FEDERAL UNIVERSITY OF RIO GRANDE DO SUL
INFORMATICS INSTITUTE
BACHELOR OF COMPUTER SCIENCE

THOMAS DA SILVA RODRIGUES

Executing SQL Queries with an XQuery Engine

Graduation Thesis

M. Sc. Caetano Sauer
Advisor

Prof. Dr. Renata Galante
Coadvisor

Porto Alegre, July 2012

CIP – CATALOGING-IN-PUBLICATION

Rodrigues, Thomas da Silva

Executing SQL Queries with
an XQuery Engine / Thomas da Silva Rodrigues. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2012.

48 f.: il.

Graduation Thesis – Federal University of Rio Grande do Sul. BACHELOR OF COMPUTER SCIENCE, Porto Alegre, BR–RS, 2012. Advisor: Caetano Sauer; Coadvisor: Renata Galante.

1. XQuery. 2. SQL. 3. XML. 4. Database. 5. Storage. 6. Parser. 7. Relational. 8. Translation. I. Sauer, Caetano. II. Galante, Renata. III. Título.

“Education is the most powerful weapon which you can use to change the world.”

— NELSON MANDELA

ACKNOWLEDGEMENTS

I would like to thank my family, they are the base of my life, they educated me and are directly responsible for this achievement. Thank you father for have invested hard in my education. Thank you mother for always help me and support me with love, it includes my aunt Patrícia, who is my second mother. Thank you grandparents for everything you did and keep doing for me, you are very special for me.

Thank you very much, Federal University of Rio Grande do Sul (UFRGS), specially Informatics Institute, for the excellence with which you conduct everything in the educational process. It is very hard to achieve this level of excellence, being completely free of charge. Also thank you Technical University of Kaiserslautern, for the opportunity given to international students.

I would like to thank Prof. Dr. Theo Härder for giving me the outstanding opportunity of studying in his excellent research group, in Germany. It has provided me an amazing experience both professionally and personally. Special thanks to Caetano Sauer, my advisor, who had lots of patience and dedication to help me in every detail of this work, being not only an advisor, but a friend. Thank you, Brackit group, for all the support in the way. Thank you, Renata Galante, for the important contributions and the helpfulness in this work since I arrived back in Brazil.

Finally, I could not forget to thank an icon, thank you Pablo. I also want to thank all my friends in these last 6 years, with who I spent enjoyable moments. Your company have been very important to help me go through the hard periods of this journey.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

DB	Database
SQL	Structured Query Language
XML	Extensible Markup Language
DBMS	Database Management System
XDBMS	XML Database Management System
XDM	XQuery Data Model
FLOWR	For Let Order by Where Return
AST	Abstract Syntax Tree
CSV	Comma-Separated Values
ANTLR	Another Tool for Language Recognition
HSQLDB	Hyper SQL DB
DOM	Document Object Model

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In the last years, we witnessed the rise of XQuery, a language natively designed to query over XML documents. The Internet, as we know today, makes a massive use of XML documents, but for data exchange, not as a storage technology. Relational databases, in the other hand, are dominant in the industry, for storage. To provide an integration between these two worlds, our work presents a tool capable of compile SQL queries into XQuery operators ready to be processed by an XQuery engine. In this way, we intend to give more flexibility to developers, offering language independence.

Keywords: XQuery, SQL, XML, database, storage, parser, relational, translation.

RESUMO

Nos últimos anos, nós presenciamos o crescimento da XQuery, uma linguagem nativamente projetada para consultas em documentos XML. A Internet, como conhecemos atualmente, faz um uso massivo de documentos XML, mas para intercâmbio de dados, não como uma tecnologia de armazenamento. Bancos de dados relacionais, por outro lado, são dominantes na indústria, para armazenamento. Para prover uma integração entre esses dois mundos, nosso trabalho apresenta uma ferramenta capaz de compilar consultas SQL em operadores XQuery, prontos para serem processados por um motor XQuery. Dessa forma, nós pretendemos dar mais flexibilidade aos programadores, oferecendo independência de linguagem.

Palavras-chave: XQuery, SQL, XML, database, storage, parser, relational, translation.

1 INTRODUCTION

1.1 Overview

Section 1.2 gives a quick introduction to the subject of the work. Section 1.3 talks about the context, motivation and proposed goals of this work. Section 1.4 presents some works related to the present graduation thesis, and section 1.5 depicts the structure followed by the work.

1.2 Basis

XQuery has become known in the last years for being a language with the purpose of querying XML documents. The XML format became widely spread in the last decade and it is nowadays the standard format for electronic data interchange. Furthermore, it is a standard that was born to drive content presentation on the web, hence being of utmost importance in the modern web-based applications. However, XQuery is a powerful data programming language, whose functionality goes beyond querying and manipulating XML data. XQuery provides mechanisms to query these XML-based documents and to construct them as well. It is basically a functional programming language, with features like absence of side-effects, fully-composable expressions, variable bindings, recursion, and higher-order functions. For querying, XQuery offers FLOWR expressions, a construct influenced by the SQL syntax, capable of performing the standard SPJ operations—selection, projection, and join. Therefore, we can conclude that XQuery actually subsumes basic SQL functionality.

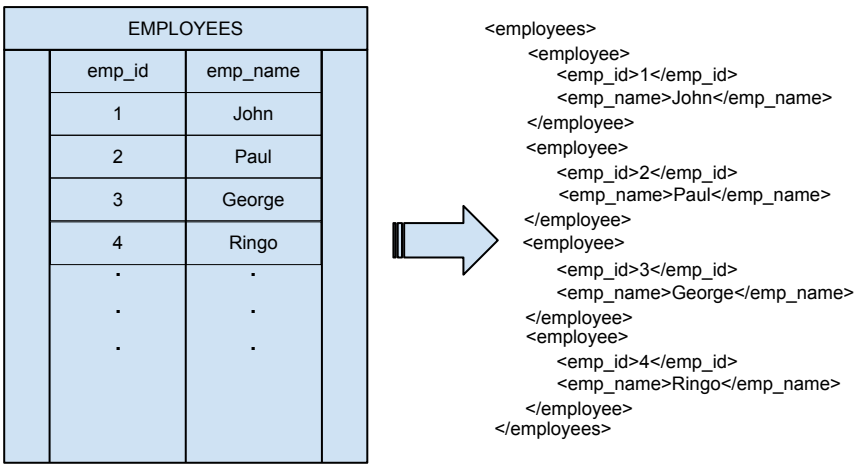

Figure 1.1: SQL Table Mapped to a Flat XML File

The XQuery language incorporates the XML data model, in which data is modeled as XML fragments. Unlike other data models, XML does not need to conform to a schema, allowing a varied degree of structuredness, from well defined tabular data to deeply nested hierarchies of arbitrary data items. Thus, this hierarchical data representation offers both flexibility and extensibility, allowing us to represent a large range of data formats, such as tables, documents, graphs, images, music notation, among others. For instance, Figure 1.1 shows how a relational table can be mapped to an XML document. In this example, standard SQL operations on the table at the left side can be mapped to FLOWR expressions on the document at the right side.

1.3 Motivation and goals

Despite all its advantages and widespread use on the web, XML is not heavily adopted as storage technology, an area where relational databases and SQL still dominate. Native XML databases (XDBMS) are well known and abundant as scientific prototypes, but remain exotic in enterprise applications. A possible reason for this is that the high flexibility of the XML data model adds an additional layer of complexity to well-known database techniques, such as optimization, isolation, recovery, storage, indexes, and so on. Hence, an XDBMS product from major database vendors was not yet established on the market.

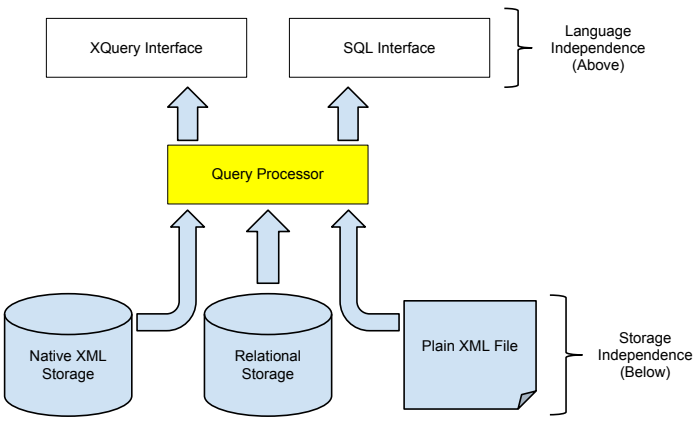

Figure 1.2: System Architecture

Given these aspects, we can observe the lack of a mechanism that integrates both XQuery and SQL, allowing syntax independence in addition to the data storage independence. The predominance of XML as a data-interchange format and the dominance of SQL as querying language, require a tool that allows us to handle both, by hiding expensive conversions done internally. Most efforts in this direction use an SQL query engine adapted to deal with XQuery, which involves encode the hierarchical model of XML in tables. Since XQuery is a more powerful language than SQL, it is meaningful to use an XQuery processor as base to create this mechanism. Aiming at more data manageability, we present a framework capable of interpreting SQL using an XQuery engine. The SQL processing infrastructure is migrated to XQuery, adding a parser to interpret the SQL language, providing language independence in the layer "above" the processor. This eliminates the need to rewrite code to XQuery. Furthermore, it allows the SQL developer to keep working with the system without needing to learn a new query language.

The framework is able to map foreign data models to XML, allowing the XQuery engine to process not only XML data, but also relational tables. This provides storage

independence to the layer "below" the query processor. This scenario is depicted in Figure 1.2, which illustrates the architecture of our system. Here, a single query processor is able to provide both XQuery and SQL interfaces, while seamlessly managing, at the storage layer, XML stores together with relational data. A conversion is not necessary from relational tuples to XML data. To develop our framework we use the Brackit[1] engine as base XQuery processor.

1.4 Related work

The major efforts in this SQL and XQuery integration area have simply provided data mapping and techniques to import (export) XML data into (from) relational databases. The SQL/XML standard (Eisenberg e Melton 2004), for example, is an extension of SQL that allows SQL queries to create XML structures with a few XML publishing functions. Some well difunded relational database engines, such as PostgreSQL (PostgreSQL 2012), implement this standard. Another tool that uses an relational engine as base to query XML is the SQL Server from Microsoft (Garcia 2007). It presents an extension to deal with XQuery in its engine, making possible to compose SQL and XQuery in the same query.

Differently from the previously mentioned technologies, some approaches use a native XQuery engine in the process. AquaLogic Data Services Platform (Jigyasu et al. 2006) translates SQL queries into XQuery expressions to be processed by an XQuery engine. Unlike our approach, the AquaLogic solution needs to reprocess the XQuery query after the translation, instead of generating XQuery operators directly from the SQL query. Another relevant related work is the IBM project named ROX (Halverson et al. 2004), it focuses on querying only over native XML documents and compare performance results. However, the published text does not gives much explanation about the actual translation process.

1.5 Work structure

This work is organized as follows. Chapter 2 describes the Brackit engine (Sauer e Bächle 2011), the data-independent XQuery compiler which served as base for our work. Chapter 3 presents the mechanism we used to map relational tuples into XML nodes, which corresponds to the layer below the processor. In Chapter 4, we show the implementation of the rules to translate SQL to XQuery, which allow the query engine to recognize both languages, which corresponds to the layer above the processor. Chapter 5 presents some empirical results. Finally, Chapter 6 draws our conclusive remarks and future work.

[1] http://www.brackit.org/

2 THE BRACKIT XQUERY ENGINE

2.1 Overview

This chapter seeks to provide the basic concepts of this work. Section 2.2 explains the fundamentals of XQuery and its basic mechanism to query data, the FLOWR expressions. Section 2.3 shows how Brackit processes the FLOWR expressions, which will be widely used in the translation of SQL queries.

2.2 XQuery semantics and FLOWR expressions

XQuery is a powerful functional programming and querying language, created to manipulate XML-based data (Katz et al. 2003). It is based on the XPath language (Query e Groups 2010), that was designed for addressing parts of an XML document, using a compact navigation syntax. XQuery offers a large variety of expressions, such as primary, sequence, path —through XPath—, comparison, arithmetic, among other expressions as well as recursive functions. All of them have no side-effects, which means that there are no assignments, and can be generally composed. When it comes to the querying capabilities, some authors evaluate the expressiveness of XQuery as comparable to the "relational completeness" criterion (Codd 1972), which means that XQuery achieves the basic relational expressiveness. XQuery is defined as a transformation on the XQuery Data Model (XDM) (W3C 2010).

According to XQuery, each expression evaluates to a sequence, which is composed by zero or more items that can be an atomic value or a node. An atomic value has a simple type —defined in the XML Schema standard (W3C 2004)—, which includes the special "untyped" type, whereas a node is one of the seven kinds of node defined by XPath standard (Query e Groups 2010): document, element, attribute, text, comment, processing instruction, and namespace. Sequences containing a single item (singletons) are defined to have the same semantics as the item alone. The integer value "4", for instance, represents the same entity as the sequence "(4)". In XDM, documents are abstracted as nodes of type "document". This data abstraction allow us to work with XML and a large range of others file formats as long as mapping mechanisms are provided. Figure 2.1 shows the summarized XDM structure.

An important aspect in the XQuery semantics is the expression context, which contains all the information that can affect an expression in some way, e.g., variable bindings, type information, available functions, etc. The expression context can be divided into static and dynamic, where the former is used in the compilation phase and the latter is used in the execution phase. The dynamic context is essential for the XQuery FLOWR expressions, and it consists of the values of the variables. When a expression is evaluated,

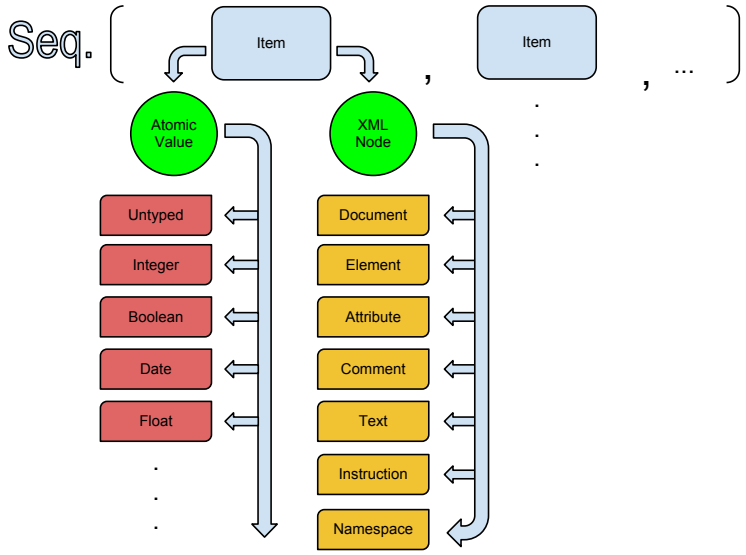

Figure 2.1: XDM Sequence-Based Structure

its dynamic context is a tuple which contains the values of each variable. In a loop, the dynamic context changes at each iteration, by walking over the tuple stream.

The basic tool for querying data in XQuery is the FLOWR expression (Katz et al. 2003), that is very similar to the well-known relational clauses. FLOWR is a short for the basic XQuery querying clauses: *for*, *let*, *order by*, *where*, *return*, and, additionally the *group by* clause, which was introduced in later revisions of the standard. The basis of a FLOWR expression is the *for* and *let* clauses, because they initiate the dynamic context, by binding the first variable. Therefore, a FLOWR expression needs at least one *for* clause or *let* clause, to generate a combined sequence of bound variables, called tuple stream, which will be the input to be modified by the other optional clauses, such as *where*, *order by*, *group by*, or even by others *for* and *let* clauses. Finally, the *return* clause processes it, to produce an outcome based on the dynamic context.

The *for* clause is responsible for iterating over a data sequence and binding a unique variable to each value of the sequence, creating a tuple stream, that can be referenced by the bound variable. Nested *for* clauses generate combined tuples, where each one of the tuples from the lower *for* clause is associated to each value from the higher *for* clause, generating a combined tuple stream (as in a Cartesian product).

The second letter from the abbreviation represents the *let* clause, that binds an entire expression result to a variable, producing one single tuple, unlike the *for* clause, which produces one tuple for each element of the sequence. The tuple generated by the *let* clause is concatenated to the tuples generated by the previous *for* clause, if they exist.

The *where* clause filters the tuples according to a predicate, an expression that can be evaluated to true or false — eliminating the tuples that do not satisfy the specified condition. The *order by* clause sorts the tuples by some key, which results from the evaluation of a given expression into a single atomic value. These keys are contained in *order-specs*, each order-spec can define sort aspects like ascending or descending, how empty evaluated sequences will be sorted, etc. We also have the *group by* clause, which groups the tuples by a key, that must be a variable or a set of variables, merging the values of the others (non-key) variables.

Finally, the *return* clause generates the query outcome, it is comparable to the SQL select, where we can specify expressions to produce the desired result. However, the re-

```
for      $e in collection(''emps'')
where    $e/emp_salary > 10000
let      $department := $e/department
group by $department
let      $salary := avg($e/emp_salary)
order by $department
return   <record>
           {$department,
            $salary}
         </record>
```

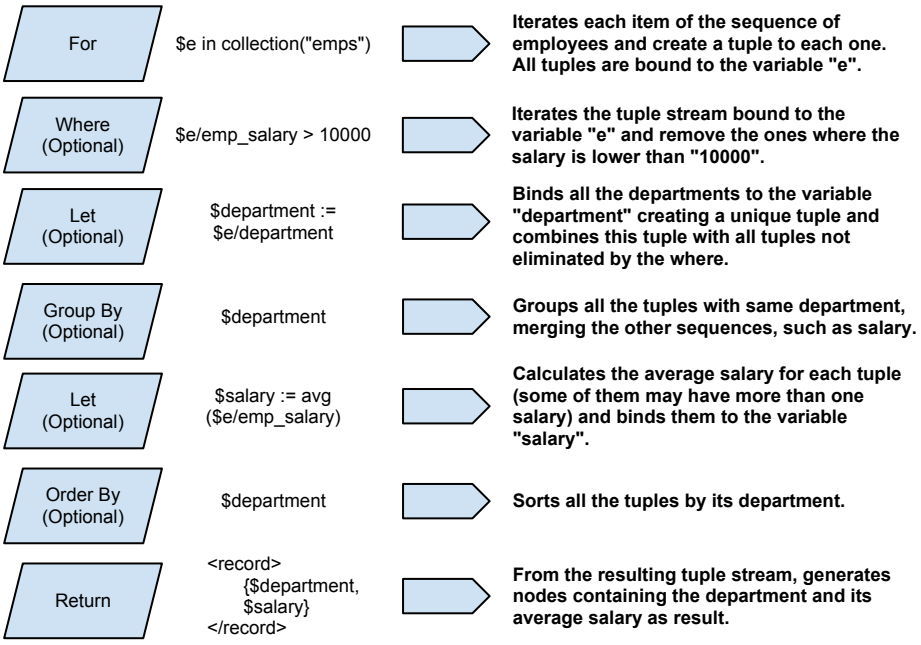

Figure 2.2: FLOWR Expression

turn clause is more flexible, offering dynamic node construction expressions, which can be arbitrary nested and deal with untyped values. The outcome is a result sequence of the whole FLOWR expression, where is generated one value for each tuple in the stream, flattening the nested sequences. Figure 2.2 presents a query with a simple FLOWR expression and its clauses. This query iterates over a collection of employees whose salary is greater than 10000 and groups them by their departments, returning a document containing the departments and their average salaries.

2.3 Brackit pipelines

The Brackit engine processes the FLOWR expressions in a simple and clear way, executing them as close as possible to the semantical description (W3C 2010). Every FLOWR expression consists of a sequence of clauses, which are evaluated to a sequence of items, as defined in the XDM. In practical terms, Brackit creates a pipeline from the FLOWR semantical tree, where a tuple stream is passed step by step through this pipeline.

The FLOWR clauses are represented in the pipeline by relational-style operators (Sauer e Bächle 2011). A especial operator named *Start* begins the pipeline by passing an empty context (single tuple) to the next operator which can be a *ForBind* or a *LetBind*,

that represent, respectively, the *for* clause and *let* clause. These two operators generate the tuples that are going to be sorted, filtered and grouped by the following operators. The *where* clause is performed by the *Select* operator, which filters the tuple stream by removing the tuples that do not satisfy the attached boolean expression.

The *OrderBy* operator sorts the tuple stream received by the previous operator, according to an expression. Is important to remember that *order by* expressions have to be evaluated to a singleton, otherwise it is not viable to sort the tuples, because is not possible to define which value is going to be compared. Unlike the others, the *GroupBy* operator works different in SQL and XQuery, where concepts of grouping and aggregation are separated. As we can see in the Figure 2.3, the *group by* just merges the tuples containing the same key into a single one, with no need for aggregation in the merged values. At the end, the pipeline is connected with the *return* clause by a special *pipe* operator.

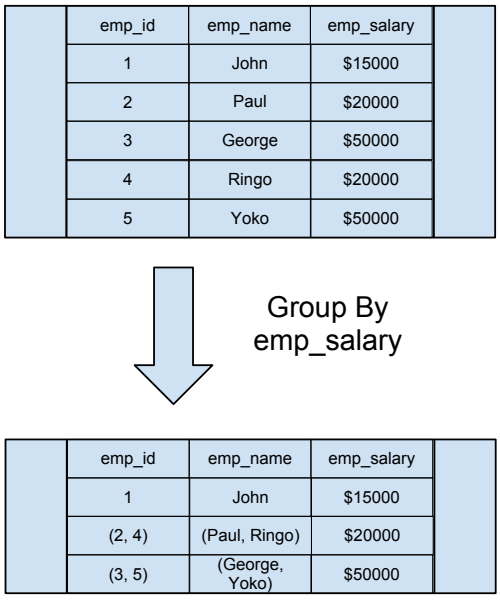

Figure 2.3: XQuery Group By Mechanism

The Brackit compilation process includes further optimizations, which are not relevant for this work, such as lifting, rewriting, and joins. Nested *ForBinds* followed by a *Select* operator are replaced by a *Join* operation, avoiding unnecessary large data amounts caused by Cartesian products. Nested FLOWR operations also can be undone, by using the Brackit *pipe lifting* mechanism. The pipelines are compiled from the *Start* to the *pipe* operator in a bottom-up process. The Brackit operators, as described above, work in a relational way, handling tuple streams in a way similar to how relational operators manipulate tables.

3 MAPPING RELATIONAL MODEL TO XML

3.1 Overview

This chapter concerns about the layer "below" the query processor, introduced in section 1.3 and demonstrated in Figure 1.2. Here we explain the process of mapping relational tuples to an XML abstraction, making it readable to the query processor. Section 3.2 gives the fundaments used in sections 3.3 and 3.4 to explain the mapping of relational tables. Section 3.5 shows how schemas are composed to be used by the SQL interpreter.

3.2 Node abstraction

The data handled by the Brackit engine is modeled by the XDM. External data sources are accessed by retrieving nodes from documents and collections thereof, by using the functions "doc()" and "collection()". The interaction between processor and nodes is given by an interface that provides a range of navigation primitives. This mechanism is very similar to the DOM API (Wood 1999), where functions that allow the navigation between nodes, such as "getChildren()", "getFirstChild()", and comparisons, like "isRoot()", "isParentOf()", "isSibling()", etc, are given. This interface implementation will better detailed in the next subsection.

The process of retrieving and mapping data to feed the processor can be divided in two different parts. The first is responsible to make the binary translation, deserializing tuples from container files. The second makes the logical abstraction, by interpreting the tuples read in the first step and creating the nodes, which can be understood by the query processor. Since these both steps are completely independent from each other, with a well-defined interface between them, the second step is not concerned with the file formats deserialized in the first one. Therefore, the deserialization can support a large range of file formats, offering a series of specific deserializers.

The node interface provides an object abstraction, thus allowing the processor to work with several kinds of nodes, such as memory nodes (pointers), DB Nodes (XDBMS), among others. It provides the storage independence to the engine. As we are going to see in the next section, this mechanism permits to represent relational tuples coming from SQL tables by using the node abstraction.

3.3 Mapping tuples to nodes

An essential part of our work is to map the relational model to an XML structure, linking it to the Brackit engine. This subsection details the logical translation from tuples, that is the second part of the data mapping. Serialized files are read into tuples that are

mapped to nodes, which can be understood by the Brackit engine. This mechanism allow us to logically map a variety of different data models to the XDM. The structure used in this process is the *RowNode*, which is a class that wraps a record containing atomic values. As we saw previously, the first step, made by the scanners, deserializes container files, creating an object that represents an array of atomic values. These objects are used to construct instances of a *RowNode*, that simulates an XML node.

The *RowNode* structure provides navigation primitives to access its fields. Despite it simulates an XML node, which is a tree, internally the accesses are performed as array accesses, performing a similar efficiency than a native relational engine. Is important to note that this infrastructure simulates the XML behavior without need to actually physically convert the data. With this approach, is possible to achieve a good performance while reading data from heterogeneous sources and formats.

3.4 Mapping tables

The deserializers produce arrays of atomic values (tuples), from the container files, which can be interpreted as simple rows. In our approach, mapping from relational data model, the association is trivial: the deserializer generates one tuple for each relational table row. Each resulting tuple is converted, thus, in a *RowNode*. At the end, a relational table is logically mapped to a set of *RowNodes*. Since we are dealing with a database engine, the process of mapping an entire table is just logical; to read a relational table, a *RowNode* stream is created and each row is read at a time.

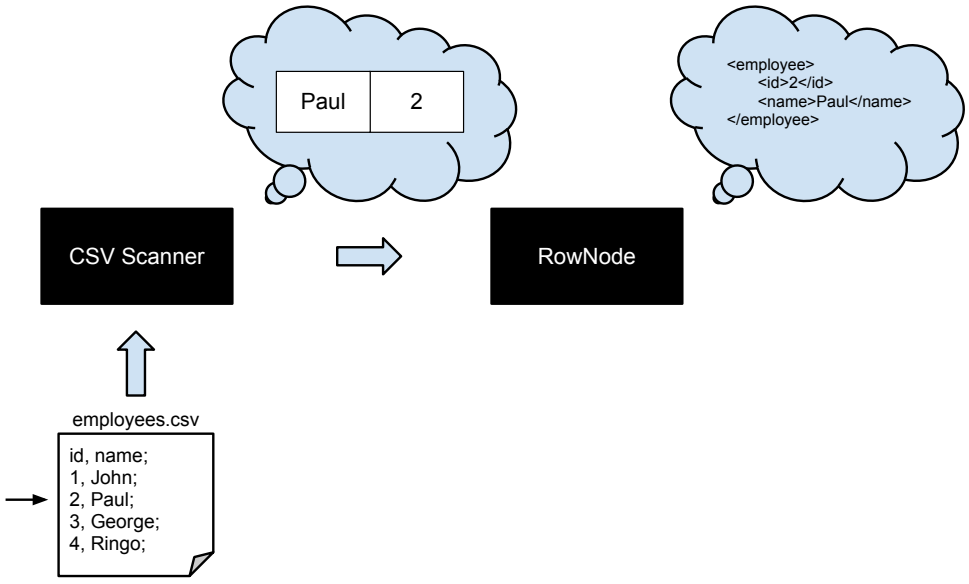

Figure 3.1: Mapping a Relational Table to a Logical Node

Figure 3.1 shows the process with a simple example, in which we are mapping a row to a logical node. The example scans a CSV file, but it could be any file format with a specific scanner associated to it. In our file, the scanner knows that each row is separated by a semicolon, and the columns of each row are separated by commas. Is also defined that the first line is used as embedded schema, so the scanner starts from the second line of the file. Since the stream pointer is on the second logical row of the represented table, the scanner reads it to the tuple *{2,Paul}*, which is wrapped by the *RowNode* simulating an XML node.

3.5 Schema's

To query relational data, it is necessary to define a schema, which is responsible to specify the organization of the stored data. Table and column names, types, lengths and etc. are essential to make possible to the engine to run queries and iterate over the data. An XML document does not requires a schema, but relational tables must be structured. Given this need, we provide an interface to connect different meta-data sources with the SQL interpreter.

In our solution, have been developed specific classes to implement the schema structure, depicted in Figure 3.2. Three main classes store all the required information. The *ColumnInfo* stores specific column informations, such as name, type, length, etc. The *TableInfo* class has all the table information, like stored physical location, name, and an array containing the *ColumnInfo* objects of all the columns belonging to the table. Finally, the *RelationalSchema* class makes the bridge between the meta-data sources and the engine. This class contains all the tables respective *TableInfo* objects, providing methods to access the necessary schema information for each table.

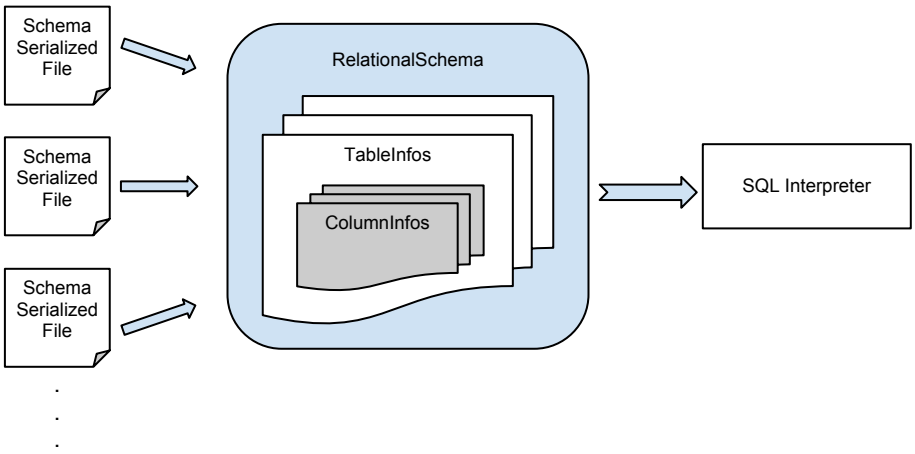

Figure 3.2: Schema Structure

Auxiliary classes help to build the *TableInfo* structure. Similarly to the data scanners, approached previously, the schema makes use of specific reader classes to deserialize the meta-data from distinct data sources, importing this information to the mentioned classes, which are connected to the SQL interpreter. To process a relational query, the tables meta-data is needed, thereby, unlike the data scanners, the schema readers read all the schema information before the query execution, instead of open a stream in the container file. Two different schema readers were implemented in this work:

- Embedded: Reads the schema information from the same CSV file containing the table data.
- XML: Reads the schema information from an specific XML file.

This provided infrastructure allows to generate schemas from a large range of file formats and structures. Each reader is specific, having to understand both the binary codification and the meta-data structure, to correctly generate *TableInfo* objects. Therefore, the SQL interpreter can disregard how the schema data is physically stored/organized, just concerning to communicate with the interface provided by the *RelationalSchema*.

4 TRANSLATION

4.1 Overview

This chapter concerns about the layer “above” the query processor, introduced in section 1.3 and demonstrated in Figure 1.2. Here the SQL interpreter will be detailed, the query translation process is composed by three steps:

1. Parser
2. Analyzer
3. Translator

These steps are explained in sections 4.3, 4.4 and 4.5, while section 4.2 explains the basic mechanisms used by the translation steps. Figure 4.1 elucidates the operation of this pipeline structure.

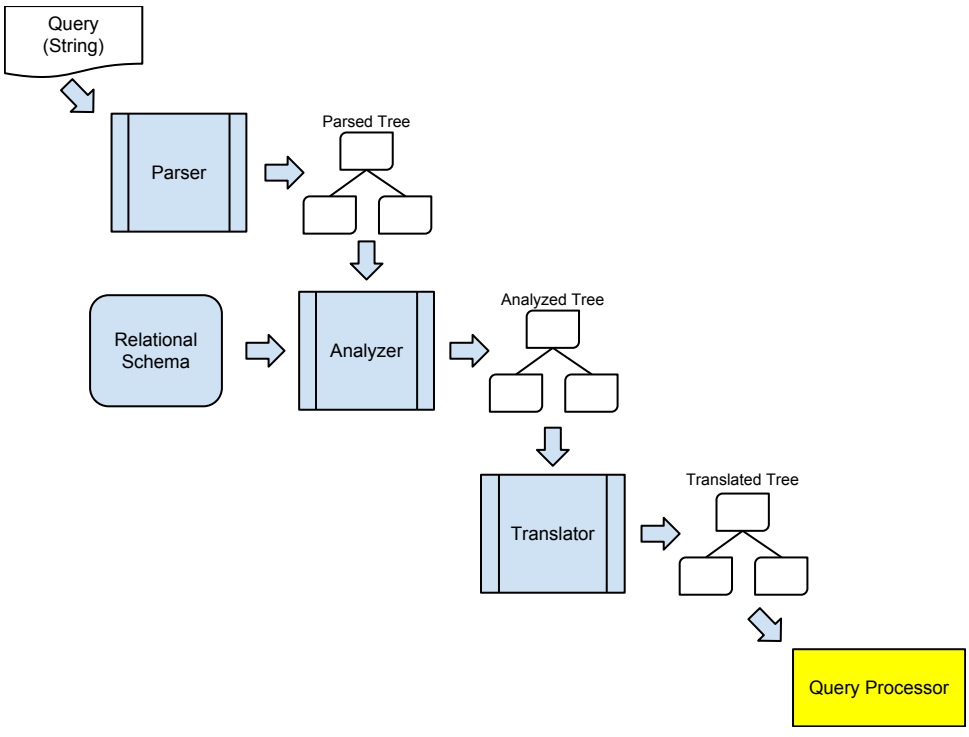

Figure 4.1: SQL Interpreter Pipeline

4.2 Tree grammars and rewrite rules

Aiming to interpret SQL in the Brackit engine, we implemented a tool which parses SQL queries, generating an abstract syntax tree (AST). The tree is read and some analysis are performed, such as table name resolving, aggregation functions detection, columns and tables association, and so on. Finally, it is translated to an XQuery AST, which can be executed by the Brackit engine. As mentioned, the work is divided into three phases, and all of them require manipulation of ASTs. To generate and manipulate these trees a framework called *ANTLR* is used. *ANTLR* is "Another Tool for Language Recognition", it offers a range of mechanisms to create interpreters and compilers from grammatical descriptions (Parr 2007). The target code of these recognizers can be generated in a large variety of programming languages. In our case the target language is Java.

The process of parsing an expression begins with a lexer, which is a tokenizer. It receives a character stream (the string to be parsed) and generates a token stream that will be the parser input. These tokens are generated by matching character sequences of the stream with a list of pre-defined strings (tokens) or regular expressions. The parser is, essentially, constructed by rules. A rule can be defined as an arbitrary composition of other rules or tokens. In this work, we define a set of rules that describe the SQL language. For the beginning of the process, an initial rule is defined, which the input string will be compared to. The tokens produced by the lexer are matched with the sub-rules or tokens, and the parser keeps matching the tokens in a top-down process, until all the input tokens find a match — or not, resulting in an parsing error. The rules return ASTs and can use its sub-rule's ASTs to compose its own one. In this way, composing ASTs, the initial rule returns a unique AST as output that will be processed in other steps of our work.

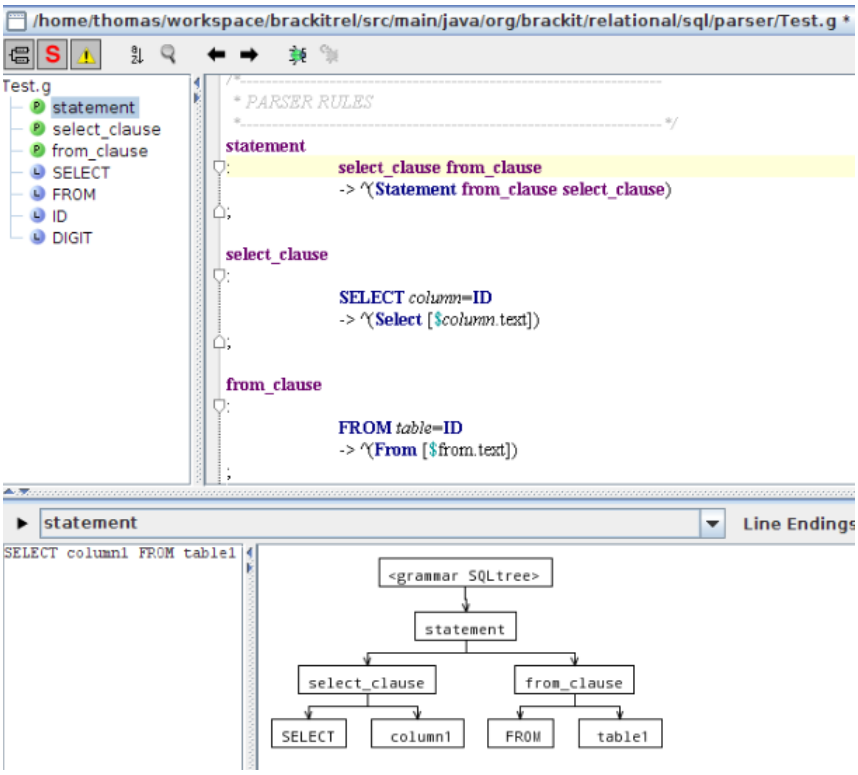

Figure 4.2: Tree Generation Syntax

Figure 4.2 shows a simple example, with 3 simple rules in ANTLR we create an AST correspondent to a simple *select* statement. The picture is a print of the ANTLR screen. In

the upper part, the *ANTLR* code is shown. The first line of each rule contains its definition, whereas the second line is responsible for generating its outcome AST. The first rule is composed by the two following rules. In the lower part, is shown an interpreter with a simple SQL command on the left and a graphical representation of the parsing process generated by *statement* rule, on the right. Although it is similar, this representation do not correspond to the output AST. The arrow indicates a tree construction and it is followed by the nodes structure, represented by nested parenthesis. The nodes are represented by their names and the symbol ^ is used to start a new branch in the tree, where the first node is the parent and the following nodes of the parenthesis scope are its children, Figure 4.2 elucidates this syntax.

After parsing the query and generating an AST, we still have to process the output AST. *ANTLR* provides numerous mechanisms to manipulate them. The parser rules can receive parameters and modify them. Additionally, code in the target language can be embedded into the rules to help in the parsing process, making some more complex manipulations that can not be made with grammar rules alone. An important feature of *ANTLR* is the possibility to create and manipulate trees, known as tree-parsing. It provides tools to recognize, remove, add and move nodes, making it possible to rewrite a tree, for example.

4.3 Parser

The first step in our SQL query processing is to parse the query. Parsing is the process of analyze a text, consisting in a sequence of tokens, to determine its grammatical structure according to a given formal grammar. Our Parser respects the SQL grammar, and produces a tree of tokens, conforming this grammar structure. It receives a string, as input an returns an AST as output.

The most simple of the three query processing steps, but not less important, the parser is responsible for the lexical and the syntactical analysis, producing an output ready to be processed by the analyzer. To facilitate the analysis step, our parser executes a little trick, putting in the outcome AST the *from* clause generated branch as the first one, while the natural behavior would be to put the *select* clause branch as first. This action aims to make it easier to the analyzer, because the tree is parsed from left to right. This trick makes possible to read all the tables first and afterwards process the columns references knowing from each tables they can be from. Besides, this artifice allows the early detection of simple problems, such as inexistent table references. At last, the branch created for the *select* clause is moved to the last position, since it will become the branch correspondent to the XQuery *return* clause.

The main rule of the parser is the *statement* rule, which consists in the *select_statement* rule, followed by a semicolon. The *select_statement* rule is composed by the sequence of sub-rules listed below:

- *select_clause*: Accepts prefixed column references, asterisk (all the columns from all specified tables), prefixed asterisks (all the columns from certain table), and expressions, which can be column references, function calls and etc.

- *from_clause*: Accepts table references and alias assignment. Besides, allows subqueries, which are recursively defined as *statement* rules, but interpreted as table references by the queries above it.

- *where_clause* (optional): Accepts any predicate.
- *groupby_clause* (optional): Accepts column references that can be followed by an optional *having_clause*.
- *orderby_clause* (optional): Accepts column references, each one can be followed by the ASC or DESC optional modifiers.

The parser uses an auxiliar Java function "isAggregate(String functionName)", which identifies aggregation functions, generating thus, an aggregation function branch or a scalar function branch, otherwise. This will be useful to the translation step, as we are going to see afterwards. The rest of the work is made by the *ANTLR* tree generation mechanism, using the defined main rules listed above and their auxiliar sub-rules. Figure 4.3 shows the parsing operation, presenting an input query example an its related output AST after the parsing process. As shown, the parser essentially organizes the input string in a grammatical tree. To demonstrate the evolution of the process, each step of the SQL query processing is ilustrated with a figure, which shows the resulting AST for each one. The query below is the pipeline input and reflects an SQL version of the query shown in Figure 2.2.

```
SELECT    department,
          avg(emp_salary) AS salary
FROM      emps
WHERE     emp_salary > 10000
GROUP BY  department
ORDER BY  department;
```

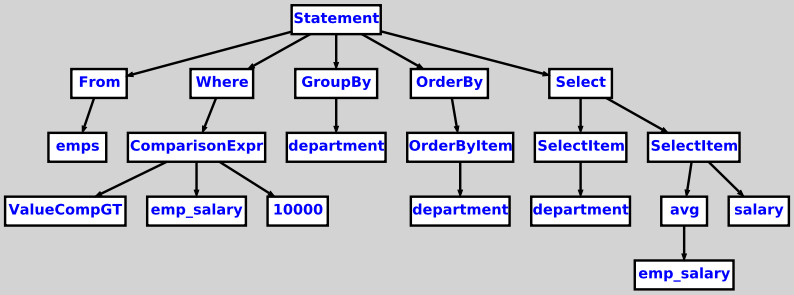

Figure 4.3: Parsed Tree

4.4 Analyzer

This step processes, in high-level words, the action of organize the relation between column references, tables, sub-queries and aliases. Starting from a syntactical tree, the analyzer is responsible for the semantical analysis, producing a tree with resolved references to serve as input to the translation phase. This step makes a wide use of the provided query schema, also making use of a reasonable amount of Java embedded code within the *ANTLR* code.

In addiction to the query schema, is generated an "SQLAnalyzerState" object for each statement, which means one for the original query, and one for each sub-query as well. This object contains all the auxiliar statement structures, that are listed below:

- tableAliases: Map containing the query tables and their respective aliases.
- subquerySchema: List containing a dynamically generated *TableInfo* object to each sub-query, composing a schema.
- outputColumns: List of query column references.
- subqueryAlias: If the statement is a sub-query, indicate its alias.

The first action is to process the *from* clause, which has been passed as the first branch by the parser. In this part, we just register all the tables that compose the clause, followed or not by an alias, in the "tableAliases" map. In order to facilitate the work, all the tables are required to have an alias, dummy aliases are created to tables which come without aliases from the parser. So, from now on, every table reference is uniform, made by its respective alias. Artificially generated, dummy aliases conform a *Tn* pattern, where *n* is a sequential number starting from zero. All aliases, artificial or defined in the query, are attached in the tree, as child nodes of their respective table references. Sub-queries aliases are registered in a specific variable, because they are part of a special push and pop behavior, better explained hereafter.

After storing all the query tables and aliases, we process the column references in all the other clauses. To each column reference, a function that resolves from which table it comes from is called, attaching the table alias to the column reference node in the tree. This function, called "getAliasForColumn", starts looking into the original schema for a table containing the column. If none is found, it looks in the subquerySchemas structure. After finding the table or dynamic table (sub-query) which contains the column, the function returns its alias, otherwise, if no table was found, an error is raised.

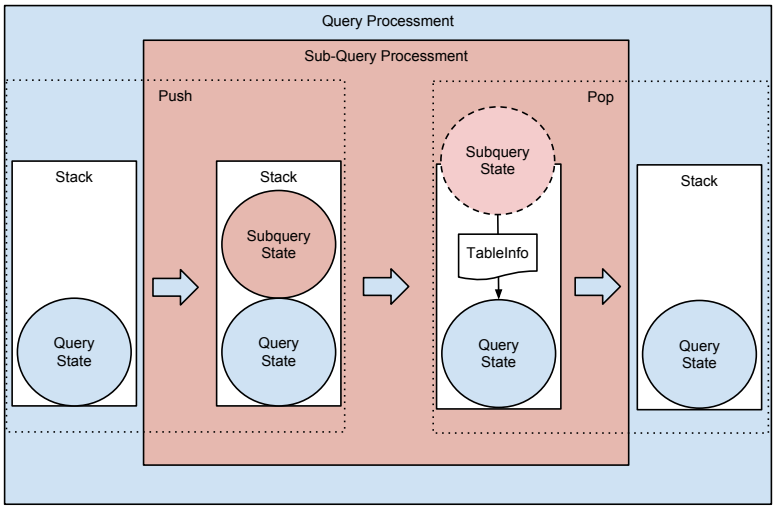

Figure 4.4: Handling Sub-Queries with Push and Pop Operations

To handle sub-queries the analyzer contains a stack of "SQLAnalyzerState's" and manage this trough push and pop operations. The first "SQLAnalyzerState" refers to the main query, while the subsequent refers to sub-queries. Push operations just create new "SQLAnalyzerState's", putting them over the stack. Each pop operation, in the other hand, generate a *TableInfo* object, which is attached to the "subquerySchema" structure of the parent "SQLAnalyzerState". In this way, each query receives all its sub-queries information when the translator finish to process each one, calling a pop operation. The

sub-query “outputColumns” list serves to generate a list of *ColumnInfo* objects that compose a *TableInfo* object, having the “subqueryAlias” as table name. Figure 4.4 shows the functioning of the push and pop operations, while Figure 4.5 shows the AST after the analyzing process.

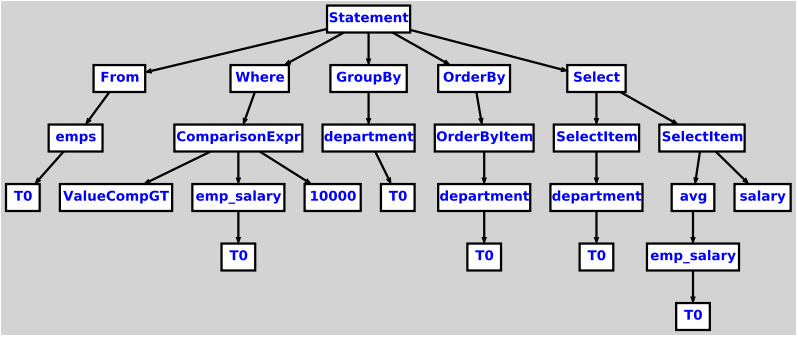

Figure 4.5: Analyzed Tree

4.5 Translator

Taking advantage of the *ANTLR* tree-parsing mechanism, this step performs the last and most complex part of the work, turning the analyzed tree into an XQuery AST. The translator, similarly to the analyzer, contains state objects, aiming to manage sub-queries by stacking them. Each query or sub-query has an associated “SQLTranslatorState”, which contains the following structures:

- letBindings: List of *Let branches* that will be placed as children of the current *FLOWR branch*, after the *For branches*.
- groupedColumns: map containing the column names of the *group by* clause and the dummy variable generated to each one.
- aggregationLetBindings: List of *Let branches*, correspondent to the column references of the *select* clause. They will be placed as children of the current *FLOWR* node, after the *group by* node, because they may refer to aggregation functions, which have to be processed after the grouping.
- implicitGroupBy, selectClause, groupBy: Booleans used to identify the need of adding an artificial *GroupBy branch*.

This state object is also changed in push and pop operations to deal with sub-queries. However, unlike the analyzer, the translator push/pop operations are very simple and do not pass data from lower to upper states.

Initially, the main statement node and each sub-query statement node are replaced by FLOWR nodes. Thereby, the *select* statements turn into *FLOWR* statements. The descendant branches of each statement are translated as follows.

Clearly, we can make an association between the SQL *from* clause and the XQuery *for* clause, because they have, essentially, the same functionality, i.e., they define a data source to be processed by the other clauses. Therefore, in the translated tree, *from* clauses become *For branches*, where the table references are children of a collection node, which

is a special node referring to a function used to read from the serialized stored table files. Sub-queries are not treated as collections, since the resulting data is generated dynamically from another *FLOWR* statement and do not come from serialized files. Both tables and sub-queries are bound to specific variables with the same name than their aliases.

The *select* clause returns the queried data as defined in the query text. Its correspondent in XQuery is the *return* clause, that receives the resulting sequence of the pipeline and writes it in the output XML, according with the specified in the query. To translate the *select* clause, is created a *Return branch* with an *ElementConstructor branch* having as children the string "result", that is the outer tag of the resulting XML node, and a *Sequence branch* containing as children all trees corresponding to each select item.

Each type of expression generates a specific sub-tree. Column references without aliases are directly replaced by *PathExpression branches*, using the column name and its table source. Column references with aliases are replaced by variable references. When the translator matches a column reference with alias, it calls a function that resolves the path expression and binds it to a variable, by putting the expression in a *Let branch*. This *Let branch* is stored in the “letBindings” list, that will be placed as a *FLOWR branch* child, afterwards.

Another kind of expression are the function calls. Scalar function calls are just replaced by a similar one with XQuery syntax, as well as comparison and arithmetic expressions. Aggregation functions, such as sum, average, and so on, are treated in a different way by the translator. Aggregation functions operands must be processed before the function call, since the function needs all the values to aggregate them. Thus, the expression outcome values inside the aggregation function are bound to a variable, by placing the *Expression branch* under a *Let branch* and storing in the “letBindings” list. The *FunctionCall branch* just refers to the bound variable, whose dummy name is generated according to a *Fn* pattern, where *n* is a sequential number starting from zero, just like the dummy table aliases in the analyzer.

Where branches remains, basically, the same, and their child nodes are processed as expressions. *GroupBy branches* have a special behavior. Column references that are going to be grouped must be processed previously and bound to variables, as are the aggregation function operands. A *Let branch* is generated to each column in the *GroupBy branch*, and its name is stored in the “groupedColumns” map, along with its dummy variable name. The dummy variable names follow the *Gn* pattern, where *n* is a sequential number starting from zero. The “groupedColumns” map is consulted in every column reference, if the referred column matches a register in the structure, a variable reference is used, instead of a *PathExpression branch*. A *group by* clause can be followed by a *having* clause, in this case the *Having branch* is replaced by a *Where branch* containing the same expression child. Therefore, each statement may contain two *Where branches*, the second one simulating an SQL *having* clause.

SQL presents a particular behavior when the *select* clause contains only aggregation functions and there is no *group by* clause. All the resulting rows are grouped to one, containing the aggregation functions results. We call this behavior “implicit group by”. To handle this, boolean variables are used as flags to determine if there are only aggregation functions in the *select* clause, and whether or not a *group by* clause exists in the query. This mechanism is used to identify an implicit *group by* behavior, which is handled by creating a dummy variable set to zero, and adding an artificial group by branch in the structure, grouping by this dummy variable. In this way, each aggregation function contained in the *select* clause returns a single value.

Finally, the *OrderBy branch*, as well as the *Where branch*, does not have any special peculiarity, being replaced by a branch with specific XQuery *group by nodes*. Figure 4.6 depicts the final result of the SQL query translation. It is possible to observe that the process outcome reflects quite similarly the structure of the query presented in Figure 2.2, which is an XQuery version of the SQL query used as example to the translation process.

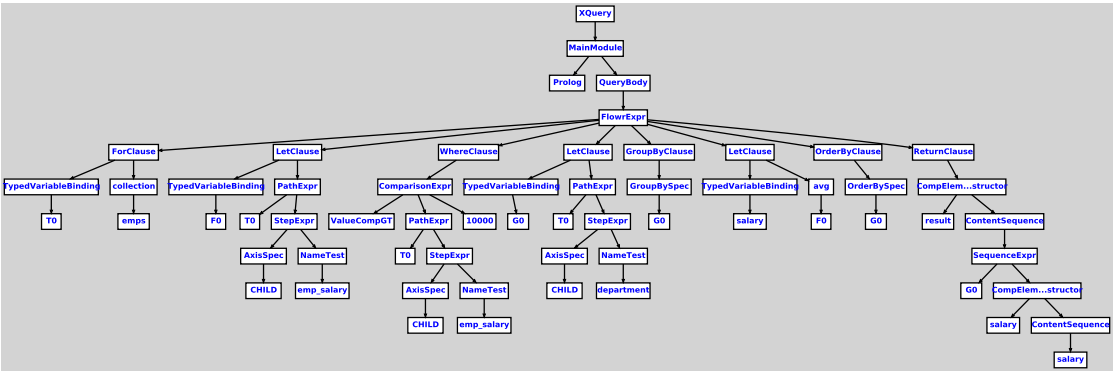

Figure 4.6: Translated Tree

5 EXPERIMENTS

5.1 Overview

Aiming to verify the correctness of our translation, six queries have been written, which cover all the important constructions cited in this work. Consequently, they are enough to verify the achievement of the proposed goals. The test is simple, consisting of running each query in both the Brackit XQuery engine and a native SQL engine, checking whether the same results are generated. Section 5.2 contextualizes the mechanisms used for the experiments. Section 5.3 depicts the datamodel and the queries used for the experiments, while section 5.4 shows the results and analyze them.

5.2 Environment, tools and functions

The native SQL engine used for tests is *HSQLDB* (HSQLDB 2012), which is an open source embedded database, written in Java, offering portability and flexibility. It can be totally integrated with the Java code, without need to install and pre-configure it, such as the most relational databases around. *HSQLDB* is just a jar file, which is included into the project libraries. It has a JDBC driver and supports a large subset of SQL-92 and fully the SQL:2008 standards (HSQLDB 2012).

A database model, including six tables, has been generated to run the experiments. The tables are stored in a folder, under CSV formated files, used as input for both engines. Another folder contains all the queries in separated files, also including two special files with commands needed by *HSQLDB* to create the tables and make the connection with the CSV stored tables. The schema is an XML file assigned to a string within the code.

A function has been developed to compare the output rows for both engines. Since XQuery queries without an *order by* clause produce results in an arbitrary order, when there is no *order by* clause in the query, the XQuery and SQL engines output rows may come in different order. Given this, the comparison function do not care about order, and handle the resulting rows as a set comparison, just verifying if the XQuery result set contains the same registers than the SQL result set, regardless of order. A similar comparison function has been programmed to test queries having *order by* clause, where the order of the resulting rows influences the correctness of the queries.

Finally, a function named “runAndCompare” is responsible for running the same query in both engines and check the correctness, comparing the results using the appropriate comparison function, depending on whether the query has a *order by* clause. This function receives, as parameters, the name of the file containing the query and the desired number of execution repetitions. Although the performance is not the main concern in this work, the average time of the executions is measured, as well as the compilation time

of each one. This measurement aims to show that an acceptable performance is achieved in comparison to a native relational engine.

5.3 Datamodel, datasets and queries

Figure 5.1 shows the database model constructed to test our SQL parser. The model used is a simple sales business, involving persons, products and transactions. There are three kinds of persons: clients, managers and salesman. Each salesman has a manager associated and each manager is responsible for one different department of the store. Every product has a price and a cost, from which is possible to calculate the gain in each sold unit, by subtracting the cost from the price. A transaction is basically a sale of a product from a salesman to some client, also including the quantity of units sold and a possible discount, that have to be considered to calculate each transaction gain.

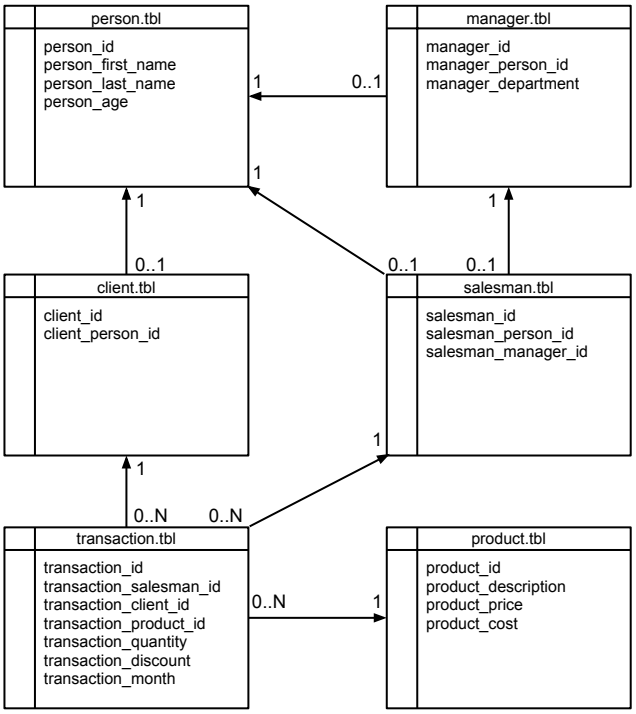

Figure 5.1: Test Database Model

Assuming that performance is not a concern in this work, a dataset of around 100 KB was generated, to test the designed queries, which are shown below:

- Q1: Lists the first and last name of all the persons. Tests the basic clauses *select* and *from*.

```
SELECT person_first_name,
       person_last_name
FROM   person;
```

- Q2: Lists all the salesmen and their respective managers. Tests the *where* clause, as well as comparison expressions. Also tests ambiguous tables, solved by prefixed columns references.

```
SELECT p1.person_first_name AS manager_name,
       p2.person_first_name AS salesman_name
FROM   salesman,
       manager,
       person p1,
       person p2
WHERE  salesman_manager_id = manager_id
       AND manager_person_id = p1.person_id
       AND salesman_person_id = p2.person_id;
```

- Q3: Returns the largest person age. Tests an aggregation function and the implicit group by, that occurs when there are only aggregation functions in the *select* clause and no *group by* clause.

```
SELECT max(person_age) AS max_age
FROM   person;
```

- Q4: Lists the gains of each salesman, ordered from largest to smallest. Tests *group by* clause, *order by* clause with DESC modifier, and arithmetic expressions.

```
SELECT    person_first_name,
          sum((product_price - product_cost) * transaction_quantity
          * ((100 - transaction_discount) * 0.01)) AS gains
FROM      transaction,
          product,
          salesman,
          person
WHERE     transaction_product_id = product_id
          AND transaction_salesman_id = salesman_id
          AND salesman_person_id = person_id
GROUP BY  person_first_name
ORDER BY  gains DESC;
```

- Q5: Lists the gains of managers with gains above 15000, ordered from smallest to largest. Tests *having* clause and *order by* clause with ASC modifier.

```
SELECT   person_first_name,
         sum((product_price - product_cost) * transaction_quantity
         * ((100 - transaction_discount) * 0.01)) AS gains
FROM     transaction,
         product,
         salesman,
         manager,
         person
WHERE    transaction_product_id = product_id
         AND transaction_salesman_id = salesman_id
         AND salesman_manager_id = manager_id
         AND manager_person_id = person_id
GROUP BY person_first_name
HAVING   sum((product_price - product_cost) * transaction_quantity
         * ((100 - transaction_discount) * 0.01)) > 15000
ORDER BY gains ASC;
```

- Q6: Return the gains of the most lucrative month. Tests sub-queries.

```
SELECT max(gains) AS best_monthly_gain
FROM
        (SELECT   transaction_month AS MONTH,
                  sum((product_price - product_cost) * transaction_quanti
                  * ((100 - transaction_discount) * 0.01)) AS gains
         FROM     transaction,
                  product
         WHERE    transaction_product_id = product_id
         GROUP BY transaction_month) AS monthly_gains;
```

5.4 Results

Each query is executed as a unit test, whith *JUnit* (Mentor 2012), using the "runAndCompare" function. *JUnit* is an open source Java framework that facilitates the creation of code for test automation with presentation of results. With it, we can see whether each method of a class works as expected, showing possible errors or failures. Every test has been executed fifteen times, to return the average execution time for XQuery and SQL. The compilation time have been also measured, to calculate the time taken to process the translation of the SQL query to XQuery operators.

Results show that all queries produced the exact same results, confirming the correctness of all the tested constructions. Since the performance was not the focus of this study, the tests were ran in small tables, resulting in a significant time slice being considered by the compiler. However, when it comes to bigger storage files, the compilation time is still the same, while the total execution time of each query grows, decreasing the impact of the compilation time. Regardless of tables size and time taken to execute some

Query	Correct	Compilation time	XQ time	SQL time
Q1	Yes	0.46ms	0.93ms	0.2ms
Q2	Yes	2.07ms	3.53ms	1.4ms
Q3	Yes	0.64ms	1.2ms	0.2ms
Q4	Yes	2.57ms	4ms	1.8ms
Q5	Yes	2.71ms	4.8ms	2.2ms
Q6	Yes	0.92ms	1.8ms	1.6ms

Table 5.1: Experiments Results

query, the translation process remains in the same time magnitude, being not noticeable to the user eyes, thus not significantly affecting the engine performance. Nevertheless, as future work, it is possible to carry out some code optimizations, improving the query compilation time.

6 CONCLUSION

We observed the diversity of data source formats being manipulated on the web togheter with the growth of XQuery as a language to query a widely used kind of data: XML. In addition to this, considering relational databases as a consolidated storage technology, a lack of an integrated mechanism to query different types of data sources offering language independence was noted. Our work presented a solution to the mentioned problem, showing mechanisms to parse and translate an SQL query to XQuery expressions, being able to connect them with an XQuery engine.

In our solution, an SQL parser has been developed. Attaching it to the top of the Brackit engine, we are faced with a very flexible tool, which supports storage independence in the layer "below" the query processor and language independence in the layer "above". In this way, we aimed to abstract as much as possible, in both ends, the process of querying data from different data sources. To validade the developed solution, some queries were run, aiming to check its correctness, which was confirmed with the obtained results. Considering the unconcern with the performance, the proposed objective of this work has been achieved.

Dividing the process into three different steps reduced the work complexity and increased the organization. However, the translation step is still with some complicated manipulation and workarounds, which can be better depicted as future work, enhancing the code legibility and extensibility. Other related works are not as flexible as ours, but some of them offer more SQL constructions, being more abrangent in the regard of SQL coverage. In this way, it is interesting, as future work, to implement missing SQL constructions, such as count, outer join, union and others that are beyond the basic clauses, often used in the queries.

An important topic to be improved as future work are the experiments. More solid tests can be executed, and consolidated benchmarks, such as TPCH (TPC 2012), can be used to verify the SQL standard coverage, correctness and performance of the developed SQL interface. Also, queries optimizations can be performed, aiming to improve the execution time.

REFERENCES

[Codd 1972]CODD, E. F. Relational completeness of data base sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[Eisenberg e Melton 2004]EISENBERG, A.; MELTON, J. Advancements in sql/xml. *SIGMOD Record*, v. 33, n. 3, p. 79–86, 2004.

[Garcia 2007]GARCIA, R. *Foundations Book II: Understanding SQL Server 2005 Supporting Technology (XML, XSLT, XQuery, XPath, MS Schemas, DTD's, Namespaces).* [S.l.]: Lulu.com, 2007. ISBN 1430324465, 9781430324461.

[Halverson et al. 2004]HALVERSON, A. et al. Rox: relational over xml. In: *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*. VLDB Endowment, 2004. (VLDB '04), p. 264–275. ISBN 0-12-088469-0. Available from Internet: <http://dl.acm.org/citation.cfm?id=1316689.1316714>.

[HSQLDB 2012]HSQLDB. 2012. Available from Internet: <http://hsqldb.org/>.

[Jigyasu et al. 2006]JIGYASU, S. et al. Sql to xquery translation in the aqualogic data services platform. In: *Proceedings of the 22nd International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2006. (ICDE '06), p. 97–. ISBN 0-7695-2570-9. Available from Internet: <http://dx.doi.org/10.1109/ICDE.2006.147>.

[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.

[Mentor 2012]MENTOR, O. *JUnit.org Resources for Test Driven Development*. 2012. Available from Internet: <http://junit.org/>.

[Parr 2007]PARR, T. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. [S.l.]: Pragmatic Bookshelf, 2007. ISBN 0978739256.

[PostgreSQL 2012]POSTGRESQL. 2012. Available from Internet: <http://www.postgresql.org/>.

[Query e Groups 2010]QUERY, W. X.; GROUPS, X. W. *XML Path Language (XPath) 2.0 Standard*. [S.l.]: Network Theory Ltd., 2010. ISBN 190696601X, 9781906966010.

[Sauer e Bächle 2011]SAUER, C.; BÄCHLE, S. Unleashing xquery for data-independent programming. 2011.

[TPC 2012]TPC. *Transaction Processing Performance Council*. 2012. Available from Internet: <http://www.tpc.org/>.

[W3C 2004]W3C. *XML Schema Part 2: Datatypes (Second Edition)*. 2004. Available from Internet: <http://www.w3.org/TR/xmlschema-2/>.

[W3C 2010]W3C. *XQuery 1.0: An XML Query Language (Second Edition)*. 2010. Available from Internet: <http://www.w3.org/TR/xquery/>.

[Wood 1999]WOOD, L. Programming the web: The w3c dom specification. *IEEE Internet Computing*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 3, n. 1, p. 48–54, jan. 1999. ISSN 1089-7801. Available from Internet: <http://dx.doi.org/10.1109/4236.747321>.