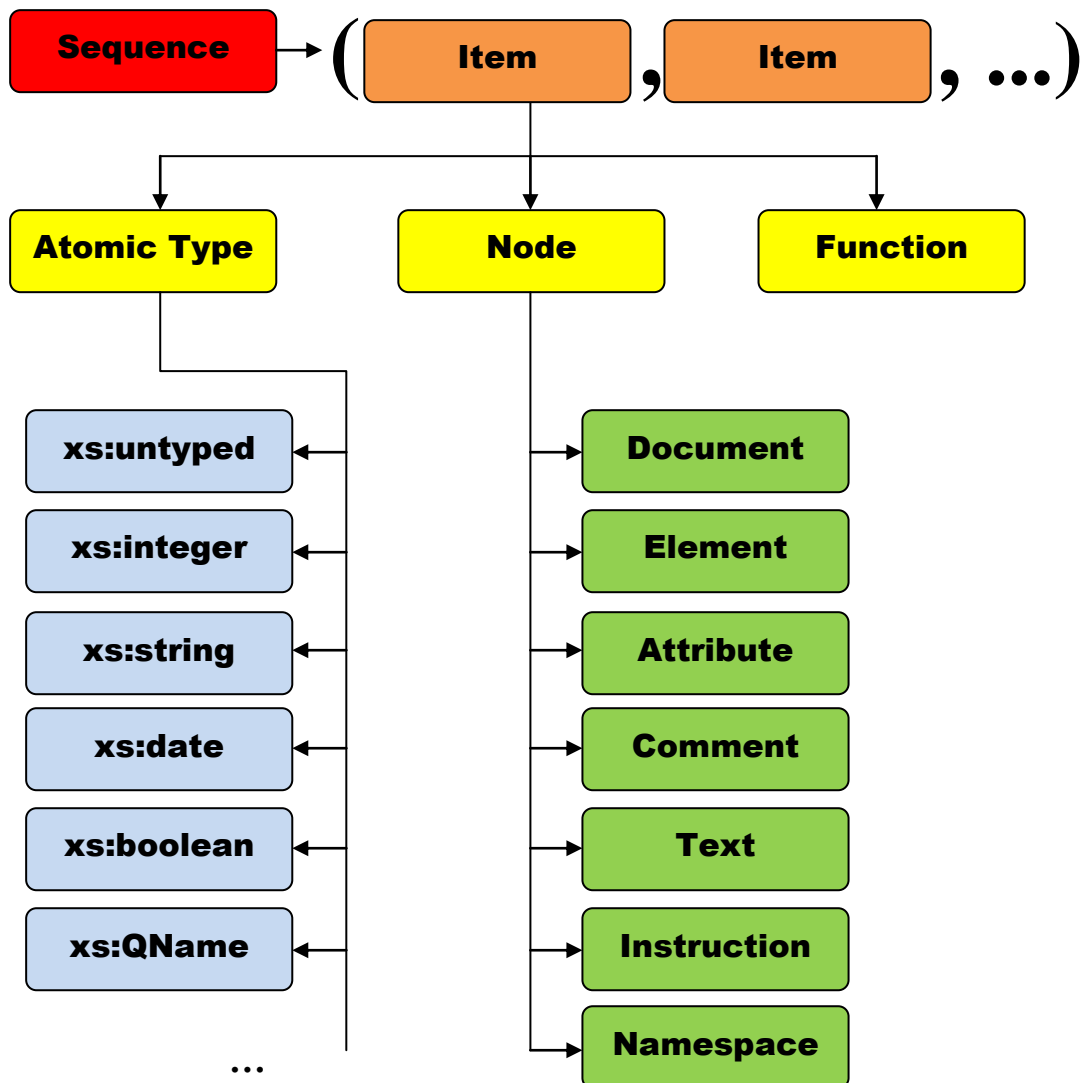


Figure 2.1: XDM structure based on sequences of items.

2.1.2 Relevant XQuery Expressions

As already seen in Table 2.1, XQuery has a good variety of expressions. For this thesis, we point out four very relevant expression types within XQuery: FLWOR expressions, quantified expressions, path expressions and filter expressions.

The basic tool for querying data in XQuery is the FLWOR expression (Katz et al. 2003). This name comes from the 5 clauses which originally compose this kind of expression: *for*, *let*, *where*, *order by* and *return* (some of them are optional). These clauses represent, respectively, the language primitives for iterating a sequence of tuples, mapping values to variables, filtering tuples, sorting and projecting the remaining tuples. The minimal FLWOR structure is a *for* clause followed by a *return* clause. Another property of FLWOR expression is that it must begin with a *for* or a *let* clause because these are the clauses which can make the first variable binding start the query's sequence stream. FLWOR expressions must also finish with a *return* clause because that is what reduces the sequence stream into a single XDM sequence. To illustrate all these concepts, it is used as example a simple FLWOR expression, with the

respective expression tree and a conceptual analysis of each clause, given by Figure 2.2 After some revisions, to increase the language expressiveness and to let it be similar to other data programming languages (like the ones for relational databases), it was also introduced other optional clauses: *group by*, *window* and *count* (W3C 2010b).

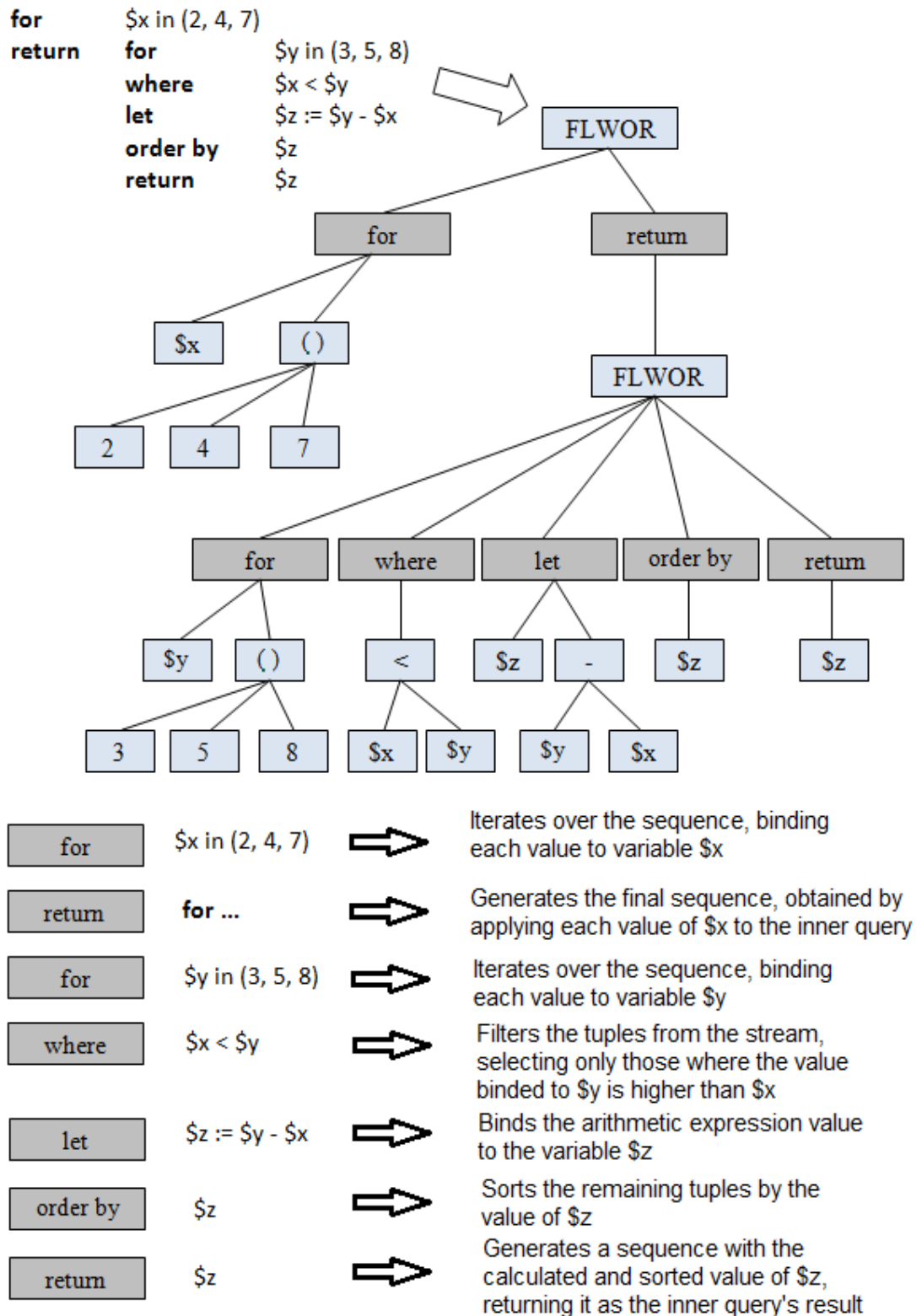


Figure 2.2: FLWOR expression example, with expression tree and conceptual analysis.

The second important kind of expressions is the quantified expression. It provides a boolean value according to the existentialism or the universality of a predicate in a collection, iterating over this collection. In a *some* quantified expression, it is returned true if exists a value inside the collection that makes the predicate which comes after the *satisfies* clause be evaluated as true. In an *every* quantified expression, it is only returned true if all values in the collection yield a true value for the predicate. The latter requires a special attention because they do not evaluate empty collections, since it would be discarded in the result instead of be evaluated as false.

Quantified expressions have a very high relevance in this work: they represent an important part of query nesting cases in XQuery. Furthermore, in this thesis, these nesting cases will receive most of the focus (especially the existential case). It is reasonable to consider that the compiler may be able to understand the semantics of quantified expressions, thus performing its evaluation not in the most naïve way, i.e., stopping the nested loop as soon as possible. Even though, it remains the problems of nested straightforward evaluations, mainly in cases of an unfavorable order of the collection.

The other two expression types previously highlighted, path and filter expressions, enable XML tree navigation and selection of values in a collection, respectively. The former's general structure is e_1 / e_2 , where e_1 and e_2 are individual subexpressions, but the evaluation of the second one is done in the first's context. The latter's general structure is $e_1 [e_2]$, where e_1 is a collection and e_2 is a position of this collection or a selective predicate, then the result of this type of expression is a collection with the items from e_1 which satisfies e_2 .

catalog-authors.xml:

```
<personal-data>
  <name>
    Name
  </name>
  <address>
    <country>
      Country
    </country>
    ...
  </address>
  ...
</personal-data>
```

lib.xml:

```
<book>
  <title>
    Title
  </title>
  <publication-info>
    <author-name>
      Name
    </author-name>
    ...
  </publication-info>
  ...
</book>
```

```
let $brazilian_author := doc("catalog-authors.xml")//personal-data[address/country='Brazil']
for $book in doc("lib.xml")//book[publication-info/author-name=$brazilian_author/name]
return $book/title
```

Figure 2.3: Semi-join example based on path and filter expressions.

Together, those two kinds of expressions are a useful tool for obtaining specific items from an XML database. Moreover, the combination of those expressions can be used to simulate a semi-join operation, and that is where it comes the relevance of those expressions for this thesis (more specifically, in Chapter 4, where the experimental analysis is done). For instance, let us consider the XML data structure and the query given by Figure 2.3.

The given query begins getting the author nodes whose country is Brazil. Then it performs a semi-join on the book nodes from the document “lib.xml”, selecting those whose author’s name matches one of the names of Brazilian authors obtained previously.

2.2 Brackit Pipeline

The pipeline of Brackit engine for XQuery processing is composed of 4 steps, as shown in Figure 2.4: parsing, analysis, optimization, and translation. The first step consists in a syntactical evaluation of the query (given by a string), where it is created an abstract syntax tree (AST) with each expression being represented by a node in this tree. This AST is the object which will be manipulated in the following steps through rewritings until translation, where the plan generation will take place. The second step performs the semantic analysis of the query, where it is done the static typing.

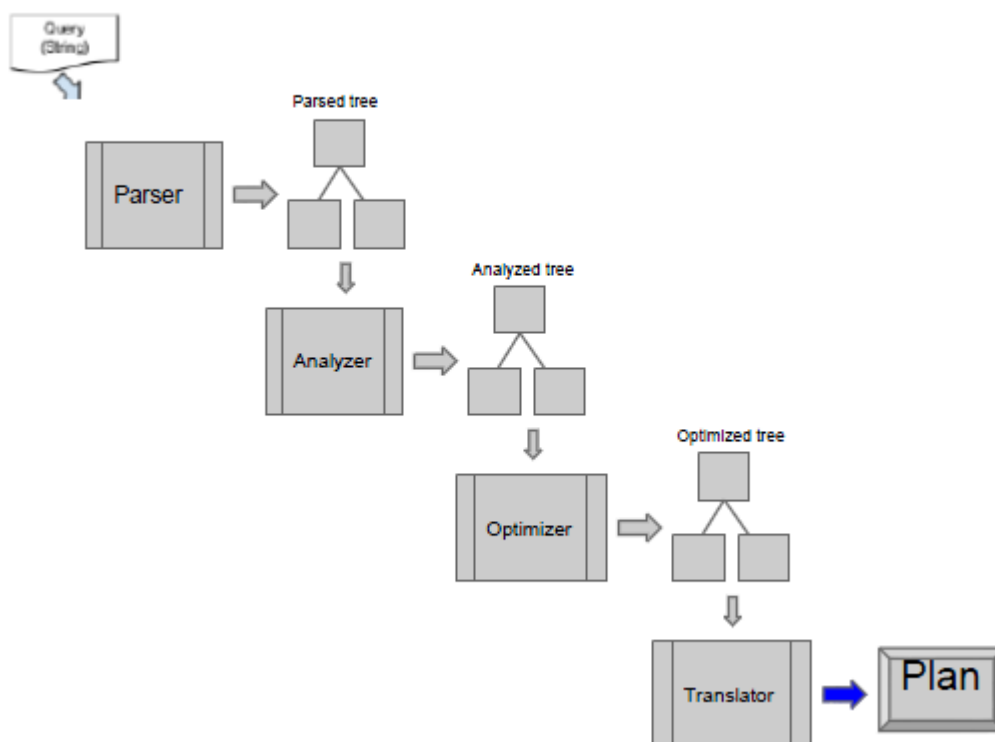


Figure 2.4: High-level view of Brackit Pipeline.

The third step is actually a group of smaller steps towards the query optimization. This group varies according to the compiler version. The current version, based on a pipeline lifting and using a 4-way Left Join (Bächle 2013), is detailed in section 2.3. It is also important to point that here is the relevant step for this thesis.

In the last step, it is generated an executable plan from the final AST, where it is assigned a concrete implementation for each expression type, compiling the expression tree in a single pass, from the root to the right-most leaf. The translation process is quite simple because each subtree in the AST is self-contained and the plain variable references are the only dependencies. Accordingly, variable bindings are totally managed by its own expression or operator. In this process, the variable must be declared only before the respective sub-expression and undeclared at the end of its scope. In the case of variable dependency, a simple lookup in the variable table is enough to resolve the corresponding position in context tuples.

The stable version of Brackit performs the query compilation based on its top-down perspective. In this version, the translation step creates a push-based pipeline, where each operator act as a sink, generating a tuple stream as output after receiving another tuple stream.

But there is also the bottom-up compiler version, where the translator generates a pull-based pipeline, which is very similar to the other one, except for each operator being realized as a cursor, implementing a simple open-next-close interface (Graefe 1994), which works in the opposite direction. In this translation, the cursor pipeline is initialized with a *Start* operator, and it uses this cursor to propagate the call to the last operator in the pipeline through the individual pipeline stages.

As known, when a query is seen in an algebraic level, e.g. relational algebra or Natix Algebra (May et al. 2006), it is seen in a bottom-up perspective. For this reason, it was chosen the bottom-up version of Brackit compiler to implement the algebraic unnest (as it is going to be shown in Chapter 3), because then it was possible to apply the unnest strategies via AST rewritings.

2.3 Brackit's Optimizer

In this section, it will be described the optimization stage of the stable version of the compilation pipeline in the engine, which we may call the state-of-the-art on query optimization in Brackit. As commented in the previous section, the optimization step of the Brackit's pipeline can be defined as a set of minor steps. In an ideal compilation optimizer, such set includes simplification, pipelining, pipeline optimization, data access optimization, parallelization, and distribution. Most of these are present in Brackit. For this thesis, the focus goes to the pipeline optimization, where it is performed join unnesting using a 4-way Left Join (Bächle 2013), whose complexity is one of the points of restructuring the optimization in Brackit through algebraic unnesting.

The first step at Brackit's optimizer is the simplification, where it is done some basic pruning operations, like dead-code removal. In the following step, FLWOR clauses are converted into operator pipelines. To illustrate this procedure, Figure 2.5 brings the respective pipelined tree from the expression tree given by Figure 2.2. The importance of pipelining is the possibility applying set-oriented rewritings on the operator tree, including optimizations from the relational world. Such optimizations will be more detailed in the sequence of this chapter.

After the optimizations on the pipeline operators, the optimizer in Brackit works on the physical data access level, with rewritings performed specifically to the target platform, introducing optimizations in multiple granules and multiple dimensions.

Examples of such operations are eager value coercion, indexing and the inclusion of a multi-bind operator, which reduces many binding twig branches to a single twig operator. And finally, the last optimization step improves parallel processing.

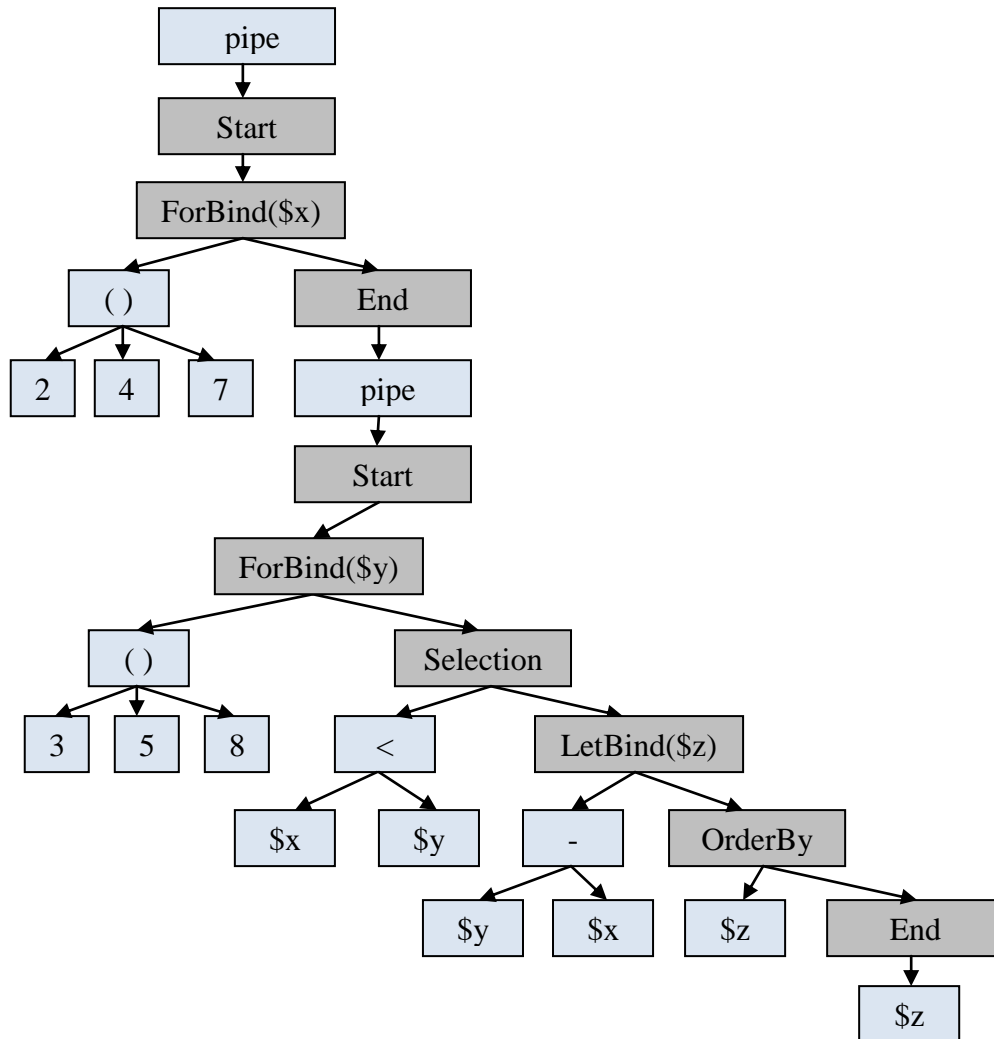


Figure 2.5: Converted operator tree after pipelining.

2.3.1 Pipeline Optimization

As already commented, the pipelining process enables the optimizer to apply several modifications on the data operation flow based on set orientation, always without changing the outcome. In Brackit, the rewritings are mainly focused on increasing how join operations perform, since these operations use to be the bottleneck of query processing, and this condition depends a lot on the way the join is evaluated.

The first operation in pipeline optimization is the pull-up of select predicates, in the same way that one may consider, in relational algebra, for pulling up a Selection before a Join/Cartesian product operation. For this process, it is analyzed the Select subtree, checking whether the predicate is independent of any upper operation, and then putting the Select to be evaluated before that. After this process, it begins the join optimizations. Naturally, the first thing to do is to recognize joins in the AST. The

importance of such procedure is the same as from relational data processing. Here, multiple for-bindings work as cartesian products, resulting on the undesirable nested-loops. Hence, it is primordial to find the join possibilities and adapt the AST, in order to filter the tuple streaming. The relevance of this process increases as XQuery, in contrast to SQL, does not have specific operators for deploying joins explicitly, which means joins will always be defined implicitly, hidden under *for* clauses. Accordingly, the next step involves the insertion of join, which is detected in the case of a Cartesian product immediately followed by a Select that correlates items from each branch of the Cartesian product. This process is followed by the push-down of any independent binding placed before the join. However, it is important to note that not always it is possible to apply such push-down operation due to the complex binding structure. Hence, it is also added a join group demarcation stage, which searches these complex binding occurrences and inserts a Count operator between the variable binding and the inner join, in order to trigger a rebuild of the lookup table used by the join only when necessary, i.e., the bound variable has changed. This mechanism defines sequential groups of individual nested operations that can share intermediate results.

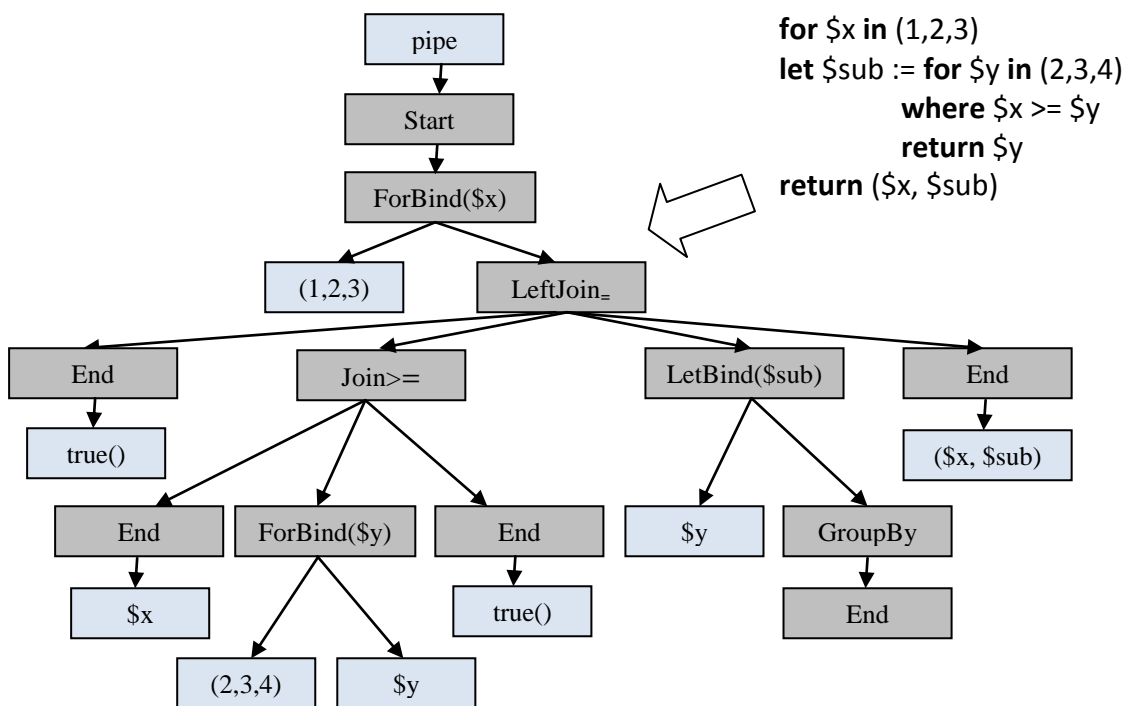


Figure 2.6: XQuery and the respective AST, after optimization steps until 4-way left join generation.

All those pipeline optimizations described above deal only with for-bound variables, which may form cartesian products that could (and should) be transformed into a join, or which could be manipulated to avoid unnecessary recomputations. However, XQuery allows nesting FLWOR expressions in *let* clauses, binding a whole sequence of values,

e.g., subqueries, to a variable. The problem is that, in this case, previous evaluations of the nested FLWOR and lookup tables are lost, since pipeline sharing would violate the expression evaluation independence. To manage this situation, it was developed the Pipe Lifting. And here is the place where the optimization structure gets a bit complex. This process is composed of three steps that “lift” the pipelines nested inside a let-bind.

The first step is the basic one: conversion of let-bind into a 4-way left join. This new operator is actually a macro, which is implemented by small operations to produce the lifting, executing: triggering of rebuild on the lookup table, filtering, concatenation of pipeline partitions and grouping. Further the 3 children that a normal join operator has in the AST (one for each input and the last one for the output), this special join has an additional child branch, related to the group generation. Once this conversion is done, one of left join inputs is the whole nested loop, and the other is empty, which means that it only echoes the tuple. To exemplify this conversion, let us consider the XQuery in Figure 2.6 and the respective AST, after all optimizations steps commented until now, including the 4-way left join conversion.

The following two steps of Pipe lifting aim to simplify the left join structure generated previously, adapting it without that empty operator. To obtain that, it is made a swapping of the nested for-bind to an extra left join, and then the initial left join, which is unnecessary at this point, is removed. After that, the pipeline optimization finishes with some rewritings on join trees and *group by* aggregation.

As one may notice, the previous steps of pipeline optimization have increased considerably the complexity of the AST that is manipulated, adding extra children to the operator nodes on the tree and several special operations. Accordingly, the implementation of further optimizations becomes harder, in different aspects: the walking process on the AST structure to visit the nodes and define rewritings or conversions need to consider those new exceptional operations and the new range of possible node children. In other words, it gets necessary to deal with some heavy code legacies in order to develop newer optimization steps. For this reason, it has come the idea to restructure the Brackit optimizer.

2.4 Chapter Remarks

In this chapter, it has been introduced the XQuery language, presenting the main concepts, with an overview on the sequence-based structure of XQuery Data Model and the language expressions. It was given a particular attention to four kinds of expressions in XQuery: FLWOR expressions, because they are the backbone of any query; Quantified expressions, because they correspond to the nesting type with more focus in our unnesting strategies, detailed in Chapter 3; Path expression and filter expression, because they, together, enable us to simulate semi-join operations, which was necessary to obtain the results on Chapter 4.

The Brackit engine, an XQuery compiler, was also presented in this chapter. It was introduced Brackit’s compilation pipeline, detailing the optimizer, where the algebraic equivalence rules for unnesting queries were applied, which implementation is described in next chapter. Here, it was detailed the state-of-art version of the optimizer, which includes several complex rewritings on the query’s AST, creating some trouble to add further optimizations, and then encouraging us to restructure the pipeline optimization in Brackit as follows in Chapter 3.

3 UNNESTING QUERIES

The objective of this chapter is to explain how the algebraic unnesting on Brackit optimizer was implemented, beginning the restructuring aspired in the previous chapter to reduce the AST manipulation and to simplify further optimizations. Initially, it is introduced the unnesting strategies for XML databases based on the translation of XQueries into an algebra, whose ideas are relevant for our work. Then, the motivations to bring these ideas to Brackit are presented, pointing out advantages and disadvantages of applying such strategies on the Brackit compilation pipeline, in order to obtain an optimized query evaluation. Finally, it is described how we implemented the algebraic rewriter, explaining how it works, detailing each step: normalization, algebraic pattern recognition, the decision tree – which defines the rewrite equivalence rule – and the rewrite itself.

3.1 Unnesting Strategy

As previously commented, lots of studies have been done towards query unnesting, with different approaches focused on distinct databases. We highlight one that is focused on XML databases, which is a framework based on unnesting queries after a translation into the Natix Algebra (May et al. 2006). This algebra includes some operators known from the relational algebra, e.g., Selection and Projection, and some new operators, e.g., Map and Unnest-Map, working on sequences of tuples. These four operators are the basis of the translation because they represent the clauses *where*, *return*, *let* and *for*, respectively.

	Algebraic Pattern	Example in XQuery
Existential Quantifier	$\sigma \exists x \in e_2 : p (e_1)$	where some \$x in e ₂ satisfies p
Universal Quantifier	$\sigma \forall x \in e_2 : p (e_1)$	where every \$x in e ₂ satisfies p
Implicit Grouping	$\chi g : f(\sigma p (e_2)) (e_1)$	let \$g := for \$a in c [e ₂] where p return r [f]

Table 3.1: Algebraic patterns and examples in XQuery.

In this research, the queries pass by a normalization phase, which transforms the query to simplify the algebraic pattern recognition and to enable the application of more equivalence rules. This normalization consists of steps like the break of any complex expression into simple ones (creating new variables for sub-expressions), the transformation of implicit computations into explicit ones and the normalization of predicates into conjunctive normal form (CNF). After normalization, the nested queries obtained from the translation from XQuery to that algebra are classified into three types: existential quantifiers, universal quantifiers, and implicit grouping. The first and the second are characterized by the occurrence of a quantified expression while the last occurs when the values returned by a FLWOR expression are aggregated into tuples from an outer one. This last one is called implicit grouping because it used to be the only way to perform grouping since explicit grouping was not provided originally in XQuery. Table 3.1 shows the three algebraic patterns and examples in XQuery, where the element e_1 corresponds to the rest of the query and is omitted in the examples.

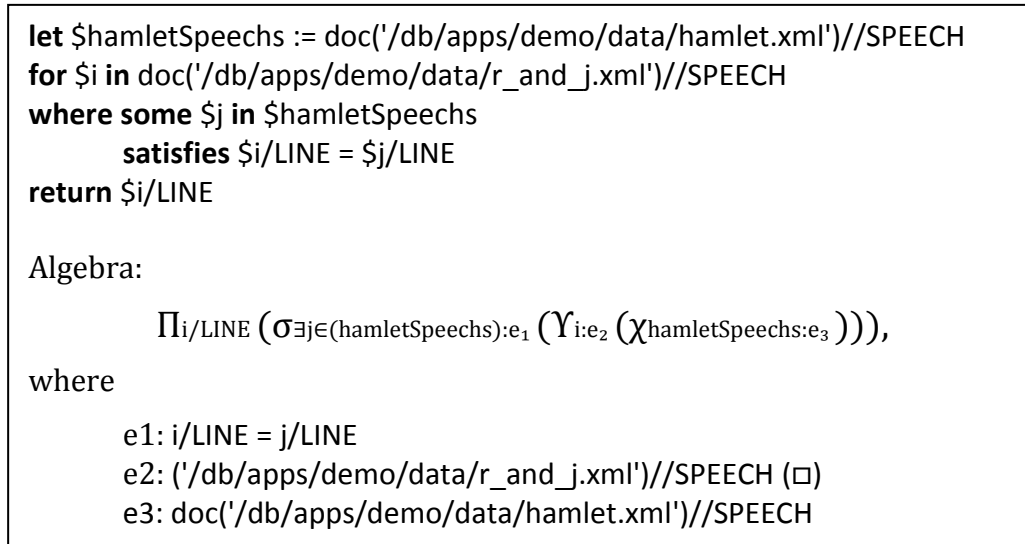


Figure 3.1: Example of a query with existential quantifier nesting.

To illustrate the classification, let us consider the Figure 3.1, Figure 3.2 and Figure 3.3. The first query, with existential quantifier, looks for speeches from “Romeo and Juliet” (r_and_j) that has a line equal to a line from speeches from “Hamlet”, returning the lines of that speech. The second one, with a universal quantifier, performs an anti-join: it is searched citations from 2014’s publications whose authors does not appear in 2013’s publications. The query used to illustrate implicit grouping is the same from Figure 2.6, used to show the 4-way left join insertion.

For each query nesting type, there is a list of algebraic equivalence rules that can be applied to perform the query unnesting, i.e., to get a query equivalent to the first one that does not result in a naïve nested straightforward evaluation. To choose the equivalence rule(s), each query class has a decision tree, which according to some characteristics of the query, determines the best equivalence to be applied. Also, it includes some auxiliary equivalences, which may be used to manipulate the algebra if necessary.

```

for $i in doc('publications-2014.xml')//element-citation
where every $j in doc('publications-2013.xml')//element-citation
    satisfies ( $i/author/surname != $j/author/surname
                or $i/author/name != $j/author/name)
return $i

```

Algebra:

$$\Pi_i (\sigma_{\forall j \in e_1: e_2 \vee e_3} (Y_{i: e_4} (\square))),$$

where

```

e1: doc('publications-2013.xml')//element-citation
e2: i/author/surname != j/author/surname
e3: i/author/name != j/author/name
e4: doc('publications-2014.xml')//element-citation

```

Figure 3.2: Example of a query with universal quantifier nesting.

```

for $x in (1,2,3)
let $sub := for $y in (2,3,4)
    where $x >= $y
    return $y
return ($x, $sub)

```

Algebra:

$$\Pi_{(x, sub)} (\chi_{sub: (\Pi y (\sigma_{x \geq y} (Y_{y: (2,3,4)} (\square))))} (Y_{x: (1,2,3)} (\square)))$$

Figure 3.3: Example of a query with implicit grouping.

3.2 Importing Ideas to Brackit

In the end of Chapter 2, it has been introduced one of the main goals of this work, which is the achievement of higher level optimizations, with reduced complexity and heavy code legacy, in comparison to the state-of-art version of Brackit's optimizer. Another factor is that the top-down perspective used by that optimizer may not be considered the most intuitive view of the query to check out optimization algorithms and to manipulate. The usage of algebraic unnesting strategies, then, seemed to be a good approach to implement on the optimization pipeline of Brackit, because they do satisfy pretty well those targets.

The main advantage of using those strategies to develop query optimizations is the good correspondence between the algebra and the Brackit operators. Accordingly, the application of algebraic unnesting equivalence rules could be performed via AST manipulation. This manipulation, as better detailed in the sequence of this chapter, is

implemented in the form of rewrites of the operator tree. Furthermore, it is also important to highlight the fact that good correspondence means that there are only few physical operators of the algebra which need to be implemented. For the existential quantifiers nesting case, the implementation of three physical operators would be enough: Semi-join, Theta-join and Cartesian product.

On the other hand, there is the fact that the bottom-up version of Brackit, which fits perfectly on the algebra, needs several complex code fixes to be able to evaluate queries. So the only stable version of Brackit works in a top-down fashion, which goes against the algebra structure, that is bottom-up by definition. For this reason, it was necessary to improvise a method to check the performance gains on applying these algebraic equivalence rules, which is discussed on Chapter 4. Despite that situation, we still decided to implement these unnesting strategies in Brackit, considering the advantages listed previously.

The Brackit compilation process, as explained in Chapter 2, is composed of parsing, analysis, optimization, and translation. In this process, once the AST relative to the query is created and semantically analyzed, this AST is used at the optimization proceeding. The optimizer operation manipulates the AST in such way that the resulting AST whose translation will generate an execution plan with higher performance (in most cases) than the original AST would do. This manipulation can be seen as rewritings of the original AST based on strategies to optimize queries, inserting and removing nodes, modifying operators, changing the order of the operators, etc. The idea of implementing optimization strategies throw AST rewrites is the basis to understand how the unnesting strategies were developed on Brackit.

From the three nesting classes described above, the chosen to be the focus of our work was the existential quantifier case. The factors of this decision were:

- the complexity to recognize the algebraic pattern and to apply equivalence rules;
- the relevance of the nesting type, i.e., how often the nesting type occurs in normal queries.

Regarding the complexity of the nesting type, naturally the first case to be implemented would not be the most complex one. Accordingly, the chosen case was not the implicit grouping because the algebraic pattern is not so directly obtained from an arbitrary XQuery as the quantifiers cases are. And concerning the relevance of the nesting classes, the existential case corresponds to a very common expression type, standing out for being largely used to perform implicit semi-joins. In the universal case, though, the main usage is probably to perform implicit anti-joins, which is a not so common query type. Hence, the implementation of universal quantifiers and implicit grouping rewritings has become a matter for future work.

3.3 Algebraic Rewriter

The algebraic rewriter, as Figure 3.4 illustrates, is composed of 4 procedures: normalization, algebraic pattern recognition, choice of equivalence rule with a decision tree and tree rewriting. The special arrow on the unnesting pipeline refers to the fact that, as explained in the sequence of this chapter, part of the normalization can be done just before the decision tree.

3.3.1 Normalization

Most of the normalization steps proposed on that research to precede the translation aim to favor the implicit group recognition and to apply its equivalence rules. Considering that the full implicit grouping implementation is not in the scope of this work, it was decided to only provide one normalization step, which is relevant to unnest existential quantifiers.

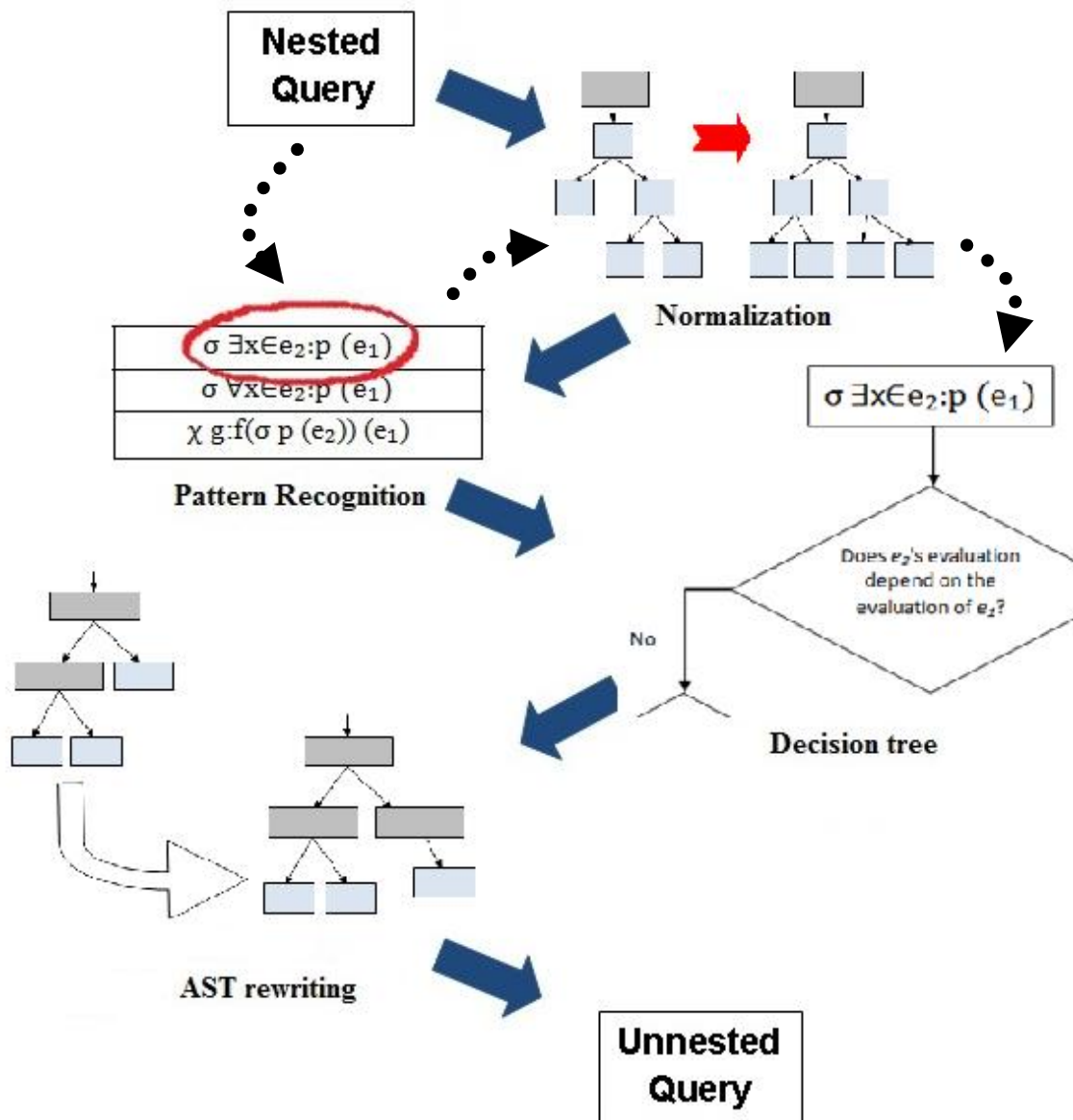


Figure 3.4: Algebraic rewriter overview: query unnesting process.

The normalization operation developed for the algebraic rewriter is the conversion of non-CNF predicates inside a Selection to CNF (conjunctive normal form) predicates. A CNF predicate is a predicate composed of 0 or more conjunctions of clauses, where the clauses are composed of 0 or more disjunctions of literals (bringing this definition to XQuery, a literal can be expressions like comparison, function, boolean value, etc.).

This conversion is based on logical properties, like commutative, distributive and Morgan's laws. Figure 3.5 shows a simple AST conversion example. The importance of this normalization step is that it enables the separation of one conjunction branch from the rest of the predicate. This is useful to discover a better equivalence rule at the decision tree or to filter the tuple flow. It means that the conversion can be made just before the decision tree, which is actually done in Brackit. The advantage of doing this normalization after the algebraic pattern recognition is that after recognizing the pattern, we already know where the predicate to be normalized is.

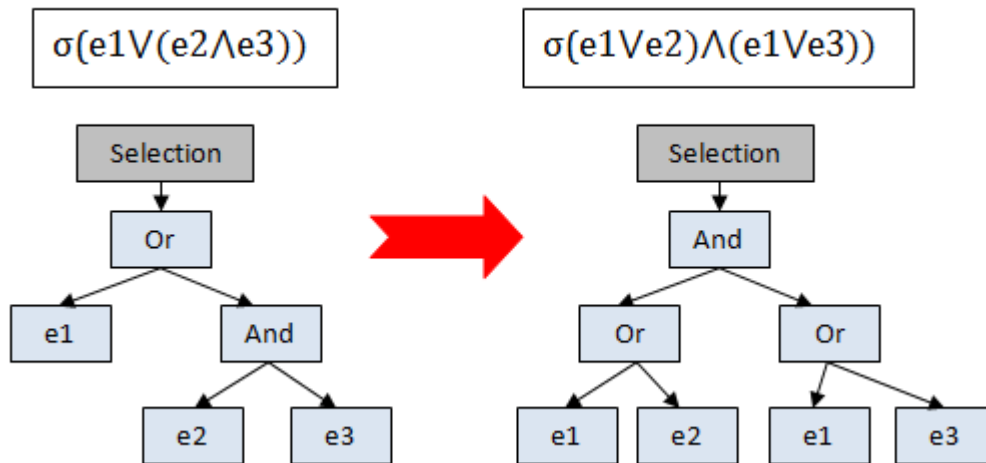


Figure 3.5: Converting predicate to CNF.

3.3.2 Algebraic Pattern Recognition

The pattern recognition, for quantifier cases, is a very simple step. The occurrence of existential quantifier nesting is given by the presence of a *some-expression* node at the AST, because the rest of the algebraic structure is just how the syntax of this kind of expression is defined in XQuery. Hence, this pattern can be recognized on a simple AST scan. The same works for universal quantifiers.

The implicit grouping pattern is recognized if there is a *let-bind* node followed by a new pipe or a function call. The problem is that many further normalization steps are needed to let all the queries with implicit grouping be recognizable with this pattern.

3.3.3 Decision Tree

The next step consists on checking which algebraic equivalence rule to unnest the query is applicable on the normalized AST. To do this, it is used a decision tree, which analyzes aspects of the query like the type of selection predicate and variable dependencies between the inner and outer query. For each query nesting class, there is a correspondent decision tree. This is the reason for recognizing the algebraic pattern in the previous step.

To exemplify this process, it will be considered the existential quantifier unnesting. Figure 3.6 shows the decision tree to unnest this kind of query.

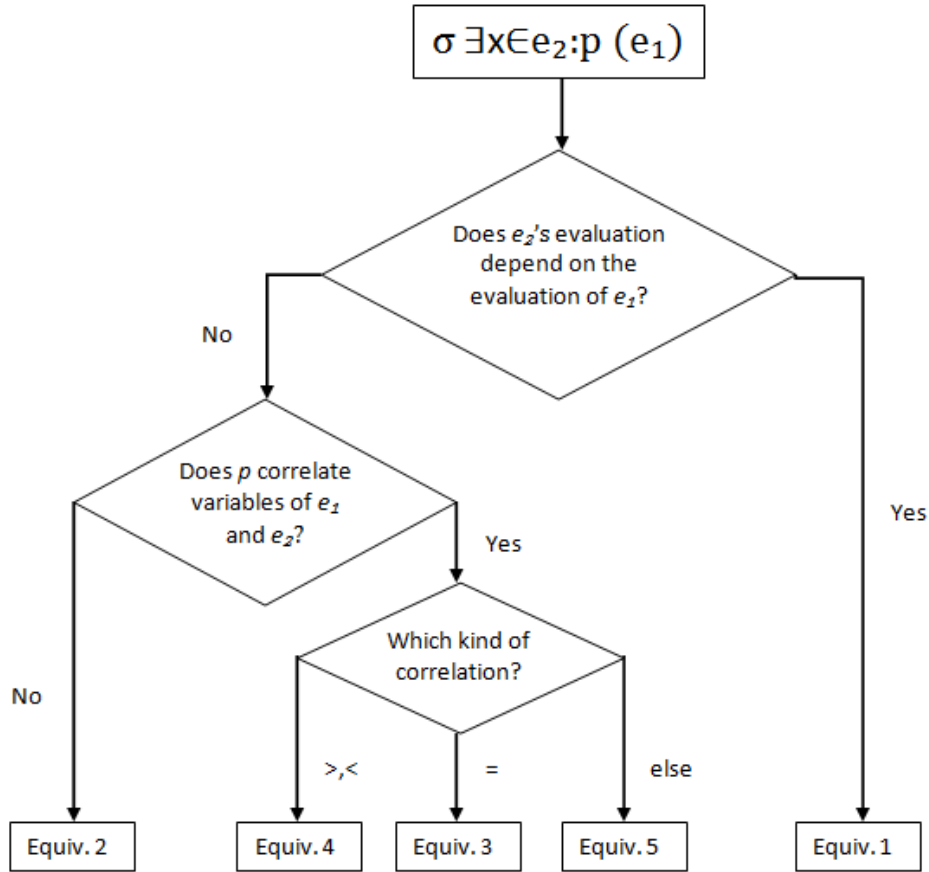


Figure 3.6: Decision tree to unnest existential quantifiers.

The first aspect to be checked is whether the evaluation of the inner (e_2) query is variable dependent on the outer (e_1) evaluation. This is done by verifying if the set of free variables in e_2 (i.e., references in e_2 which are not bound, and thus the value must be provided by the outside) and the set of bound variables (context coming from e_1) have any elements in common. If so, this means that it is not possible to unnest, and the only optimization to do is the elimination of duplicates, obtained applying Equivalence 1. If not, the next aspect to check is the Selection predicate (the p from the algebraic pattern). If it does not correlate variables of e_1 and e_2 , it is not possible to use any kind of join to unnest the query, and thus we apply the Equivalence 2, which is based on a Cartesian Product. Although, if it does correlate them, we verify how the correlation is done. It is important to note that the predicate can be rearranged to find a better unnesting equivalence. If there is a comparison by equality between variables of e_1 and e_2 , we can apply the Equivalence 3, which is based on Semi-join. If there is a comparison with “higher than” or “less than” operator between them, we can instead use an aggregation function to unnest the query. The idea of this unnesting is that it is possible to get the correct boolean value of the query with only the minimal or maximal value coming from e_2 . Considering the algebraic pattern, given by the expression

$\sigma \exists x \in (\sigma_{A_1 \theta A_2}(e_2)): p(e_1)$, where $\theta \in \{<, \leq, >, \geq\}$, A_1 and A_2 are variables respectively coming from e_1 and e_2 ,

it is reasonable to define that tuples from e_1 satisfies the existential quantification predicate if A_1 is in the range $[\min(A_2), +\infty)$ or $(-\infty, \max(A_2)]$, according to θ . And

finally, if the correlation is done without any of the comparison types commented above, we can only apply the Equivalence 5, which is based on Theta-join. To illustrate the application of these equivalences, it is shown below examples of queries that would fall into each type.

QUERY FOR EQUIVALENCE 1:

```
let $aug2014 := doc('/db/apps/demo/data/drugs_2014_aug.xml')//element-citation
let $feb2014 := doc('/db/apps/demo/data/drugs_2014_feb.xml')//element-citation
for $i in $feb2014
where some $j
  in $aug2014[person-group/name/surname=$i/person-group/name/surname]
  satisfies $i/year gt $j/year
return <result> { $i/person-group/name/surname, $i/year } </result>
```

QUERY FOR EQUIVALENCE 2:

```
let $users := doc("users.xml")//usertuple
let $items:= doc("items.xml")//itemtuple
let $bids:= doc("bids.xml")//bidtuple
for $i in $users
where some $j in $items
  satisfies some $k in $bids
  satisfies ($i/userid eq $k/userid and $j/itemno eq $k/itemno)
return $u/name
```

QUERY FOR EQUIVALENCE 3:

```
let $aug2014 := doc('/db/apps/demo/data/drugs_2014_aug.xml')//element-citation
let $feb2014 := doc('/db/apps/demo/data/drugs_2014_feb.xml')//element-citation
for $i in $feb2014
let $ipns := $i/person-group/name/surname
let $iy := $i/year
  where some $j in $aug2014 satisfies
  let $jpsn := $j/person-group/name/surname
  let $jy := $j/year
  return $ipns = $jpsn and $iy eq $jy
return $i
```

QUERY FOR EQUIVALENCE 4:

```
let $hamletSpeeches := doc('/db/apps/demo/data/hamlet.xml')//SPEECH
for $i in doc('/db/apps/demo/data/r_and_j.xml')//SPEECH
where some $j in $hamletSpeeches satisfies count($i/LINE) gt count($j/LINE)
return $i/LINE
```

QUERY FOR EQUIVALENCE 5:

```
let $hamletSpeeches := doc('/db/apps/demo/data/hamlet.xml')//SPEECH
for $i in doc('/db/apps/demo/data/r_and_j.xml')//SPEECH
where some $j in $hamletSpeeches satisfies contains($i/LINE, $j/LINE)
return $i/LINE
```

3.3.4 AST Rewriting

Once it is known the equivalence rule to be applied, we map this algebraic equivalence to a determined rewrite on the AST. This is done by inserting operators of Brackit that correspond to the algebraic expression and restructuring the AST, according to the equivalence. The best unnesting levels are obtained with rewrites based on equivalences 3, 4 and 5. To illustrate the application of the equivalences, it will be considered the generic ASTs.

In the first equivalence, taking into account that the query actually cannot be unnested, it is only added a Count operator to the AST, just before the variable binding of the e_2 (the x from the algebraic pattern). This operator – introduced in Chapter 2, where some operators employed to optimize queries were described – is a data programming language operator that enumerates a relation and attaches to each tuple its position in the input relation (Bächle 2013). The basic idea of adding this operator is to eliminate duplicates. However, depending on the nesting structure, it may also a significant performance gain by avoiding unnecessary recomputations. Moreover, it can be used on further optimizations that explore parallelism. Figure 3.7 shows the algebraic equivalence and the respective AST rewriting in Brackit.

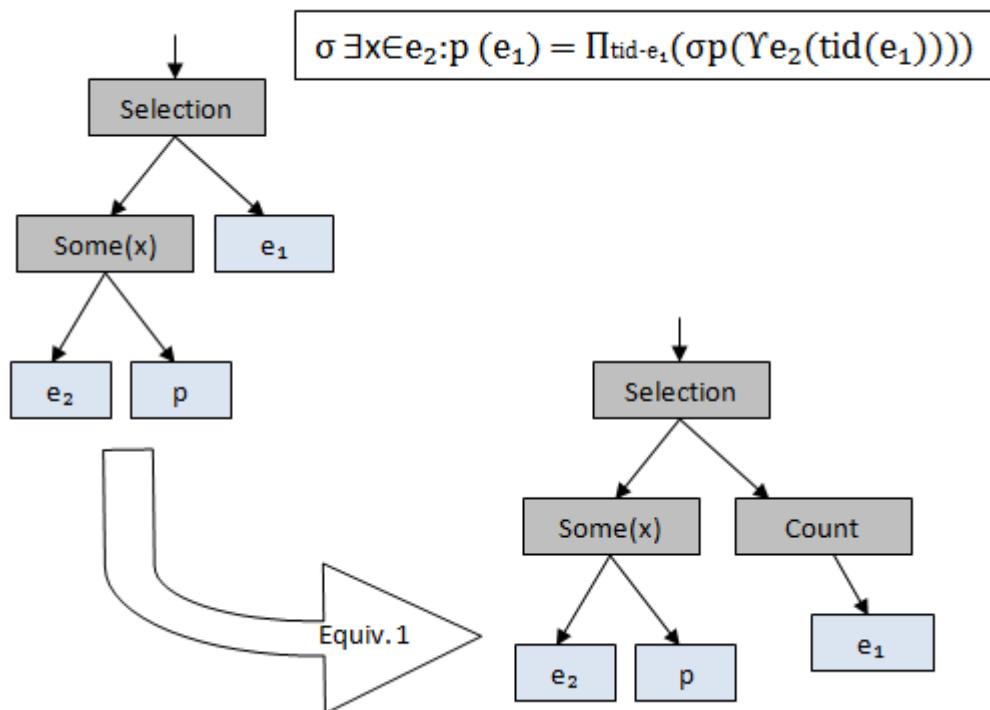


Figure 3.7: AST Rewriting using Equivalence 1, adding Count operator.

The Equivalence 2 is based on a Cartesian product. Figure 3.8 illustrates it. Firstly, it is added a Count operator for the tuples coming from e_1 in the same way as done in the first rewriting, which provides the same benefits. Then, it is made a Cartesian product of the tuples coming from e_2 (whose values are bound to the variable 'x') and e_1 (which includes the binding of the variable 'y'), followed by the filtering according to the selection predicate. Finally, it returns only the values generated in e_1 .

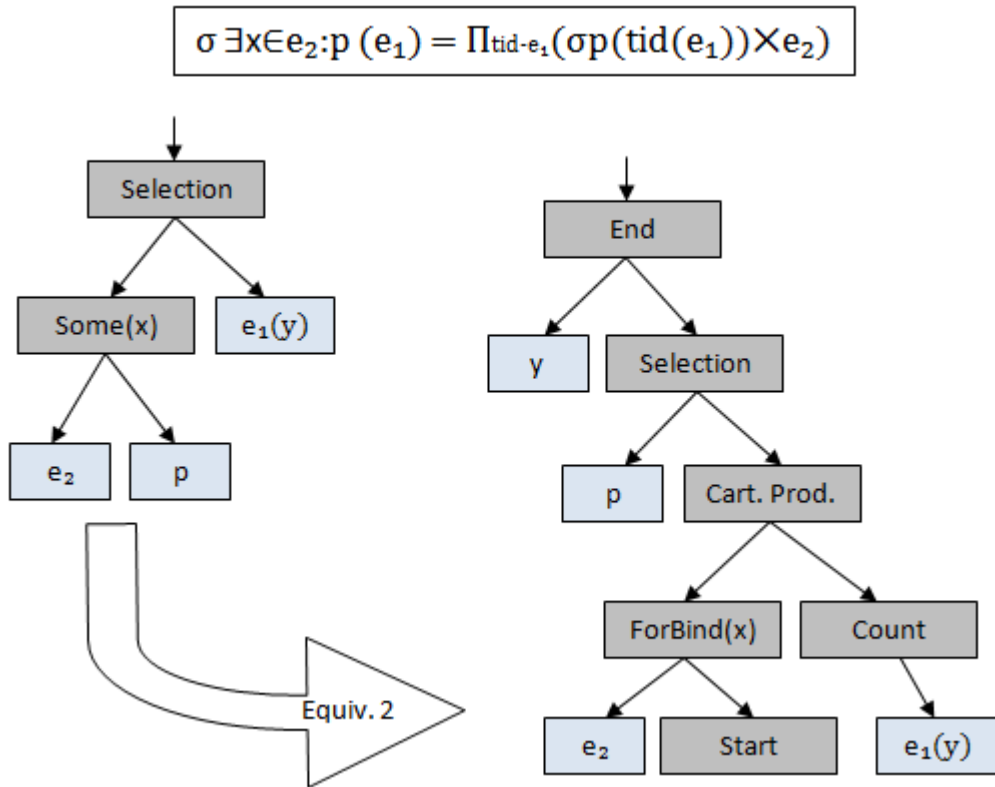


Figure 3.8: AST Rewriting using Equivalence 2, based on Cartesian product.

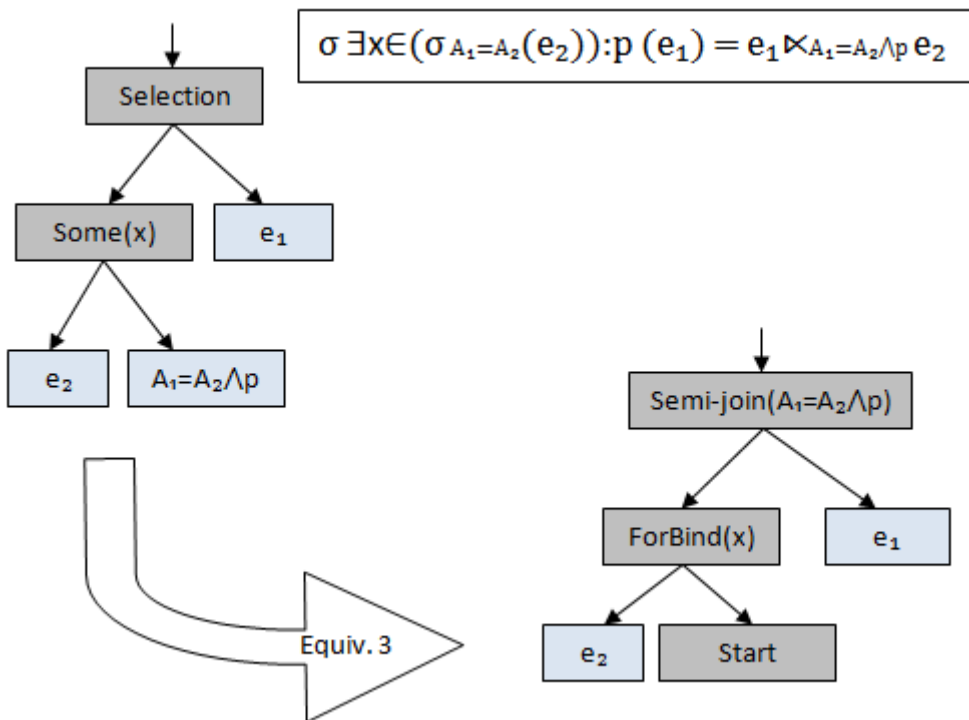


Figure 3.9: AST Rewriting using Equivalence 3, based on Semi-join.

The significant performance improvements with unnesting rewrites begin here, with the rewrite based on Equivalence 3. It basically adds an iteration over e_2 and a Semi-join operator. The Semi-join filters the tuples in accordance with the equality comparison, selecting also by the rest of the predicate. Figure 3.9 illustrates that.

The best optimization result is got with the Equivalence 4, due to the fact that it is necessary only one value from e_2 to be compared to the values from e_1 . This value is obtained with the respective aggregation function after iterating over e_2 and filtering with the rest of the predicate. The aggregation function to be used is defined by θ , according to Table 3.2. In the example given by Figure 3.10, θ is a “greater than”. Thus, the aggregation function to use in the unnested AST is a *min* function. If no tuples come to the aggregation function (e.g., e_2 is empty), the aggregation variable must be assigned in the LetBind as $+\infty$ or $-\infty$, depending on θ .

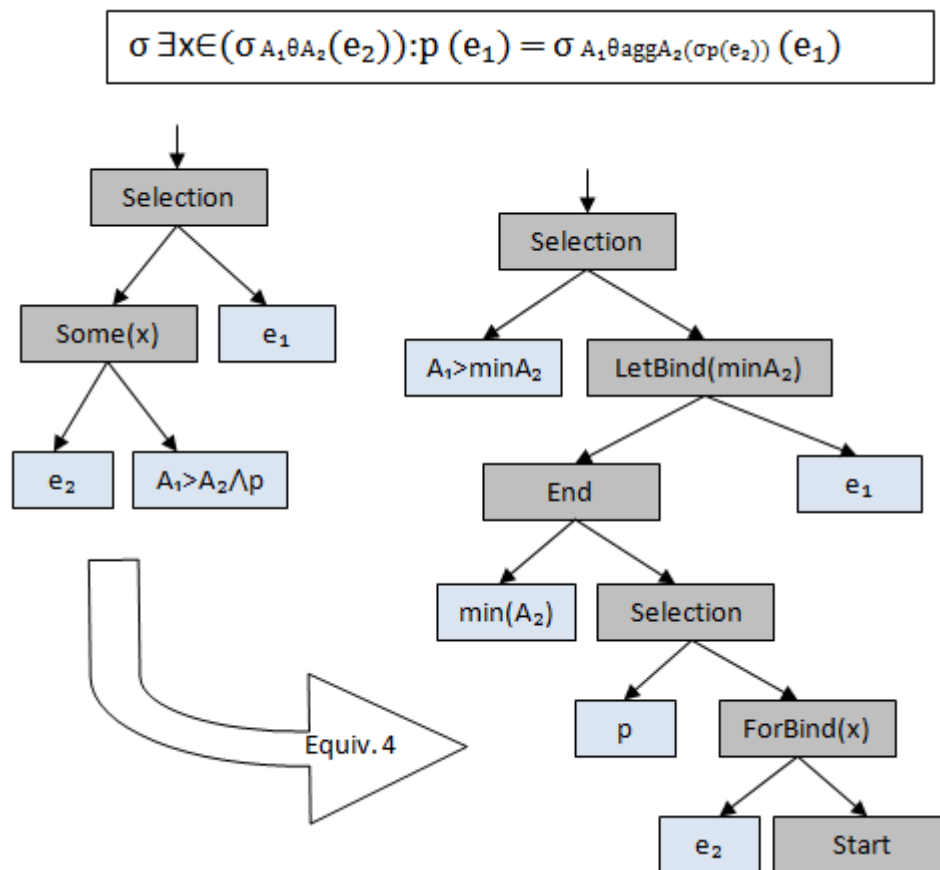


Figure 3.10: AST Rewriting using Equivalence 4, based on the comparison to the result of the aggregation function.

The last rewrite possibility is based on Equivalence 5, which uses Theta-join. As illustrated by Figure 3.11, it is added a Count operator (in the same way of the first two rewrites) that controls any change on variable ‘y’, bound inside e_1 . Then, it is made a Theta-join between the values coming from e_1 and the iteration over e_2 , where $\theta = p$. Finally, it is returned the selected values from e_1 , eliminating duplicates. As one may notice, it was presented two algebraic unnesting equivalences. The second one, which

the rewrite is based on, is a simplified version of the original equivalence, and was created here due to being easier to express with Bracket operators.

θ	agg	agg(\emptyset)
$>, \geq$	<i>min</i>	$+\infty$
$<, \leq$	<i>max</i>	$-\infty$

Table 3.2: Defining aggregation function to use and the aggregation value in empty case, according to θ .

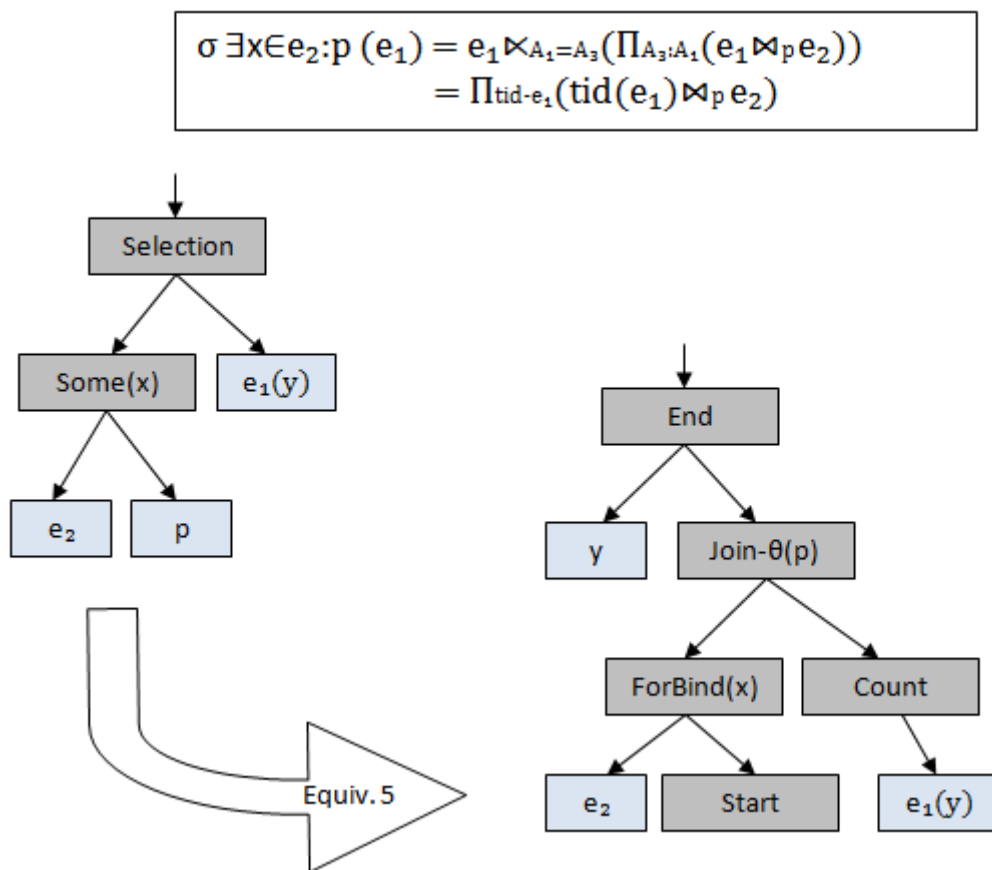


Figure 3.11: AST Rewriting using Equivalence 5, based on Theta-join.

3.4 Chapter Remarks

The purpose of this chapter was to explain how unnesting was developed inside Bracket's optimizer, whose importance was described in the introduction and – specifically for Bracket – in Chapter 2. At first, it was introduced the origin of the algebraic unnesting strategies, explaining how to bring queries to an algebraic level towards query unnesting through equivalence rules.

The second topic covered in this chapter was the advantages and issues of importing those unnesting strategies to Brackit, discussing which strategies would be applied and the reasons for applying them. Finally, it was presented the algebraic rewriter, which implements the unnesting strategies via rewrites on the AST generated from the query during the compilation process. It was detailed the four steps of this process, focusing the last one, where the algebraic equivalences are employed to apply the unnesting rewrites, whose performance improvements are analyzed in next chapter.

4 EXPERIMENTAL EVALUATION

The goal of this chapter is to evaluate the performance gains that the unnesting rewrites developed for the Brackit optimizer – presented on Chapter 3, based on application of algebraic equivalence rules – can provide to the evaluation of queries in Brackit. The trouble is that, as commented in the second topic of Chapter 3, the bottom-up compiler version of Brackit still has several points to be fixed in order to be evaluable. Bearing that in mind, the way found to check the evaluation improvement acquired by such optimization strategies was to simulate the unnesting rewrites using XQuery expressions presented in Chapter 2, applying them directly in queries and running them on an XQuery software.

4.1 Experiment Environment

The chosen XML database for XQuery processing evaluation was the eXistDB¹, a native open source XML database, with support to XQuery and cross-platform. This choice was made due to its popularity, large documentation, user-friendly IDE (eXide) and good fulfillment of our needs, including the possibility of using on any platform and satisfactory response times, without caring about transaction processing, since it has nothing to do with our tests. It was decided to use another engine instead of Brackit to execute our experiments because Brackit is currently not receiving support, then eventual trouble using it would require an extra unnecessary work since it would be done only for simulation purposes.

After some preliminary tests, it was noticed a certain irregularity on the response times given by the engine, even reducing to a minimal number of concurrent processes on the CPU, supposedly caused eventual I/O usage of those processes and indexing, beyond the unknown (non-documented) cache usage. It was also noticed a tendency of the first run of each query to be the slowest one, probably caused by caching. All the tests were made in an Intel® Core™ i7 CPU. The operational system is Windows 7 Professional, 64 bits.

4.2 Evaluated Queries

For this experimental evaluation, it has been used four XML documents as the database. Two of them are catalogs of chemistry publications² from 2014 and 2011.

¹: <http://exist-db.org/exist/apps/doc/documentation.xml>

²:<ftp://ftp.ncbi.nlm.nih.gov/pub/pmc/articles.A-B.tar.gz> and <ftp://ftp.ncbi.nlm.nih.gov/pub/pmc/articles.C-H.tar.gz>

The other two native from eXistDB: documents with full Shakespeare's plays (Hamlet, Romeo and Juliet). Figure 4.1 shows the data model of these documents. Figure 4.2 shows the data model of the catalogs (Query 2 uses catalogs with citation attributes named "mixed-citation" instead of "element-citation"). The database of publications is small, with hundreds of citation elements. The database of Shakespeare's scenes, speeches and lines is a little bigger, with more than a thousand of speeches.

```

<ACT>
  <TITLE>Title</TITLE>
  <SCENE>
    <TITLE>Title</TITLE>
    <STAGEDIR>Stagedir</STAGEDIR>
    <SPEECH>
      <SPEAKER>Speaker</SPEAKER>
      <LINE>Line</LINE>
      <LINE>...</LINE>
    </SPEECH>
    <SPEECH>
      ...
    </SPEECH>
  </SCENE>
  <SCENE>
    ...
  </SCENE>
</ACT>

```

Figure 4.1: Data model of Shakespeare's plays.

```

<element-citation>
  <person-group>
    <name>
      <surname>Surname</surname>
      <given-names>GN</given-names>
    </name>
    <name>
      ...
    </name>
  </person-group>
  <article-title>Article Title</article-title>
  <source>Source</source>
  <year>Year</year>
  <volume>Volume</volume>
  <fpage>FPage</fpage>
  <lpage>LPage</lpage>
  <pub-id>Pid</pub-id>
</element-citation>

```

Figure 4.2: Data model of publication catalogs.

To simulate the performance gain that would be obtained in Brackit with the unnesting strategies, it is presented below a set of queries with existential quantifier nesting, followed by an equivalent query without the nesting. The unnested queries were obtained using expressions in XQuery that simulate the equivalence rules presented in Chapter 3. For instance, to simulate the application of Equivalence 3 of the unnesting decision tree, it was used path and filter expressions, which corresponds to a Semi-join operation (as explained in Section 2.1). That was the case of Query 1 and Query 5. It was also possible to simulate Equivalence 4 because it is based on simple expressions and does not use any operator inexistent in XQuery. That was the case of Query 2 and Query 3. Query 4 simulates the application of Equivalence 5.

QUERY 1: NESTED

```
let $hamletSpeeches := doc('/db/apps/demo/data/hamlet.xml')//speech
for $i in doc('/db/apps/demo/data/r_and_j.xml')//SPEECH
where some $j in $hamletSpeeches satisfies $i/LINE = $j/LINE
return $i/LINE
```

QUERY 1: UNNESTED

```
let $hamletSpeeches := doc('/db/apps/demo/data/hamlet.xml')//SPEECH
for $i in doc('/db/apps/demo/data/r_and_j.xml')//SPEECH[LINE =
$hamletSpeeches/LINE]
return $i/LINE
```

QUERY 2: NESTED

```
let $mar2014 := doc('/db/apps/demo/data/nano_2014_mar.xml')//mixed-citation
let $feb2011 := doc('/db/apps/demo/data/nano_2011_feb.xml')//mixed-citation
for $i in $mar2014
  where some $j in $feb2011 satisfies
    $i/year lt $j/year and $j/volume > 10
return $i
```

QUERY 2: UNNESTED

```
let $mar2014 := doc('/db/apps/demo/data/nano_2014_mar.xml')//mixed-citation
let $feb2011 := doc('/db/apps/demo/data/nano_2011_feb.xml')//mixed-citation
let $febYear :=
  for $j in $feb2011
  where $j/volume > 10
  return $j/year
for $i in $mar2014
  where $i/year < max($febYear)
return $i
```

QUERY 3: NESTED

```
let $hamletSpeeches := doc('/db/apps/demo/data/hamlet.xml')//SPEECH
for $i in doc('/db/apps/demo/data/r_and_j.xml')//SPEECH
where some $j in $hamletSpeeches satisfies count($i/LINE) gt count($j/LINE)
return $i/LINE
```

QUERY 3: UNNESTED

```
let $hamletSpeeches := doc('/db/apps/demo/data/hamlet.xml')//SPEECH
```

```

let $countHamletSpeeches :=
  for $i in $hamletSpeeches
  return count($i/LINE)
for $i in doc('/db/apps/demo/data/r_and_j.xml')//SPEECH
where count($i/LINE) gt min($countHamletSpeeches)
return $i/LINE

```

QUERY 4: NESTED

```

let $hamletSpeeches := doc('/db/apps/demo/data/hamlet.xml')//SPEECH
for $i in doc('/db/apps/demo/data/r_and_j.xml')//SPEECH
where some $j in $hamletSpeeches satisfies contains($i/LINE, $j/LINE)
return $i/LINE

```

QUERY 4: UNNESTED

```

let $hamletSpeeches := doc('/db/apps/demo/data/hamlet.xml')//SPEECH
for $i in doc('/db/apps/demo/data/r_and_j.xml')//SPEECH[contains(LINE,
$hamletSpeeches/LINE)]
return $i/LINE

```

QUERY 5: NESTED

```

let $aug2014 := doc('/db/apps/demo/data/drugs_2014_aug.xml')//element-citation
let $feb2014 := doc('/db/apps/demo/data/drugs_2014_feb.xml')//element-citation
for $i in $feb2014
let $ipns := $i/person-group/name/surname
let $iy := $i/year
  where some $j in $aug2014 satisfies
    let $jpsn := $j/person-group/name/surname
    let $jy := $j/year
    return $ipns = $jpsn and $iy eq $jy
return $i

```

QUERY 5: UNNESTED

```

let $aug2014 := doc('/db/apps/demo/data/drugs_2014_aug.xml')//element-citation
let $feb2014 := doc('/db/apps/demo/data/drugs_2014_feb.xml')//element-citation
let $aug2014psn := $aug2014/person-group/name/surname
let $feb2014_2 := $feb2014[person-group/name/surname = $aug2014psn]
for $i in $feb2014_2
  let $iy := $i/year
  let $ipns := $i/person-group/name/surname
  for $j in $aug2014
    let $jy := $j/year
    let $jpsn := $j/person-group/name/surname
    where $iy eq $jy and $ipns = $jpsn
  return $i

```

4.3 Experiments Results

The focus of our experiments is to provide a notion of how improved would be the performance of query evaluation on Brackit if its bottom-up version was able to be

used, so that the query would pass by the optimizer’s algebraic rewrite steps. The performance improvement reflection obtained with these experiments might be considered minimalist, because database operations, such as Semi-join and Join, when implemented physically, can be implemented with optimized algorithms, such as bloom filters (bloom filter is a structure that uses hash functions to reduce the amount of transferred data).

Given the low regularity of the running time of queries in using eXide, it has been performed, for each query, 5 runs. In most cases, the first run was the lowest one, increasing the average running time. We considered relevant to keep this value in the measurement to reflect a realistic scenario. In all queries, it is evaluated the performance gain of unnesting existential quantifiers, which is the focus of our initial version of optimization using the algebraic rewriter.

The first query is a search for lines from speeches in “Romeo and Juliet” that contains a line equal to a line from Hamlet’s speeches. In the nested version of the query, the selection predicate enables unnesting with Equivalence 3 from the decision tree of existential quantifiers. The nested version of this query has run in an average time of 189.2128s while running the unnested version, the average response time was 149.8846s, as shown in Figure 4.3. The query unnesting provided a reduction of 20% on the response time.

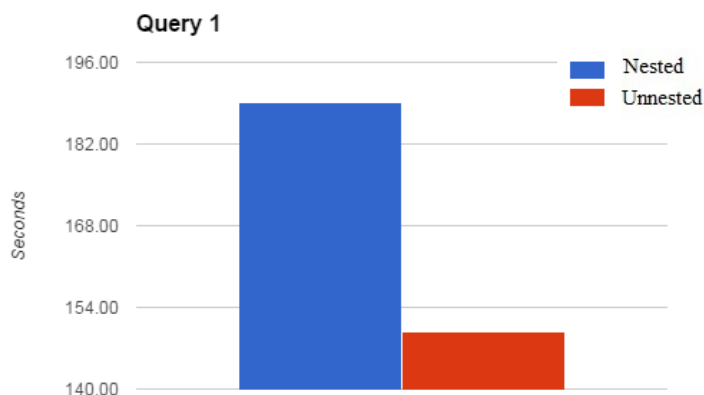


Figure 4.3: Response time comparison of the two versions of Query 1.

The second query returns citations from one catalog that are older (according to the year) than citations whose volume is higher than 10 from the other catalog. In the nested version of the query, the selection predicate enables unnesting with Equivalence 4 from the decision tree of existential quantifiers. The nested version of this query has run in an average time of 0.5868s while running the unnested version, the average response time was 0.1146s, as shown in Figure 4.4. The query unnesting provided a reduction of 80% on the response time.

The third query gets the lines from speeches of “Romeo and Juliet” whose speech is longer (in terms of number of lines) than Hamlet’s speeches. In the nested version of the query, the selection predicate enables unnesting with Equivalence 4 from the decision tree of existential quantifiers. The nested version of this query has run in an average time of 19.704s while running the unnested version, the average response time was 0.8652s, as shown in Figure 4.5. The query unnesting provided a reduction of 96% on the response time.

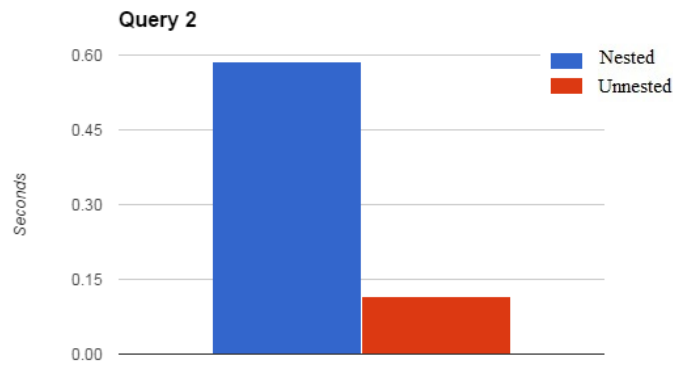


Figure 4.4: Response time comparison of the two versions of Query 2.

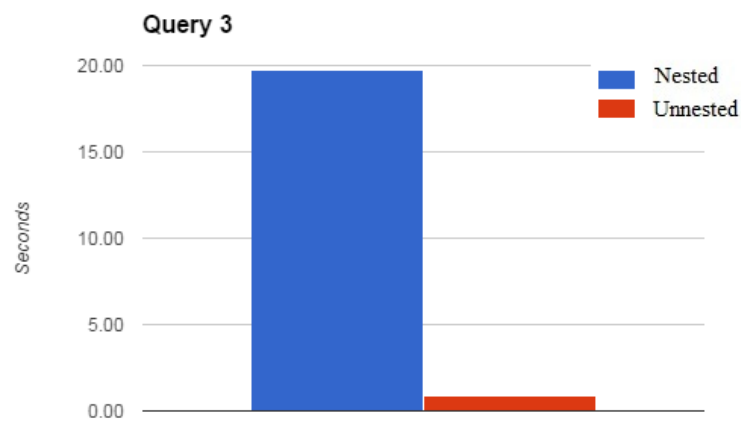


Figure 4.5: Response time comparison of the two versions of Query 3.

The fourth query gets the lines from speeches of “Romeo and Juliet” whose speech contains any Hamlet’s line. In the nested version of the query, the selection predicate enables unnesting with Equivalence 5 from the decision tree of existential quantifiers. The nested version of this query has run in an average time of 171.563s while running the unnested version, the average response time was 134.0466s, as shown in Figure 4.6. The query unnesting provided a reduction of 23% on the response time.

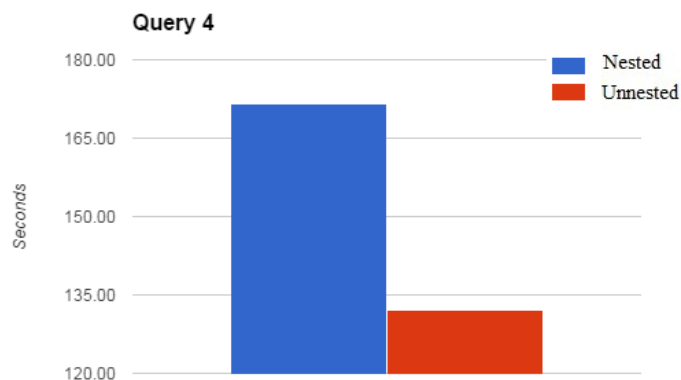


Figure 4.6: Response time comparison of the two versions of Query 4.

Query 5 searches the citations from one catalog having the same surname and year of any citation in the second catalog. In the nested version of the query, the selection predicate enables unnesting with Equivalence 3 from the decision tree of existential quantifiers, doing initially a Semi-join on publications' surnames and reducing the amount of data. The nested version of this query has run in an average time of 8.6838s while running the unnested version, the average response time was 4.0992s, as shown in Figure 4.7. The query unnesting provided a reduction of 51% on the response time.

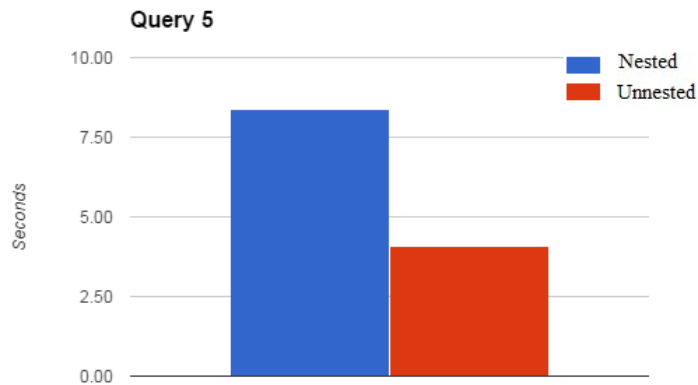


Figure 4.7: Response time comparison of the two versions of Query 5.

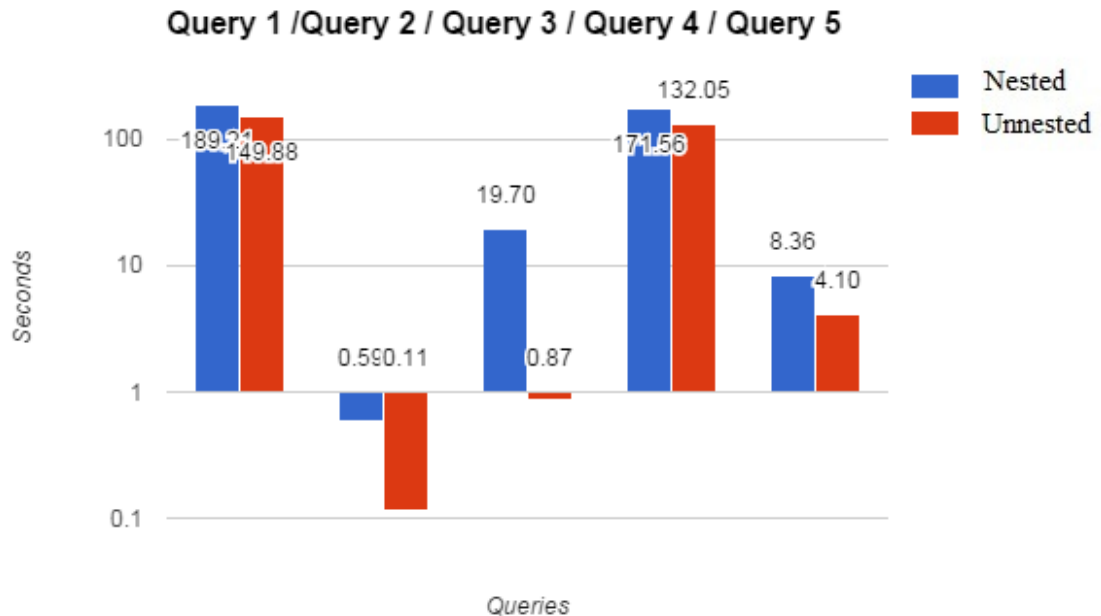


Figure 4.8: Nested and unnested queries' running time comparison.

4.4 Results Analysis

Figure 4.8 presents a comparison of running times between the nested and unnested query versions, measured in seconds. To make visible the evaluation result of the second query in the chart, it is used a logarithmic scale. The percentage performance

gain is illustrated by Figure 4.9. It is possible to note that the best results were obtained with Equivalence 4. This result was actually a bit obvious since using existential quantifiers on that kind of queries is absolutely not intelligent. The reason of that big difference is that using the Equivalence 4, the unnested query gets a lower level of complexity (it becomes $O(n)$). The other queries, optimized using Semi-joins and Joins, obtained smaller, but significant, performance gains. It is important to remember that, as commented previously, operators can be implemented to access data physically with different optimized algorithms, which would result in even shorter evaluation times. Furthermore, we must take into account that the experimental databases are small, and bigger ones might provide stronger differences between the nested and unnested versions. And last, but no least, it is important to ponder that we are using a centralized database, and unnesting strategies are known to present even better results when applied in a distributed environment.

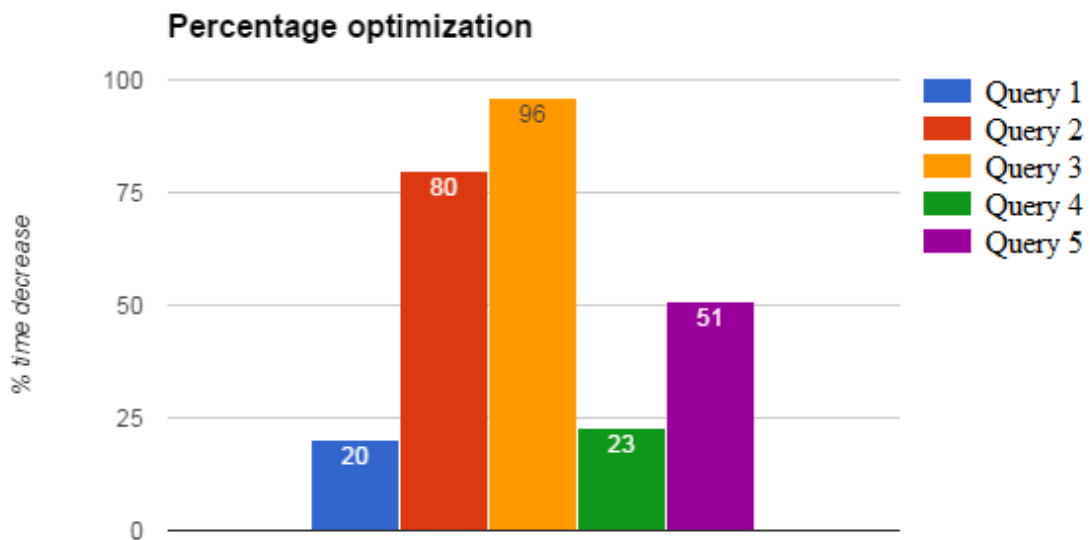


Figure 4.9: Percentage performance gains.

All those facts enable us to consider the unnesting strategy using algebraic equivalence rules not only applicable in Brackit's optimization pipeline, but also an important method to be added to Brackit to decrease the evaluation time of queries. Although it was only analyzed the existential quantifier case, the experimental results are enough to highlight the relevance of the algebraic rewriter as a part of the Brackit optimizer restructuring, encouraging the completion of these ideas to this open-source XQuery engine.

5 CONCLUSION

Despite the unstoppable increase of processing power technologically available, performance care is, and will still be, necessary, given that databases and real-time computing also grow. After studying some strategies to perform unnesting – which is known to be an important method to optimize queries on databases –, including one specific for XML databases based on algebra, we decided to adapt some of these ideas to Brackit, our XQuery engine. In Brackit, most part of query compilation is based on manipulation of trees that represents in an abstract form (ASTs). Implementing those strategies on Brackit consists in an important step to restructure Brackit’s optimizer because this brings the optimization development to a higher level, decreasing the AST rewriting complexity and heavy code legacy of the currently stable version.

This work presented how we adapted an unnesting strategy based on algebraic equivalence rules to Brackit through AST rewrites, inserting operators that manipulate data physically and modifying the structure of the tree. This strategy includes several steps to prepare the AST to be manipulated, to verify the nesting particulars, and to define which equivalence rule must be applied to rewrite the AST. As a first work, we focused on the implementation of unnesting existential quantifier expressions.

The results obtained running the experimental queries, where we used native operators of XQuery to simulate the equivalence applied on Brackit, confirmed that even queries on small and centralized databases can be significantly improved removing naïve nestings. Moreover, the performance gain provided by unnesting would be higher than the simulated results if it were used specific operators to perform joins and semi-joins because they can be implemented to access with optimized algorithms.

The experimental results motivate us to resume the development in Brackit and to continue this work on query unnesting. The work for the future includes implementing the remaining physical operators necessary to apply the algebraic equivalences and also the remaining nesting cases: universal quantifiers and implicit grouping.

REFERENCES

- [Kim 1982]KIM, W. On Optimizing an SQL-like Nested Query. *Trans. on Database Systems, Vol 9, No. 3*, 1982.
- [Dayal 1987]DAYAL, U. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. *Proc. VLDB Conf., pp.197-202*, 1987.
- [Ganski 1987]GANSKI, R. A.; LONG, H. K. T., Optimization of nested SQL Queries Revisited. *Proc. SIGMOD Conf., pp. 23-33*, 1987.
- [Muralikrishna 1992]MURALIKRISHNA, M. Improved unnesting algorithms for join aggregate sql queries. *Proceedings of the 18th International Conference on Very Large Data Bases, pp. 91- 102*, 1992.
- [Katz et al. 2003]KATZ, H. et al. XQuery from the Experts: A Guide to the W3C XML Query Language. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321180607.
- [May et al. 2006]MAY, N.; HELMER S.; MOERKOTTE G. Strategies for Query Unnesting in XML databases. *ACM Transactions on Database Systems, Vol. 31, No. 3*, 2006.
- [Sauer e Bächle 2011]SAUER, C.; BÄCHLE, S. Unleashing xquery for data-independent programming. 2011
- [W3C 1998]W3C. XML Query Language (XQL). 1998. Available from Internet: <<http://www.w3.org/TandS/QL/QL98/pp/xql.html>>.
- [Robie et al. 2000] ROBIE, J.; CHAMBERLIN, D.; FLORESCU, D. Quilt: an XML Query Language. <http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html>.
- [W3C 2010a]W3C. XQuery 1.0: An XML Query Language (Second Edition). 2010. Available from Internet: <<http://www.w3.org/TR/xquery/>>.
- [W3C 2004]W3C. XML Information Set (Second Edition). 2004. Available from Internet: <<http://www.w3.org/TR/2004/REC-xml-infoset-20040204/>>.
- [W3C 2006]W3C. Extensible Markup Language (XML) 1.1 (Second Edition). 2006. Available from Internet: <<http://www.w3.org/TR/xml11/>>.
- [W3C 2010b]W3C. XQuery 3.0: An XML Query Language. 2010. Available from Internet: <<http://www.w3.org/TR/xquery-30/>>.
- [Bächle 2013]BÄCHLE, S. Separating Key Concerns in Query Processing—Set Orientation, Physical Data Independence, and Parallelism. 2013.

[Graefe 1994]GRAEFE, G. Volcano: An Extensible and Parallel Query Evaluation System.
IEEE Trans. on Knowl. and Data Eng., 6(1):120–135, 1994.