UNIVERSITY OF CALIFORNIA,
IRVINE

XEditor:
A Language-Agnostic Framework for Graphical Query Editing

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

By

Raman Grover

Thesis Committee:
Professor Michael Carey, Chair
Professor Chen Li
Professor Sharad Mehrotra

2010

The thesis of Raman Grover is approved:

_____

_____

_____

Committee Chair

University of California, Irvine
2010

DEDICATION

To

My parents

in recognition of their worth

"what I am is because of them"

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

ACKNOWLEDGEMENTS

I am extremely indebted to Professor Michael Carey for giving me the unique opportunity of working with him. He advised, encouraged and inspired my research. I am thankful for the confidence he had in me, and for the freedom and responsibility I received in my work.

I would also like to acknowledge Google for their generous support of this project via the Google Research Awards program. I would also like to thank Dr. Alon Halevy of Google and Drs. Len Seligman and Peter Mork of the MITRE Corporation and the OpenII initiative for their encouragement of this work.

Finally, I would like to express my deepest gratitude towards my parents and my better half Payel. They constantly stood by me and ensured that I remain focused.

ABSTRACT OF THE THESIS

XEditor:

A Language-Agnostic Framework for Graphical Query Editing

By

Raman Grover

Master of Science in Computer Science

University of California, Irvine, 2010

Professor Michael Carey, Chair

Graphical editing provides a development environment where a user may write a program in text or may express it in terms of shapes and lines. The graphical representation captures the high-level task that is to be done by the program while abstracting away the underlying language syntax to ac hieve the same end goal. Understanding a graphical representation can be simpler as it does not require a mastery of the syntax rules for the language. Tools have been developed that provide a dual-editing environment for query languages. There the user is presented with a textual view showing the source as well as a graphical view showing an equivalent graphical representation of the source and is allowed to edit in either of the views. Typically such dual-editing tools are tightly coupled with a query language.

In this thesis we describe a framework that provides an environment for two-way editing that is not specific to a query language. The framework works in coordination with a lightweight pluggable language-specific component that understands the rules of the language and is invoked where an understanding of the grammar of the language is

required. Its architecture enables small pluggable components to be written for a variety of query languages. This work exploits the commonality between query languages and has resulted in the successful prototyping of an extensible dual-editing environment for multiple query languages.

CHAPTER 1

Introduction

1.1 Graphical Editing

Graphical editing provides an editing experience where a program written in the form of text is instead, or also, expressible in terms of shapes and lines. The spatial arrangement of graphic objects (shapes and lines) can highlight the structure of a program, providing an overview which can rarely be obtained with a textual description. This may be useful for inexperienced users, who may take advantage of the two-dimensional representation to better understand the basics of program. A novice user, uncomfortable with the syntax rules of the programming language, may prefer making changes in the graphical representation instead of directly editing in textual format. The graphical model is also useful for expert users, who can define complex interrelations just by drawing shapes and forming lines between them.

In two-way editing environments, the user is presented with a textual view showing the source as well as with a graphical view showing an equivalent graphical representation of the source and he or she is allowed to edit in either of the views. In this kind of two-way editing, the graphical view and the textual view are synchronized with each other so that changes made in one view are visible in the other view. Two-way editing provides a paradigm shift from the traditional method of hand-editing source, where it was required that users be fully conversant with the textual language and use it for all tasks.

1.1.1 Graphical Query Editing

A query language is a type of computer language used to formulate queries against databases and information systems. Most query languages are textual in nature, requiring a precise syntactic and semantic analysis to define or understand query expressions, and only in simple cases can a quick glance provide an understanding of the intent behind a query. It is thus desirable to able to express a query in terms of shapes and lines that describe the information to be returned from the query and the intermediate steps or conditions to retrieve the desired information. Graphical editing can be combined with textual editing to provide a two-way editing environment for a query language, which can help make the task of formulating a complex query much simpler [1].

1.1.2 Graphical Data Mapping

Transforming, merging and coalescing data from multiple and diverse sources into different data formats continues to be an important problem in modern information systems. Schema matching is the process of matching elements of a source schema with elements of a target schema. Schema mapping, on the other hand refers to the process of creating a query that maps data instances between two disparate schemas. Both of these are at the heart of data integration systems. Schema mapping tools are typically characterized by a GUI that places a source structure on one side of the screen and a target structure on the other side. Users specify correspondences by drawing lines across these structures and annotating them with features that carry some of the transformation semantics (e.g., filtering predicates, functions, etc.).

Data mapping tools shield the user from hand writing queries or programs for every translation problem at hand. The output of the data mapping process is an

executable query which can carry out the transformation from source data to target data. The generated query is a lower level representation of the data mapping. While all data mapping tools allow users to express mappings graphically, a two-way editing environment, can help user to create/edit correct data mappings more easily. Advanced data mapping tools like ALDSP [4] and Stylus Studio [3] are thus also equipped with a two-way editing environment for the query language used to implement the generated data mappings.

1.2 Motivation

Visual query editors and visual data mapping tools provide two-way editing as a useful feature. The flexibility offered by a two-way editing environment allows users to use either the graphical or the textual view of a query. Data mapping tools also allow users to express data transformations, to create and edit data mappings, as well as to formulate more general queries written in a specific query language such as XQuery.

Most query languages, particularly those used for data transformation share a common denominator - namely, a set of high-level logical operations expressible through the language. To form a graphical representation of a query, the user essentially uses shapes and lines to represent the sequence of high-level operations for the query. The same sequence of operations would be expressed differently using the syntaxes of different languages, but semantically the query remains the same.

The question that forms the motivation behind the work described in this thesis is:

Can we develop a single robust system that allows two-way editing, not for a specific query language, but that instead has a plug-and-play architecture that can enable the system to provide graphical and textual query editing for a language of interest?

In this thesis, we demonstrate that the task of building a two-way graphical editor for a given query language can be simplified by exploiting this common denominator. It is not necessary to build each editor from scratch, re-implementing a polished user-interface and re-solving problems related to layouts of shapes and connections, graphically representing hierarchical data sources, or even providing for advanced text-editing features like code assistance. Instead, these functionalities can be provided by a single re-usable framework. In this thesis, we present the principles and a prototype for such a framework, which is henceforth referred as XEditor.

The framework described here does not entangle itself with a specific language grammar or syntax rules. Instead it has a pluggable language component that understands the rules of a given language and that is invoked in various situations that require understanding of the grammar of the language. This architecture enables small pluggable components to be written for a wide variety of languages, plugging them into a single re-usable framework, henceforth yielding a dual-editing environment for a given language of interest.

1.3 Related Work

A number of tools have been developed that provide a graphical editing experience for a specific query language. A visual query language for expressing a large subset of XQuery was provided as XQBE (XQuery by Example) [5]. XQBE was designed with the main

objective of being easy to use, highly expressive, and directly mappable to XQuery. XQBE allows for arbitrarily deep nesting of XQuery FLWOR expressions, supports construction of new XML elements, and permits restructuring of existing documents.

A graphical editor for SQL was presented in [8]. This paper presented a system intended for both the direct visual specification of relational database queries as well as the visual description of SQL queries. It introduced GraphSQL, a visual language, in which SQL statements can be expressed graphically. In addition, textual SQL statements can be turned into GraphSQL graphical representations.

In mid-2005, BEA introduced the AquaLogic Data Services Platform (ALDSP 2.0) as a middleware platform developed for building services, referred to as data services, that integrate access and manipulate information coming from multiple heterogeneous sources of data. In order to provide an effective graphical tool to help data service developers quickly and easily develop their information integration queries, XQE [6] was developed as ALDSP's graphical XQuery editor. XQE handled the full XQuery language and provided a robust two-way editing experience involving both graphical and source views of each query.

Another commercial tool that is similar to XQE is the XQuery mapper from Stylus Studio. Like XQE, the Stylus Studio tool aims to provide a two-way editing experience. Unlike XQE, Stylus Studio's mapping model has a fairly rigid sources-on-the-left, target on-the-right editing paradigm. Mid-screen, Stylus Studio has a rich editing trough where one can place functions and FLWOR boxes along the paths of the left-to-right data lines. XQE and Stylus Studio differ significantly in terms of their graphical granularity. Stylus Studio represents boolean expressions and other mapping expressions

(such as arithmetic expressions and function calls) as networks of edges and nodes in the editing trough, while XQE opted to allow snippets of XQuery source to be added and edited using its expression editor.

A commercial tool that offers data integration services, similar to BEA's ALDSP, is the graphical data mapping tool by Altova, known as Mapforce [2]. MapForce is a graphical data mapping, conversion, and integration tool that maps data between any combination of XML, database and/or web service, and can even view and save the XSLT 1.0/2.0 or XQuery execution code. Mapforce allows the user to graphically generate an executable query to implement their data mapping; it does not offer a two-way editing environment or allow creation of a graphical representation of an existing textual query.

Clio[7] is another significant tool for generating mappings between relational and XML Schemas. A Clio user is presented with the structure and constraints of two schemas and is asked to draw correspondences between the parts of the schemas that represent the same real world entity. Correspondences can also be inferred by Clio and then verified by the user. Once the mapping has been completed, Clio can generate the (SQL, XSLT, or XQuery) queries that drive the translation of data conforming to the first (source) schema to data conforming to the second (target) schema. Clio does not aim to support two-way editing or graphical rendering of existing queries.

In addition to these graphical query editors and data mapping tools, a number of schema matching tools have also been developed. Harmony [9] from MITRE is one such tool. Schema mapping tools typically display three vertical panes. The left and right panes show the two schemas to be matched; the center pane is where the designer defines

the correspondences, usually by drawing lines connecting the appropriate parts of the schemas. Schema matching tools help the user to express the elements in the source and target schema that relate to each other, but they usually do not have the user express the exact method needed to transform data instances from the source schema to the target schema.

The tools that have been briefly described here each provide an editing environment for some specific query language. In contrast, XEditor is a graphical query editor that allows two-way editing, and not for one specific query language, but with a plug-and-play architecture to support multiple languages.

1.4 Organization

The remainder of the thesis is structured as follows:

- Chapter 2 describes the underlying architecture of the XEditor framework in detail. It explains how XEditor interacts with external pluggable components.
- Chapter 3 describes the graphical language used to represent a query. The language described is independent of the syntax/grammar of a specific query language and is concerned only with the high-level logical operations commonly applied on data.
- Chapter 4 details the process of parsing and subsequently analyzing a query to identify the sequence of high-level logical operations that the query intends to perform. This process is termed the Query Analysis phase.

- Chapter 5 describes the formation of a graphical representation of a query from the elements of information collected in the 'Query Analysis' phase. This is termed the Query Rendering phase.
- Chapter 6 describes the mechanisms required to edit a query, both graphically and textually. It explains multiple two-way editing scenarios and how they are handled by XEditor. It further describes how the editing responsibility is split between XEditor and the language-specific pluggable component.
- Chapter 7 describes the usage of XEditor to support two popular query languages namely XQuery and SQL. It also briefly explains how XEditor might be used to support other languages.
- Chapter 8 presents the conclusion of this work. It highlights the salient features of XEditor and the lessons learned as well as discussing several areas of required future work.

CHAPTER 2

A Language-Agnostic Framework for Graphical Query Editing

The previous chapter mentioned the concept of two-way editing, whereby a query is editable in either a graphical or textual view. The concept has been implemented for query languages like XQuery and SQL, but the implementations have been tightly coupled with the target language. State of the art two-way editing systems that support data mapping also provide for graphical editing, but most are also tightly coupled with a specific query language for implementing the given mapping. This chapter talks in detail of a different solution – XEditor, a framework that supports two-way editing of source code and yet is language-agnostic.

Being language-agnostic essentially means that XEditor does not need to understand the grammar/syntax of the target language. It instead works in coordination with a language- specific pluggable component that understands the language grammar. This chapter elaborates the design goals of XEditor. It then describes the approach that we adopted in building XEditor and details the software architecture that allows XEditor to be language- agnostic.

2.1 Basic Design Goals

This section walks through the desired characteristics of XEditor, describing the importance of each feature.

2.1.1 Loose Coupling With the Textual Language

XEditor should support graphical editing - not for one specific language, but for different query languages. XEditor should work at a semantic level and should concern itself only with the high-level logical operations that a query may perform on data. XEditor must have a default mechanism to depict these operations on the graphical user interface, and it should only interact with a language-specific component when the task requires an understanding of the language grammar.

2.1.2 Usable With Any Text Editor

Text-based source editors are available that provide advanced features like code assistance, linking with compilers, etc. These features make textual query editing easier as well as more productive. XEditor should not reinvent the wheel to provide such features; instead, it should be able to leverage investments already made in state-of-the-art textual source editors. XEditor must therefore provide an interface whereby a textual source editor can optionally be plugged into the system so that the source view is handled by that textual editor. The user should be allowed to edit using the chosen textual editor, with XEditor taking care of forming the graphical view. Similarly, changes made in the graphical view should result in modified source code in the textual editor. In this way, as new features are added to a textual editor, XEditor can benefit from them.

2.1.3 Customizable User Interface

Each language has its own grammar that describes its syntax rules as well as its language-specific constructs or clauses. XEditor should have a default way of graphically representing high-level operations, but it should also be customizable to cater to the

requirements of a specific language. For example, a language might want to show filtering of data via a box with a specific shape, or it may prefer showing it via some type of connection. XEditor should allow for such customizations.

2.2 Advanced Features

In addition to the basic design goals mentioned above, XEditor should also provide certain advanced editing features. The following sections briefly describe some of the desired advanced features.

2.2.1 Error Handling

On a graphical view of a grammatically correct query, the user will be able to edit and may potentially make the resulting query incorrect as per the grammar of the language. XEditor, though not conversant with the grammar of the specific language, should be able to detect and show such errors while staying within the graphical view. That is, XEditor should not force the user to shift to the textual view to find the error and correct it. The incorrect part of the query should be marked differently in the graphical view, clearly indicating to the user where work is required to remove the error.

2.2.2 Handling Comments

Comments embedded within a query often describe a section of the query and are visible in the textual view. Based on the proximity of a comment's boundaries to neighboring sections of the query, it should be possible to associate each comment with the section of the query that it (most likely) corresponds to. XEditor should attempt to form this association and show the associated comments as tooltips in the graphical view.

2.2.3 Graphical Editing of Subqueries

A subquery is a query within a query and is also referred to as a nested query. A subquery can represent a fixed value, a correlated value, or a list of values. The value represented by a subquery is used in a clause that is defined in the outer query. The clauses that exist as part of the subquery can be graphically represented and placed alongside the graphical representations of the clauses belonging to the outer query. Alternatively, the graphical representation of the clauses that are a part of the subquery can instead be packed inside the graphical representation of the outer clause. The former is an unpacked representation, while the latter is a packed representation. An unpacked representation results in a larger number of shapes on the user interface in comparison to a packed representation. XEditor should support both kinds of representation and allow graphical editing of subqueries.

2.2.4 Detecting Unused Sections of a Query

A large, incrementally constructed query may involve declared variables which are not used subsequently, and hence removing their declaration would have not change what the query intends to do. XEditor should be able to detect such unused variables and mark their graphical representation differently, indicating to the user that they are unused and can be removed.

2.3 Overview of Approach to Building a Language-Agnostic Framework

XEditor is a graphical framework providing both graphical and textual views of a query in a given query language. A user of XEditor uses a graphical language to form a graphical representation of a given query. In doing do, the user interacts with data using a

defined set of high-level logical operations. Sorting the data, filtering data on basis of some predicate, and iterating through a list of data items are examples of the high-level logical operations commonly performed on data. The graphical language thus defines shapes, notations and different kinds of connections that are used to represent the high-level operations to be carried out in a given query. A detailed description of the graphical language as well as the high-level operations identified by XEditor can be found in Chapter 3.

To identify the high-level operations being performed in a query, an understanding of the language grammar is required. For this, XEditor relies on a language-specific pluggable component that is conversant with the language grammar. This component resides outside the core of the framework so that a similar component written for a different language could be plugged in to obtain a similar editing environment for that language. The pluggable component transforms the query text into an Abstract Syntax Tree (AST) representation. During this transformation, any comments present as part of the query text are ignored, as they do not form a part of the language grammar. The high-level operations mentioned above are each expressible through a corresponding clause that represents the operation in the textual language syntax. Each such clause can be identified by the pluggable component by traversing the AST. During traversal, the pluggable component is also given a handle to a symbol table. The symbol table is used to record occurrences of clauses and declared variables together with the scope in which they occur. Each of the elements in the symbol table, such as scopes, clauses, and variables, are referred to by unique IDs assigned to them.

At the programming level, XEditor offers the plug-in developer a list of base classes, with each one representing a high-level operation. The language-specific component can choose to simply use these base classes or it may may derive from them (adding any custom fields) if desired. While traversing the AST, the language-specific-component keeps track of the current scope, as the rules governing begin or end of scope are language-specific and are known only to the pluggable component. If a clause definition is encountered, the corresponding class that represents the operation is identified. Information about the clause is captured in the fields inside an object of the class and a reference to the object is put in the symbol table. Similar steps are followed for each variable declaration. At the end of the traversal, XEditor has thus constructed an ordered list of clauses and variables, each grouped by the scope in which they occurred in the query, and the required detailed information resides in the symbol table.

In a query, the output of a high-level operation may be fed as an input into another high-level operation. XEditor depicts such relationships as directed lines or connections. XEditor must be able to identify all such relationships and form connections between the shapes/symbols that represent the high-level operations; the relationships can be deduced by analysis of the symbol table. The set of shapes and the connections between them then form the graphical view of the query. Breaking a query into logical operations and building the relationships between the logical operations comes under the Query Analysis phase. Once this understanding has been built, the next phase consists of forming shapes and connections, and is referred to as the Query Rendering phase. These phases are explained further in Chapters 4 and 5, respectively. Continuing the present discussion, the underlying software architecture of XEditor is described next.

2.4 XEditor Architecture

As designed, the XEditor architecture must support a dual-editing environment, as the user may switch from one view to the other at any time as per their convenience. Changes made in one view must be synchronized with the representation in the other view. Figure 2.1 shows the underlying architecture of XEditor and is followed by a description of each of the components.

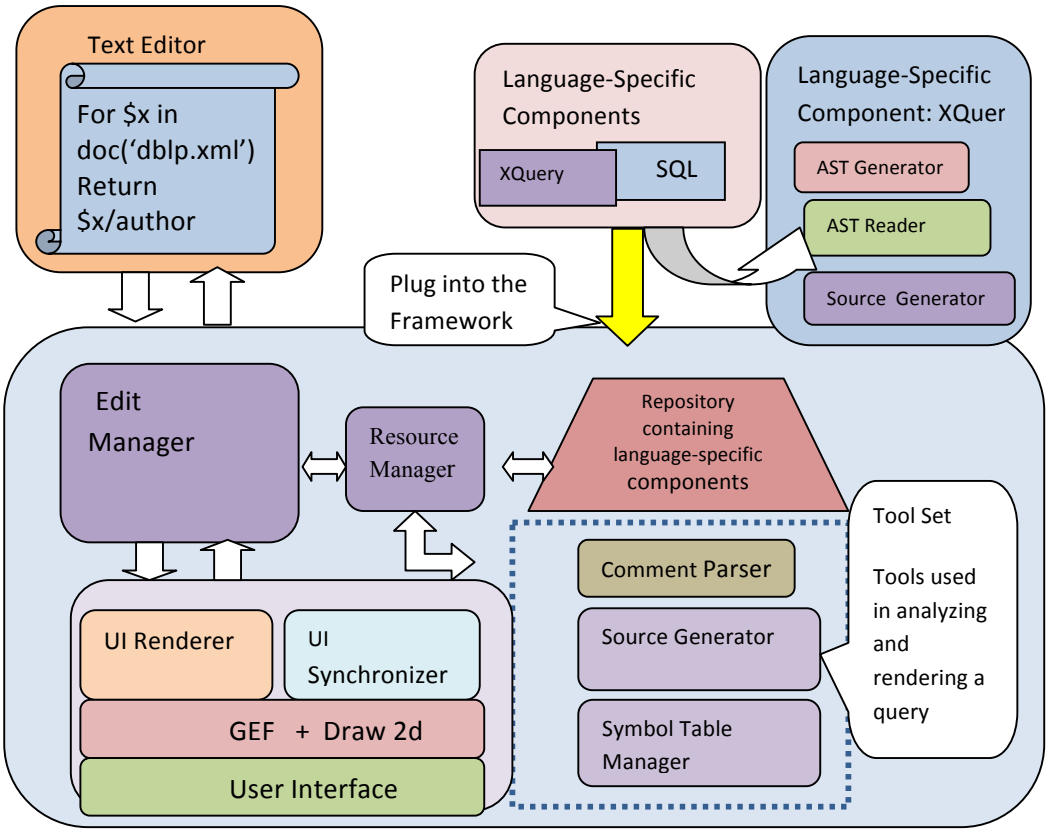


Figure 2.1 Underlying Architecture of XEditor

1. **Language-specific component**
    a. **AST Generator**: The AST Generator, as the name suggests, is responsible for converting a query into its equivalent AST representation. This step is

a prerequisite for any further analysis of the query. The AST Generator is a part of the language-specific pluggable component.

b. **AST Reader**: The AST Reader traverses the AST, keeping track of the clause definitions as well as any variable declarations. The AST reader understands the high-level logical operations and maps each encountered clause to the corresponding high-level logical operation. These high-level logical operations are described in detail in the Chapter 3.

c. **Source Generator**: A user may cause modifications in the query expression through interaction with the graphical user interface. The Source Generator is conversant with the language grammar and receives information about user actions from XEditor. If it is possible to correctly generate code, the Source Generator responds with the change; else it simply returns a null. In the latter case, XEditor opts for receiving the change in the form of text through user input. In the case of a non-null response, XEditor integrates the change into the query and then validates the new query expression.

2. **Resource Manager**: A query may refer to pre-defined functions or data sources such as documents or tables. Information about pre-defined functions and data sources is available via the Resource Manager. When any pre-defined function or data source is referenced in the query and needs to be represented graphically, the Resource Manager is able to return the information required to form a graphical representation. The Resource Manager also holds references to other tools such as

the Symbol Table Manager, Comment Parser, etc., and makes them available to the Edit Manager.

3. **Symbol Table Manager**: The symbol table manager is a store for all information related to the clauses and variables discovered in the query. It provides a simple API which is used by the AST Reader to register clauses and variables.
4. **Edit Manager**: In providing a dual-editing environment, various generic components like the Symbol Table Manager and the Comment Parser need to be instantiated with specific parameters allowing them to be used for a specific query language. XEditor follows a pre-defined sequence of steps in order to analyze and render a query. These steps are initiated in the correct order by the Edit Manager.
5. **UI Renderer**: XEditor utilizes the Eclipse Graphical Editing Framework and Draw2d [10] for painting the graphical representation of a query in terms of shapes and lines. The UI Render interacts with GEF and forms a data model that can be processed by GEF.
6. **UI Synchronizer**: Changes made in the textual view must be applied in the graphical view in a manner that does not disturb the initial layout of the query. This responsibility is handled by the UI synchronizer.
7. **Source Generator**: A change in the graphical view may result in changing the initial query expression and hence must also be applied to the textual view. If the change is complex, the Source Generator relies on the language-specific component to help generate source code in accordance with the action taken in the graphical view. Furthermore, the Source Generator makes sure that changes are

applied in the textual view without disturbing the initial indentation/alignment of the text.

CHAPTER 3

Graphical Language

XEditor abstracts itself from the varied syntax/grammar rules of any particular query language but exploits the degree of commonality in the high-level logical operations that are expressible through most query languages. XEditor uses a graphical language to form a graphical representation of these high-level operations. This chapter describes the high-level operations as well as the graphical language used by XEditor.

3.1 High-Level Operations Expressible through a Query

Query languages inherit a number of properties common to programming languages in general. At compile time, the source in a query language consists of identifiers, expressions, clauses, and optional comments. In addition, there are scoping rules which decide the visibility of variables/functions in a sub-section of the query. Considering only query languages, it is additionally possible to identify a set of high-level operations on data that are common to multiple query languages. These high-level operations are described next.

1. Iteration

   Data may be statically stored in a file, or it may be dynamically generated upon evaluation of an expression or invocation of a function. Iteration allows for the traversal of one or more such sources of data and the production of a list of data elements. Each data element traversed may be subjected to manipulation by other high-level operations. For example, when using SQL to query against a relational

database, traversal over the rows contained in one or more relations is a form of iteration and is expressed using the FROM clause.

2. Filtering

   Filtering is the process of selectively including data elements that satisfy a given predicate or a condition. For example, consider retrieving information about employees working in an organization. One may wish to include only those employees that have work experience greater than a specific value. Filtering helps in applying a set of one or more predicates on attributes of data. Filtering may also involve the joining of multiple sources of data on a common attribute.

3. Sorting

   Sorting refers to the task of arranging the elements in a list in either an ascending or descending manner based upon the value of one or more attributes. A sort operation specifies a set of attributes and their orders.

4. Binding

   Binding allows the association of a value with a variable. The value may be an expression or the output of another high-level operation.

5. Collecting

   Collecting refers to the task of accumulating and shaping the data that forms the result of a query. For example, consider having a list of employee records with each record having some number of attributes. One may wish to retrieve information about employees, but to include only a subset of the attributes for each record. Collecting helps in structuring the result of a query.

Figure 3.1 describes the high-level data operations involved in two example queries written in XQuery and SQL.

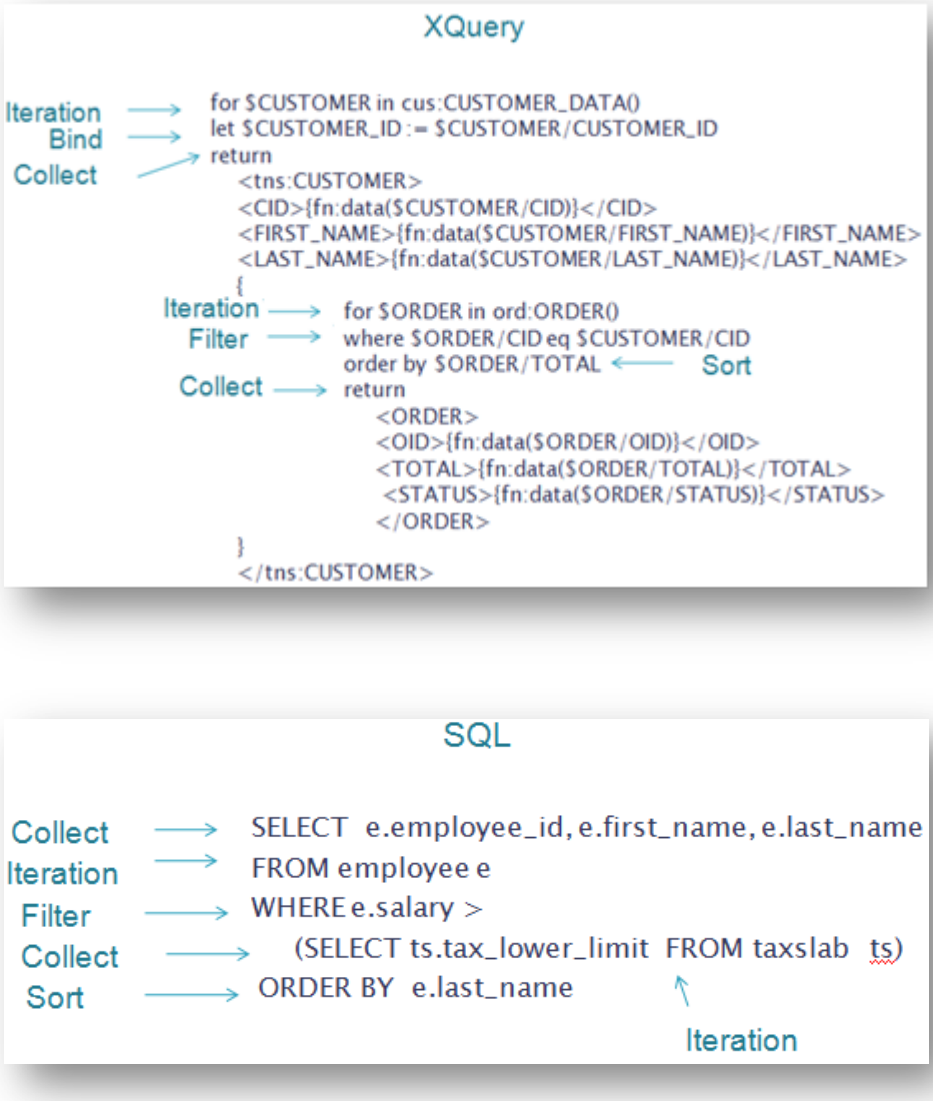


Figure 3.1: Example queries written in XQuery and SQL consist of a sequence of high-level operations.

The task of extracting the desired information from a database or any information repository involves a combination of the high-level operations mentioned above. A query language offers a set of clauses, each corresponding to one of these high-level operations. For example, XQuery offers the **for** clause for expressing an iteration, while iteration is

expressed in SQL using the **FROM** clause. The graphical language used by XEditor is based on the high-level operations discussed above; it defines shapes and other methods to graphically depict these high-level operations.

3.2 Graphical Language of XEditor

XEditor treats a query as a sequence of the high-level data operations, just discussed. For forming the graphical representation of a query, each of the high-level data operations is converted into a corresponding graphical representation. XEditor uses rectangular shapes to build the graphical view for a query. In addition to using rectangular shapes, XEditor also uses several different kinds of lines that connect the rectangular shapes and have specific semantics. In order to understand the graphical representation of each high-level operation and the relevant example queries, it is imperative to first understand the kinds of lines or connections used by XEditor. This is described next and is followed by a discussion of the graphical representation of each of the high-level operations.

(i) Cardinality line: Cardinality lines indicate the repetitive occurrence of a clause, with the repetition being controlled by another clause definition. For example, iteration allows traversal over a list of data elements. Each data element traversed during iteration may be subjected to other high-level operations like filtering and/or sorting. Each data element that is not discarded by filtering is structured by the collect operation before becoming a part of the query result. Thus the number of data elements subjected to the collect operation is influenced by the number of data elements traversed during iteration. There exists a quantity relationship between iteration and collection.

To depict this relationship, an iteration operation is connected to corresponding collect operation using a cardinality line.

(ii) Data-flow line: Data-flow lines are used to represent flow of data from a source to a destination where the data gets used. There are several high-level operations that involve a flow of data between them. For example, when the value of an expression is assigned to a variable using the bind operation, data-flow lines are drawn that connect the graphical representation of each term in the expression to the graphical representation of the variable being defined.

As another example, consider a filter operation that applies a condition on one or more attributes of data. The value of each attribute gets used in determining the output of the filter. This is depicted by drawing data-flow lines that connect each attribute to the shape that represents the filter operation.

(iii) Join line: We may wish to iterate over multiple lists of data elements and join the data elements that have some attribute in common or satisfy some predicate. In such a case, we are joining data based upon a filter condition. The filter condition may involve variables and is depicted by a join line connecting the variables.

We have described the different kinds of lines or connections that are used by XEditor. We next describe the graphical representation for each of the high-level operations understood by XEditor.

1. Iteration

Iteration allows the traversal of one or more such sources of data and producing a list of data elements. In order to form a graphical representation, each data source needs to be represented graphically. XEditor uses a rectangular shape to represent a data source. Hence, iteration over 'n' data sources is represented using 'n' rectangular boxes. The contents inside the rectangular shape depict the shape of the elements coming from the data source and may vary as per the kind of data source being represented. For example, a relational table can be a data source when writing a query in SQL, while a function returning an XML structure can act as a data source in the case of XQuery. XEditor does not understand the semantics behind a relational table or a XML schema. XEditor uses a generic tree structure to depict the shape of the elements coming from the data source. Figure 3.2 illustrates the template for building a representation of a data source.

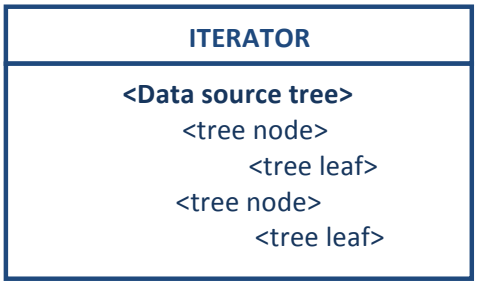
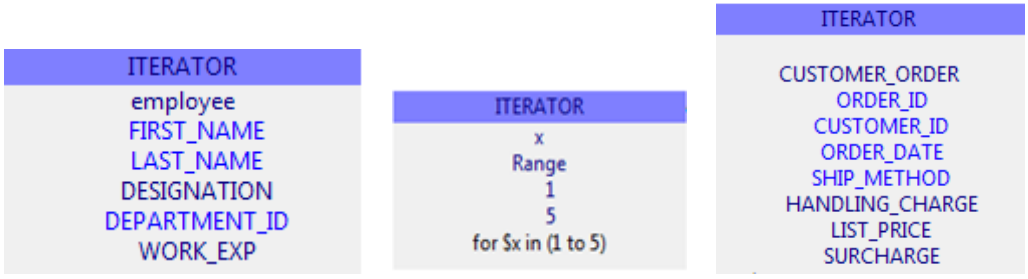


Figure 3.2: Template for building a graphical representation of an Iterator

Figure 3.3 shows different kinds of representations, derived from the same template, that differ because of differences in the type of data source.



(i) (ii) (iii)

Figure 3.3: The representation of iteration depends upon the method used to produce data elements. Case (i) represents an iterator producing data elements by traversing through a SQL table. The representation shows the table name and columns of the SQL table.
Case (ii) represents an iterator running over a range expression with start and end limits. The representation shows the start and end limits.
Case (iii) represents an iterator traversing over data elements produced as a result of invocation of a function. The representation shows the DTD of the XML type returned by the function.

As evident from the above discussion, iteration over multiple data sources would be graphically represented using as many rectangular boxes. For example, a SQL query retrieving data from two relational tables involves iterations over two different data sources and hence the graphical representation includes two rectangular boxes, each representing a relational table.

2. Filtering

Filtering involves applying a set of conditions on data elements to discard unwanted results. A filter may involve applying a conditional expression involving one or more fields or joining two data sets on a common attribute. The graphical language defines a different representation for each case. A conditional expression involving one or more fields is represented using a rectangular shape. The fields involved in the conditional expression are connected to the rectangular shape using separate

data-flow lines. Figure 3.4 shows the graphical representation of a filter involving a conditional expression.

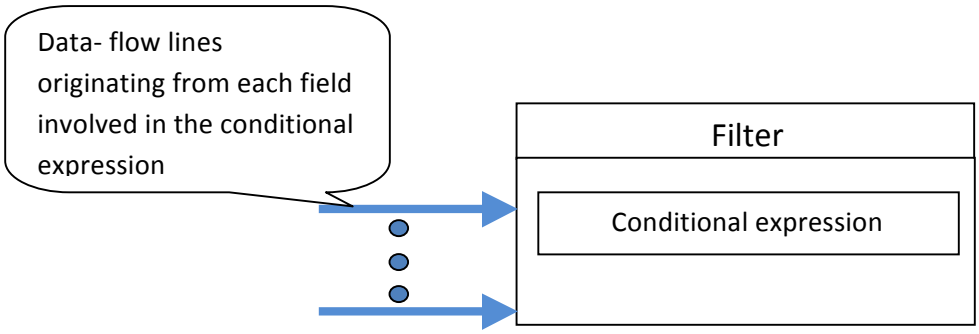


Figure 3.4: Graphical representation of a filter used for applying a predicate

To understand the representation better, consider the example query below.

```
for $EMPLOYEE in cus:EMPLOYEES()
where $EMPLOYEE/WORK_EXP >=5
return
     <EMPLOYEE>
      <ns1:FIRST_NAME>$EMPLOYEE/FIRST_NAME</ns1:FIRST_NAME>
      <ns1:LAST_NAME>$EMPLOYEE/LAST_NAME</ns1:LAST_NAME>
      <ns1:DEPARTMENT>$EMPLOYEE/DEPARTMENT</ns1:DEPARTMENT>
      <ns1:WORK_EXP>$EMPLOYEE/WORK_EXP</ns1:WORK_EXP>
      </EMPLOYEE>
```

The query selectively retrieves employees with work experience more than 5 years. In this case, work experience is a field on which a conditional expression is applied. Figure 3.5 shows the graphical representation of the query.

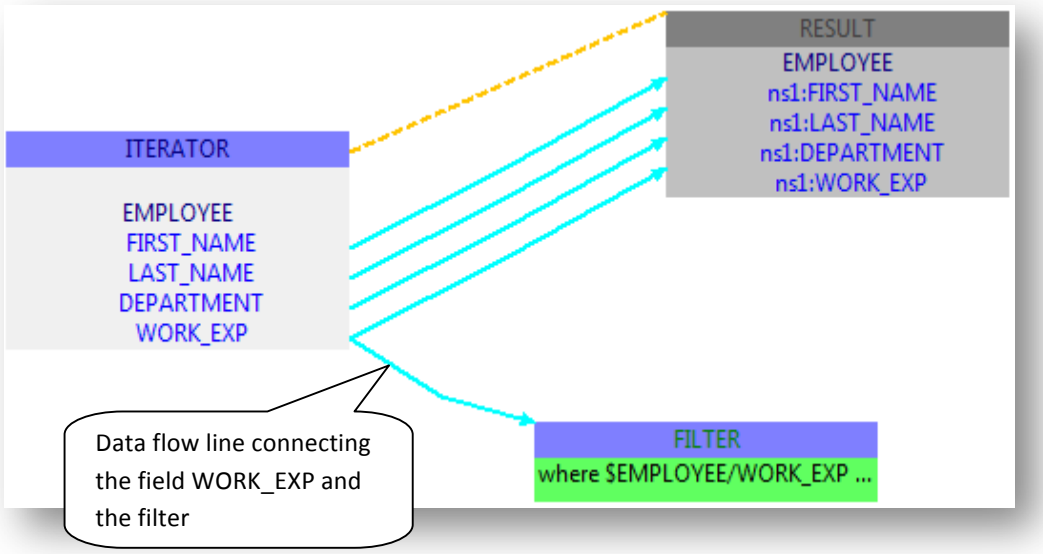


Figure 3.5 Graphical representation of a query that uses a filter used for applying a predicate

In the graphical representation, filtering is depicted using a rectangular box. A data-flow line connects the field WORK_EXP and the rectangular box. It indicates that the employee records traversed through iteration are filtered based on the value of the field – WORK_EXP.

As stated earlier, a filter may also be applied for joining two data sets on a common attribute. As an example, consider having two data sets, one comprising of a list of customer orders and other comprising of a list of customers. We wish to join the two data sets using customer id as the common attribute. The data sets themselves are represented using iterator boxes. Each iterator box shows the attributes that are part of each record. The join condition is represented using a straight line that connects the common attribute across the two iterator boxes. Figure 3.6 illustrates a filter being used for forming a join condition.

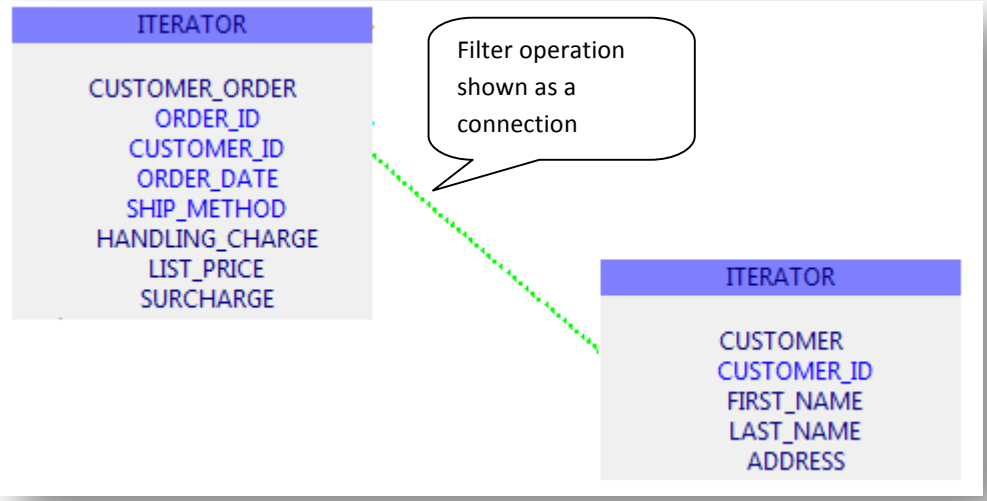


Figure 3.6: Graphical Representation of a filter used for joining. In the illustration above, a join exists between two data sets using the common attribute- "CUSTOMER_ID".

3. Binding

Binding operation involves assigning a value of an expression to a variable. The expression may be a constant, a function call, another variable, or may be composite in nature consisting of sub-expressions. For example, consider the following snippet from a query written in XQuery language.

```
let $CUSTOMER_ID := $CUSTOMER_ORDER/CUSTOMER_ID
```

The value assigned to the variable $CUSTOMER_ID is derived from an attribute of the variable $CUSTOMER_ORDER. This association is depicted using a data-flow line that in the given case, connects the specific attribute of the variable $CUSTOMER_ORDER to a rectangular shape representing the bind operation. This is illustrated in Figure 3.7.

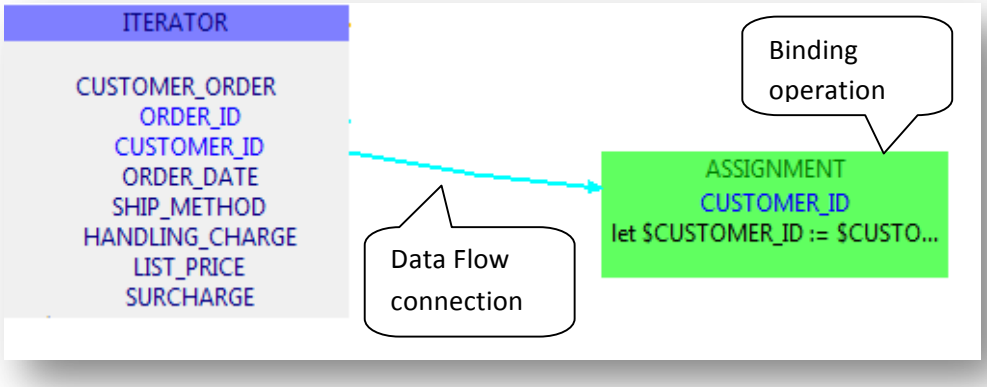


Figure 3.7: Graphical representation of a bind operation

In a general case, where the value being assigned to a variable may be derived from multiple sources, multiple data-flow lines are drawn each connecting the specific source to the rectangular shape representing the bind operation.

4. Sorting

A sort operation is described by the specifying the source of input data, the attribute(s) to sort on, and their sorting order(s) (either ascending or descending). The graphical representation for a sort operation is shown in Figure 3.8.

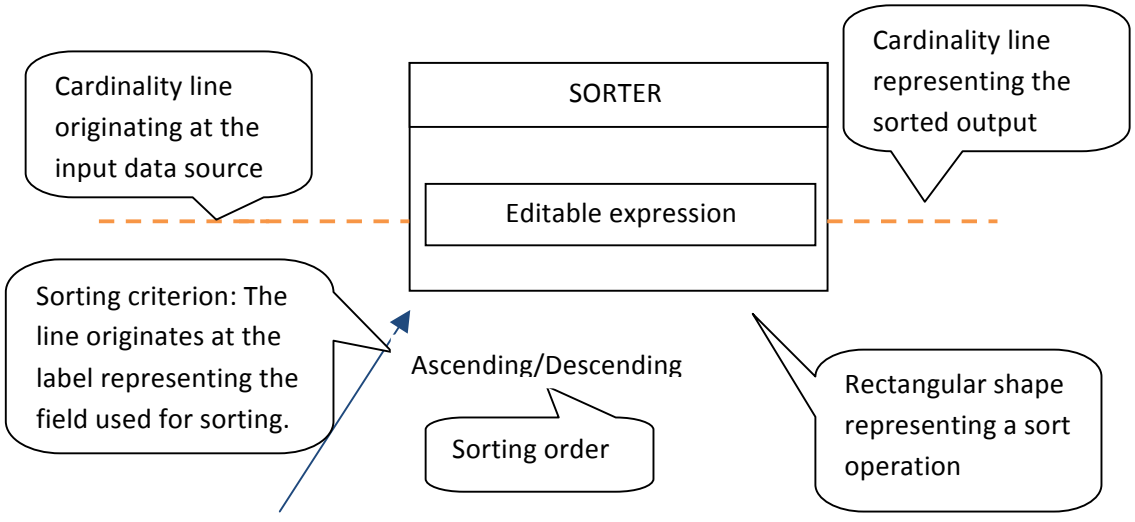


Figure 3.8: Graphical representation of a sort operation.

The attribute used as the sorting criterion is connected to the rectangular shape using a data flow connection. The connection is labeled with the sorting order. To understand this better, we illustrate the graphical representation of an example query written in SQL in Figure 3.9 below.

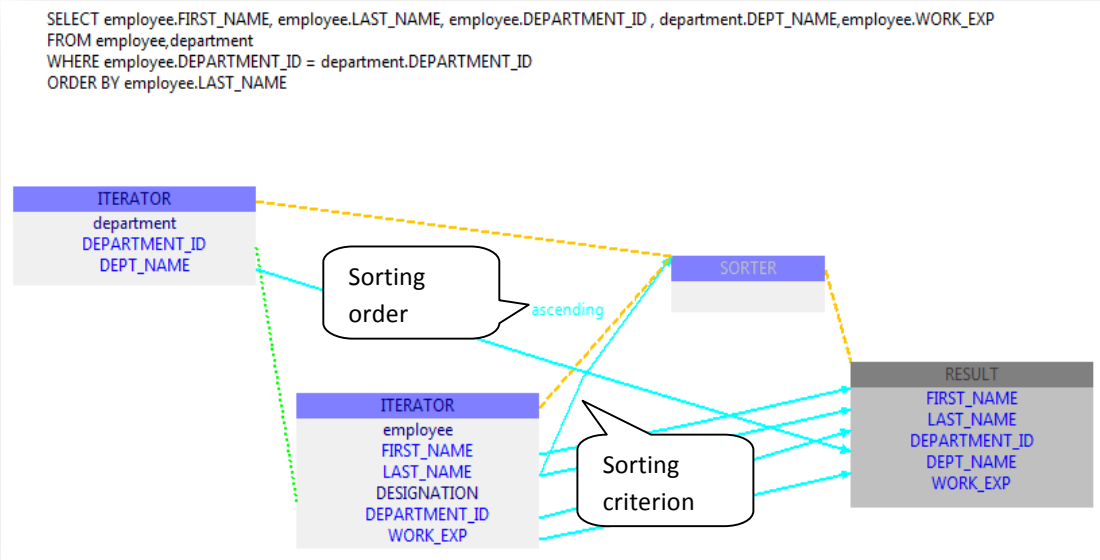


Figure 3.9 Graphical Representation of an example query involving a Sort operation

In the figure above, the query result is sorted on the basis of the "LAST_NAME" field. This is shown by a data-flow line connecting the corresponding label with the rectangular shape representing the sort operation.

5. Collecting

The data traversed during iteration may undergo subsequent operations like filtering or sorting, but are eventually collected by the collect operation to form the result of the query. The result of the query may consist of multiple data elements, but each of the resulting data elements follows a similar structure. For example, in case of a SQL query,

multiple rows might get returned, but each row has the same structure (a list of column values). The collect operation describes the common structure; with each data element forming the result that the query is expected to have. This is illustrated in Figure 3.10 using an example query.

Query Source

```
SELECT employee.FIRST_NAME, employee.LAST_NAME,
employee.DEPARTMENT_ID , department.DEPT_NAME
FROM employee,department
ORDER BY employee.LAST_NAME
```

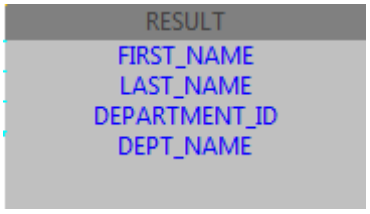


Figure 3.10: Graphical representation for the collect operation shows the structure of the result returned by the query.

The example query here returns specific fields from two SQL tables. The query's result fields are shown inside the rectangular shape representing the collect operation.

In order to illustrate the use of above described graphical representations, we take up a query example written in XQuery and describe its graphical representation.

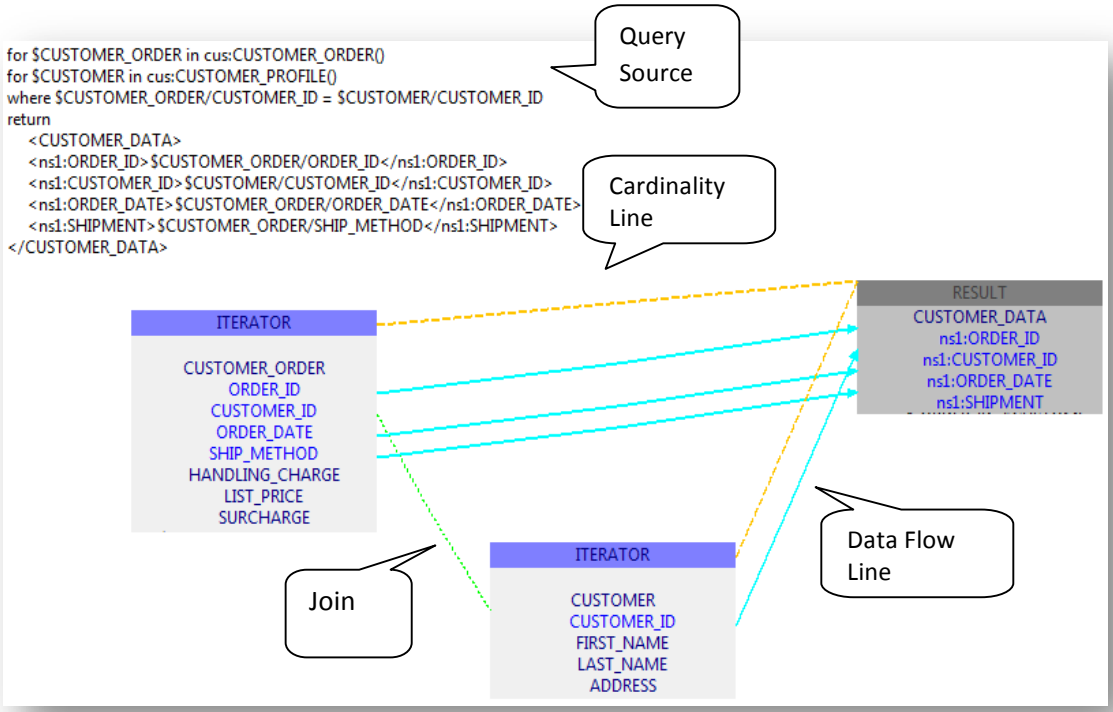


Figure 3.11: Use of different kinds of lines to depict relationships in a sample query written in XQuery.

In the query represented in Figure 3.11, we are iterating over customer orders and customer profiles and joining the data on a common attribute. The query uses the **for** clause to represent an iteration. A customer order is described by multiple attributes that are listed inside the rectangular shape representing the iteration. A similar rectangular shape represents the iteration over customer profiles. The result of the query consists of XML elements, each having specific attributes. These specific attributes are listed inside the rectangular shape representing the collection operation. For example, the first and last names, represented by the attributes FIRST_NAME and LAST_NAME respectively, are obtained from the data elements traversed during iterating the customer profiles. This is depicted using data-flow lines. The rectangular shapes representing iteration are connected to the rectangular shape representing collection using cardinality lines. A

customer order is joined with the corresponding customer profile using "CUSTOMER_ID" as the common attribute. This is represented by a join line.

CHAPTER 4

Query Analysis

The "Query Analysis" phase breaks a query in textual form into smaller components and maps each subpart into logical operations understood by XEditor. This phase involves the interaction of the language-agnostic XEditor core with a language-specific plug-in that is conversant with the grammar of the language. This chapter describes this interaction and how XEditor is fed with sufficient information about the query.

4.1 Analyzing a Query

At a high-level, a query consists of source code with optional comments embedded within. Source code consists of language-specific clauses, each signifying a high-level logical operation and perhaps defining variables. The analysis of a query begins by collecting the comments, if any, embedded inside the query. This is followed by breaking the query into language-specific clauses of the kinds that are well understood by XEditor. Information about any variables used in the query is also collected. This analysis provides XEditor with an ordered list of clauses together with a list of variables grouped by scope. XEditor must also be able to associate comments with the clauses against which they are written so that they can be shown as tool tips on the user interface. XEditor is aided by the language-specific plug-in in understanding the query.

4.2 Comment Handling

Comments start and end with language-specific string literals. For example, SQL comments start with '/*' and end with '*/'. The sequences of characters that occur

between these string literals form a comment. These string literals can also occur as part of a character constant, and in such a case, the enclosed characters are not considered to be part of a comment. Given the string literals that define the start and end of a comment and those that define the start and end of a character constant, comments can be extracted from the source code without any deeper understanding of the language grammar.

XEditor performs one pass over the source query to form a list of all comments, ordered by their occurrence. For every comment, XEditor records its precise location, that is, its starting/ending line and column numbers. Each comment is modeled as a Comment Clause, which is a pre-defined clause kind understood by XEditor. During this pass, the corresponding start/end string literals which mark the beginning/end of a comment and a character constant are obtained from the language specific plug-in.

The clauses contained inside a query are modeled as shapes in the graphical view. XEditor attempts to associate each comment with some clause in the query based upon the proximity of the comment to its neighboring clauses. A comment may be associated with either the preceding clause or the subsequent clause. For every clause, the precise location, that is, the starting/ending line and column numbers are known. This information is used to guess the associated clause for a comment from amongst its neighboring clauses. The comment then appears as a tool tip with the graphical representation of the associated clause. This is summarized and illustrated in Figure 4.1.

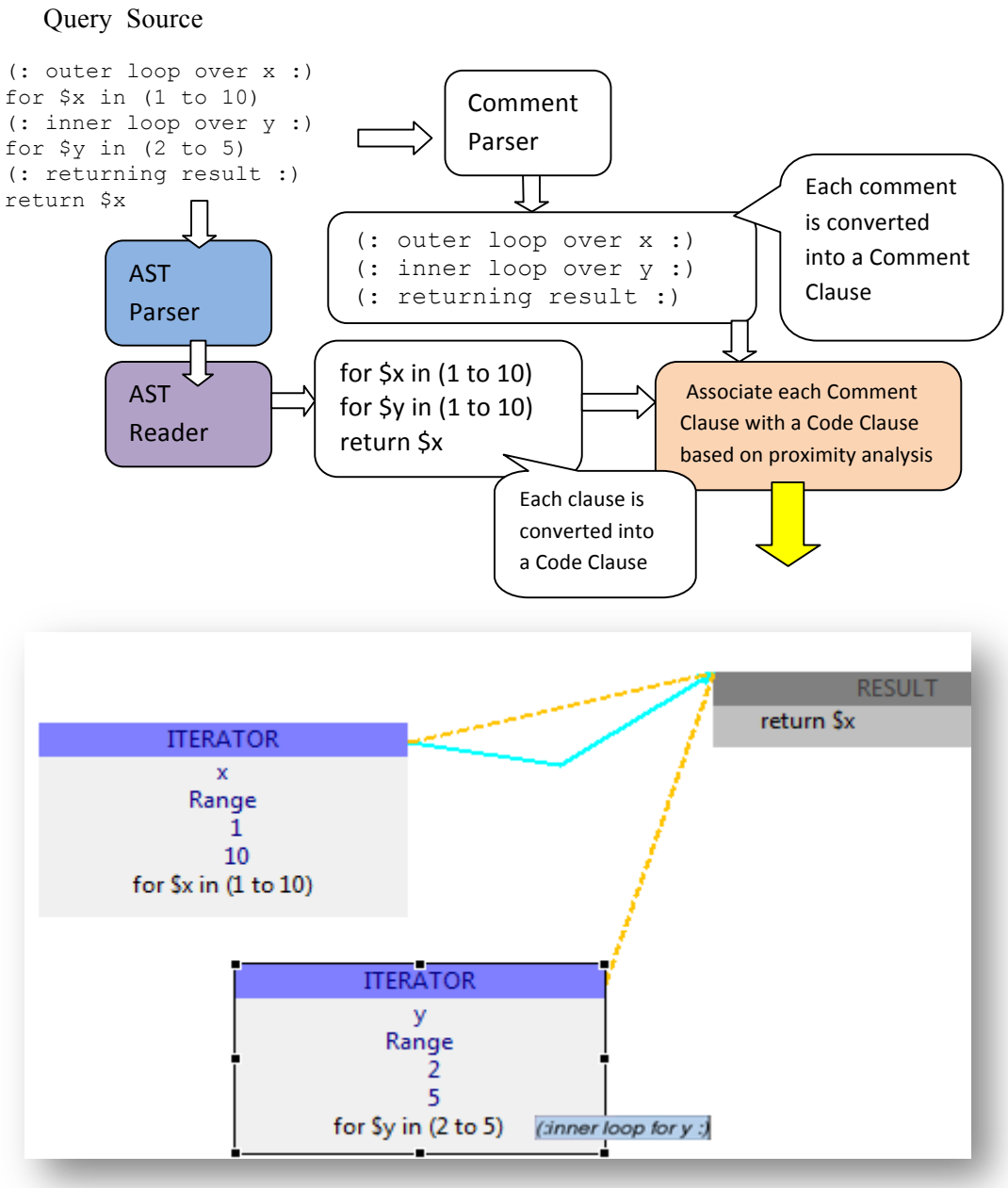


Figure 4.1 Handling comments in a query

Handling of comments in this fashion is a unique feature, in contrast to other visual query editors that we are aware of, as comments in the source query are usually just discarded in the graphical view.

4.3 Extracting Clauses and Variables

Extracting the clauses and variables of a query, grouped together by scope, is not in the purview of the core of XEditor since it does not understand the grammar of the language or the rules governing the beginning and end of a scope. XEditor relies on a language-specific plug-in that is conversant with the language grammar to parse the query into its AST (Abstract Syntax Tree).

For capturing information related to the clauses and variables in a query, XEditor maintains a symbol table. The language-specific plug-in builds the AST during its first pass over the query source. Before beginning the second pass, XEditor then instantiates a symbol table and passes a handle to it to the language-specific plug-in. The AST created in the initial pass is then traversed by the plug-in. During its traversal, the language-specific plug-in recognizes clause and variable occurrences as well as the beginnings and ends of scopes as per its language rules. Every clause has a kind based upon the high-level operation that it signifies, and each variable holds reference to the clause that defined it as well as information about other expressions required to construct its value.

The XEditor core is implemented in the Java programming language. It defines classes that represent each kind of clause and a class for representing a variable. These classes can be extended by the language-specific plug-in, if required. The language-specific plug-in is expected to instantiate these classes during traversal of the AST and to populate the fields inside with relevant information. The language-specific plug-in registers the objects with the symbol table using a Java API. Each registered object is assigned a unique ID generated by XEditor core. Figure 4.2 illustrates this phase.

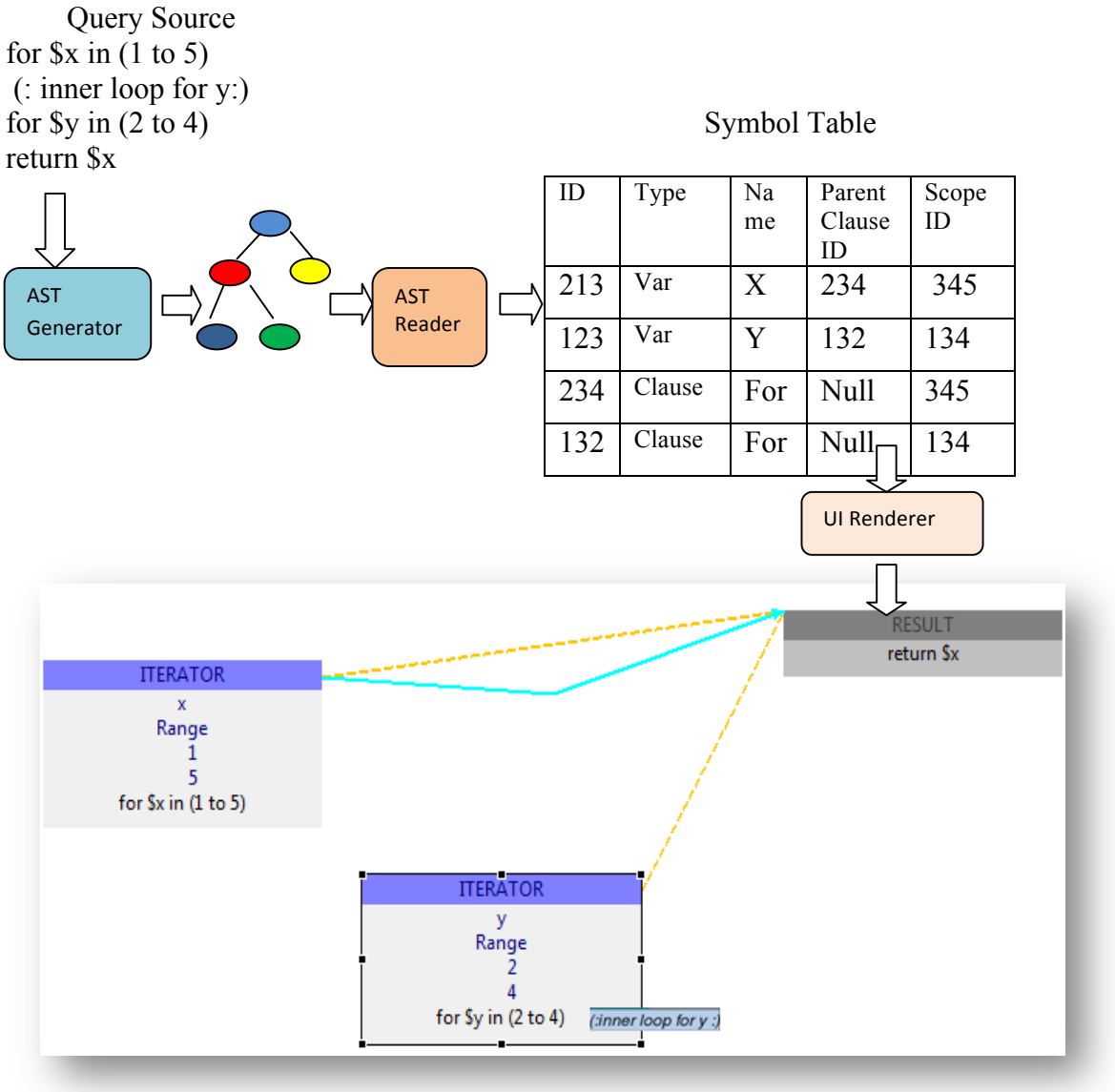


Figure 4.2: Population of symbol table as part of Query Analysis

4.4 Forming Dependency Relationships

As the language-specific plug-in traverses the AST, it populates the symbol table. Also the clauses collected are merged (on the basis of their start/end line and column numbers) with the clauses representing the comments collected by XEditor. In this way, XEditor

obtains the complete sequence of clauses, interleaved with clauses representing comments, in order of their occurrence in the query. In addition, XEditor also obtains a list of variables encountered during traversal of the AST. XEditor has now obtained what it requires from the language-specific plug-in. The task of the XEditor core is to analyze the symbol table and form dependency relationships that finally get rendered as connections in the graphical view.

4.4.1 Analysis of Symbol Table

The symbol table contains an entry for each clause or variable encountered during traversal of the AST. Each entry for a variable also has additional information about the terms used to define the value assigned to the variable. Each term may be an expression, a primitive term like an integer or a string, a function call, or a reference to another variable. Each term is treated as a contributing factor in building the value of the variable. This relationship is depicted as a data flow line between the shapes drawn to represent the term and the variable. If the term is a function call, the arguments are treated as a list of terms that each contribute in defining the value assigned to the variable. If a term is a reference to another variable, it is looked up in the symbol table to derive information about the clause which defines the variable. The resulting clause is then connected to the variable using a data-flow line. We consider an example query and show its graphical representation in Figure 4.3.

Query Source

```
for $CUSTOMER_ORDER in cus:CUSTOMER_ORDER()
let $TOTAL := $CUSTOMER_ORDER/HANDLING_CHARGE +
$CUSTOMER_ORDER/LIST_PRICE + fn:ceiling($CUSTOMER_ORDER/SURCHARGE)
return
<ns1:ELEC_ORDER>
 <ns1:CUSTOMER_ID>$CUSTOMER_ORDER/CUSTOMER_ID</ns1:CUSTOMER_ID>
 <ns1:ORDER_DATE>{fn:data($CUSTOMER_ORDER/ORDER_DATE)}</ns1:ORDER_DATE>
 <ns1:BILL>$TOTAL</ns1:BILL>
</ns1:ELEC_ORDER>
```

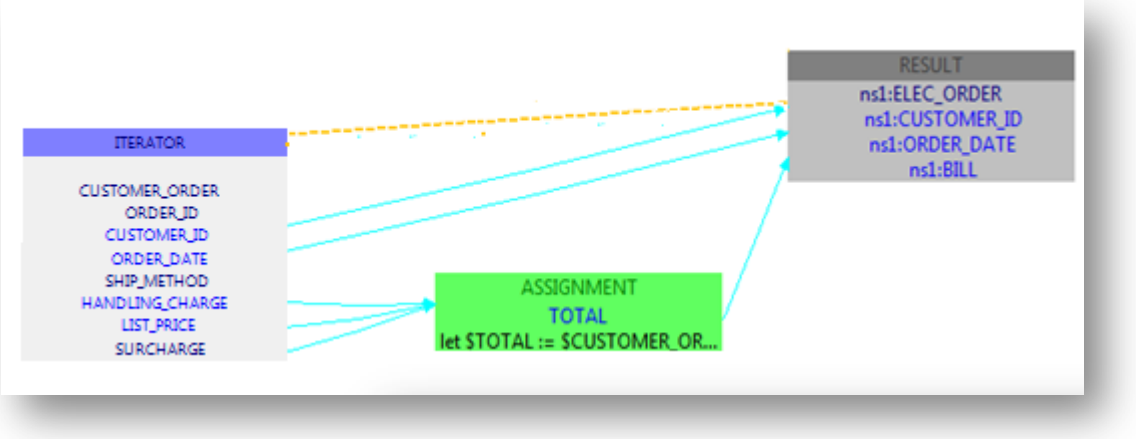


Figure 4.3: Data-flow lines derived from analysis of the symbol table.

In the example query, $TOTAL is defined using three other terms, namely HANDLING_CHARGE, LIST_PRICE, and SURCHARGE. This relationship is shown using data flow lines in the graphical representation. The value collected in the variable TOTAL is further assigned to the element <ns1:BILL>. This assignment also gets captured as a data-flow line.

As should be evident from this chapter, the task of breaking a query into smaller components and extracting information from them is divided between XEditor and the language-specific plug-in. XEditor takes care of comments and relies on the language-specific plug-in to provide an ordered list of clauses and variables grouped by the scopes in which they occur. Each clause object has a kind that signifies the high-level operation that it represents. The lists of clauses and variables generated by the language-specific

plug-in are then analyzed by XEditor to form the dependency relationships that are depicted as connections in the graphical view.

CHAPTER 5

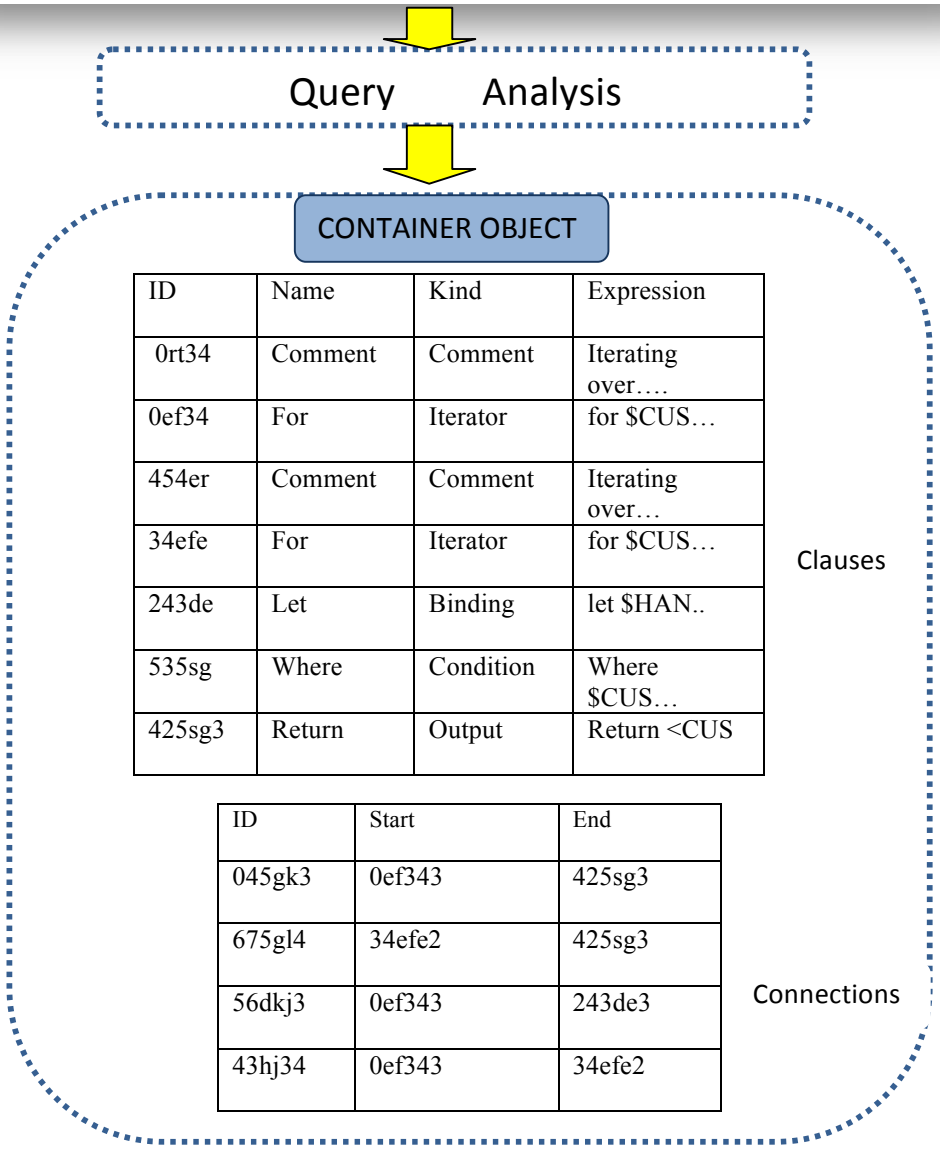
Query Rendering

The previous chapter explained the query analysis phase in which a query in textual form is broken down by a language-specific plug-in into an ordered list of clauses and variables, each grouped by the associated scope in which they occurred. The result is packed into what is referred to as a Container object, a serializeable representation that can be rendered into a graphical view. XEditor must decide on an initial layout of the query, in the graphical view, after which the layout gets saved as part of the Container. This chapter explains how a graphical view is formed out of a Container object.

5.1 Opening up the Container

As described in Chapter 3, Query Analysis is done by the back-end component of XEditor. XEditor also contains a User Interface component that interacts with the back-end component in terms of a Container object, which is the only element of information exchanged between the two components. Figure 5.1 illustrates the contents of a Container Object for a sample query.

```
(: iterating over customer orders:)
for $CUSTOMER_ORDER in cus:CUSTOMER_ORDER()
(: iterating over customer profiles:)
for $CUSTOMER_PROFILE  in cus:CUSTOMER_PROFILE()
let $HANDLING_CHARGE := $CUSTOMER_ORDER/HANDLING_CHARGE
where $CUSTOMER_ORDER/CUSTOMER_ID eq $CUSTOMER_PROFILE/CUSTOMER_ID
return
    <CUSTOMER_DATA>
      <ns1:CUSTOMER_ID>{fn:data($CUSTOMER_ORDER/CUSTOMER_ID)}</ns1:CUSTOMER_ID>
      <ns1:ORDER_DATE>$CUSTOMER_ORDER/ORDER_DATE</ns1:ORDER_DATE>
      <ns1:FIRST_NAME>$CUSTOMER_PROFILE/FIRST_NAME</ns1:FIRST_NAME>
      <ns1:LAST_NAME>$CUSTOMER_PROFILE/LAST_NAME</ns1:LAST_NAME>
    </CUSTOMER_DATA>
```



ID	Name	Kind	Expression
0rt34	Comment	Comment	Iterating over….
0ef34	For	Iterator	for $CUS…
454er	Comment	Comment	Iterating over…
34efe	For	Iterator	for $CUS…
243de	Let	Binding	let $HAN..
535sg	Where	Condition	Where $CUS…
425sg3	Return	Output	Return <CUS

ID	Start	End
045gk3	0ef343	425sg3
675gl4	34efe2	425sg3
56dkj3	0ef343	243de3
43hj34	0ef343	34efe2

Figure 5.1: Example showing a Container Object formed after analysis of a sample query written in XQuery.

The sequence of steps taken after the Container object is passed on to the Query Rendering phase is illustrated in Figure 5.2. As the UI component scans the contents of the Container object, it discovers clause definitions. By reading the kind information inside each clause, the UI component decides which shape to use to represent it. When the scan is finished, the UI component has a list of shapes that need to be drawn and it also knows which of them need to be connected with which other shapes using the various kinds of lines.

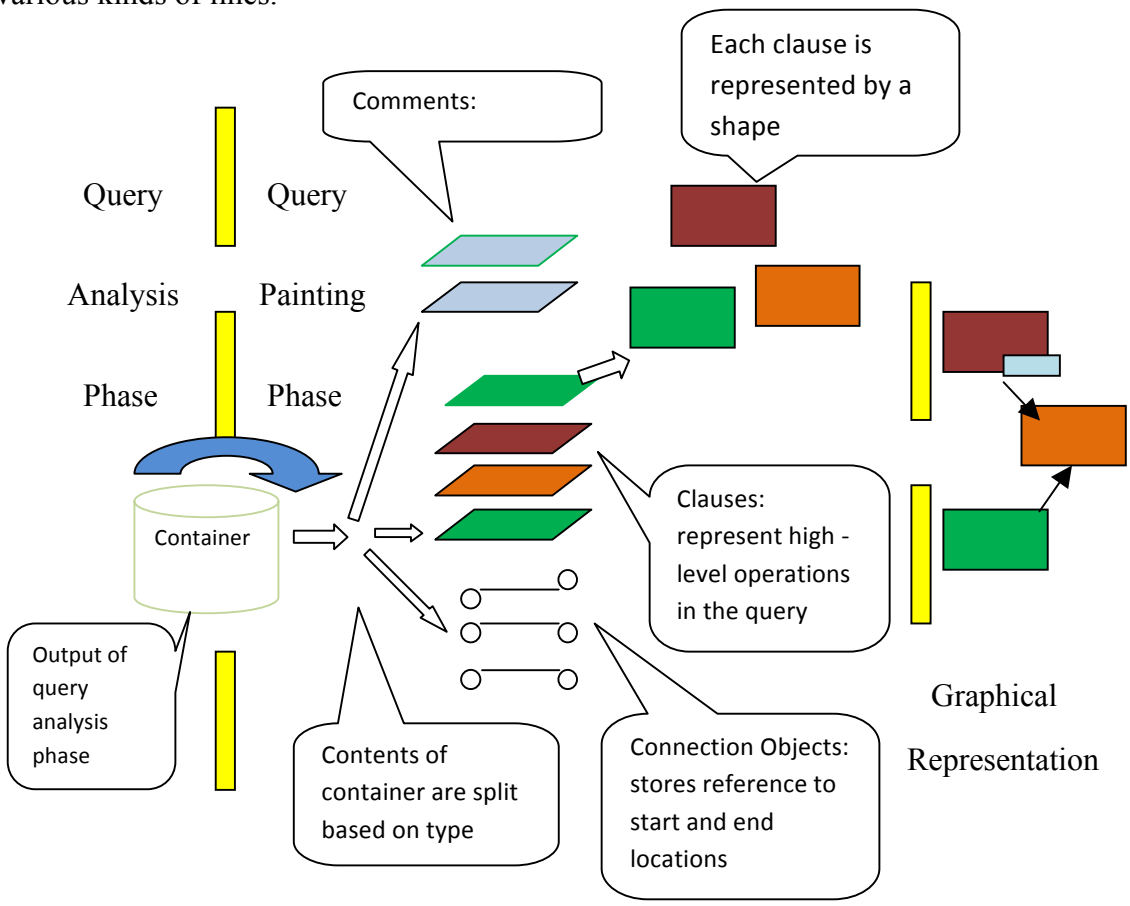


Figure 5.2: Output of Query Analysis phase is passed as input to the Query Rendering phase, where the output – a Container object - is analyzed to form a graphical representation

Each clause has a unique ID that gets assigned when the clause is registered in the symbol table. The UI component uses the unique ID to maintain a correlation between a shape and its corresponding clause so that when the user interacts with the shapes in the

graphical view, XEditor knows what part of the query is being modified. Laying out a query on the canvas does not involve any language-specific plug-in and is therefore done in a homogeneous manner independent of the language of the query. XEditor uses the Eclipse Graphical Editing Framework (GEF), which facilitates creating a user interface while being model-agnostic. GEF is based upon the model-view-controller pattern (MVC) and uses Draw2d to put shapes on the canvas. The following sections gives a brief description of GEF and Draw2d and how XEditor utilizes them.

5.3 GEF and Draw2d

The Graphical Editing Framework (GEF) was developed for the Eclipse platform. GEF follows the model-view-controller pattern. Here, the model represents the application's data model. The view represents the layer of user-interface widgets that get displayed. The intermediate controller coordinates between the view and the model such that changes in the view are reflected onto the underlying model and vice versa. Everything is in the model, and the model is the only thing that is persisted and restored. The application thus stores all important data in the model. During the course of editing, undo, and redo, the model is the only thing that endures.

For every element in the model, there is a shape that represents it on the user interface. The view consists of shapes that are essentially Draw2d objects. Draw2d provides a bag of shapes that form the building blocks of the user interface. Controllers bridge between the view and model. Each controller, or EditPart as they are called in Draw2d, is responsible both for mapping the model to its view and for applying changes to the model. The EditPart also observes the model and will update the view to reflect

any changes in the model's state. In order to render a query, it is important to form the data model as well as to instantiate appropriate shapes that correspond to the elements in the data model.

5.4 Forming the Data Model of a Query

The model layer in the MVC pattern followed by GEF is essentially hierarchical in nature. For every element in the data model there exists an element in the view component that forms the graphical representation of the element. For rendering, the model is traversed in a preorder manner, each node in the hierarchy is visited, and the corresponding shape is instantiated. Thus, a hierarchy of shapes, similar to the hierarchy existing in the model layer, has been built at the end of the traversal. The shapes are then rendered in a bottom-up manner, rendering children first and then painting the encompassing parent.

It is essential to form the hierarchical data model in order to use GEF. As a query is broken down into its constituent clauses, a shape gets associated with the clause to form the graphical representation of the clause. Each clause becomes a node in the GEF data model. A clause definition is further broken down into elements that build up the clause expression. These elements become children of the node representing the clause. The leaf level contains primitive types in the language that cannot be broken down further. The data model is then passed on to the GEF layer, which takes care of laying it out on the canvas.

CHAPTER 6

Query Editing

A query may be edited from either the graphical or textual view, and changes made in either view must be synchronized and reflected in the other view. In addition, the layout of shapes in the user interface or the alignment of source code in the textual view should not be disturbed when modifications are made through the other view. If a modification in either view yields a syntactically incorrect query, XEditor should still be able to provide a graphical view, marking the region of error, so that the user can make corrections within the graphical view. This chapter gives a detailed description of how XEditor meets these requirements in its handling of the modifications made to the query in either view.

6.1 Editing a Query in the Graphical View

In the graphical view, a query is laid out in the form of shapes and lines that connect the shapes. Considering a query to be a sequence of clauses, a modification to the query can mean changing the existing definitions of clauses, adding a clause, or deleting a clause. This section describes how XEditor handles the modification of an existing clause or the deletion a clause via the user interface.

6.1.1 Changing a Clause Definition via Text Editing

Recall from Chapter 2 that a clause has an associated expression, as per the language grammar rules. Changing a clause definition is equivalent to modifying the associated expression. In the graphical view, the expression is shown as a non-editable label inside

the rectangular shape representing the clause. Double clicking on the label changes it to an editable text box, allowing user to edit the expression. This is illustrated below in Figure 6.1.

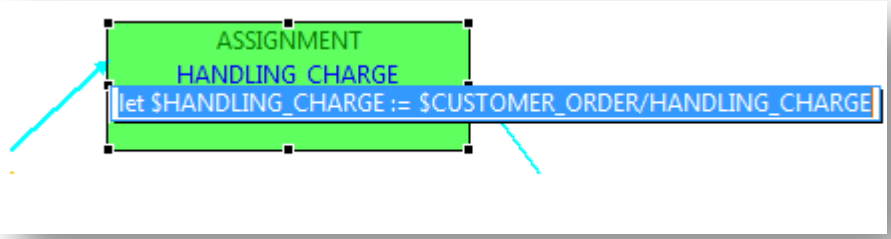


Figure 6.1 Editing a Clause Expression

When the user finishes making a change in this manner, the modified expression replaces the previous expression and a new query is formed by XEditor. The section of the query following the modified section is adjusted according to the length of the modified expression in order to maintain formatting. This is illustrated in Figure 6.2.

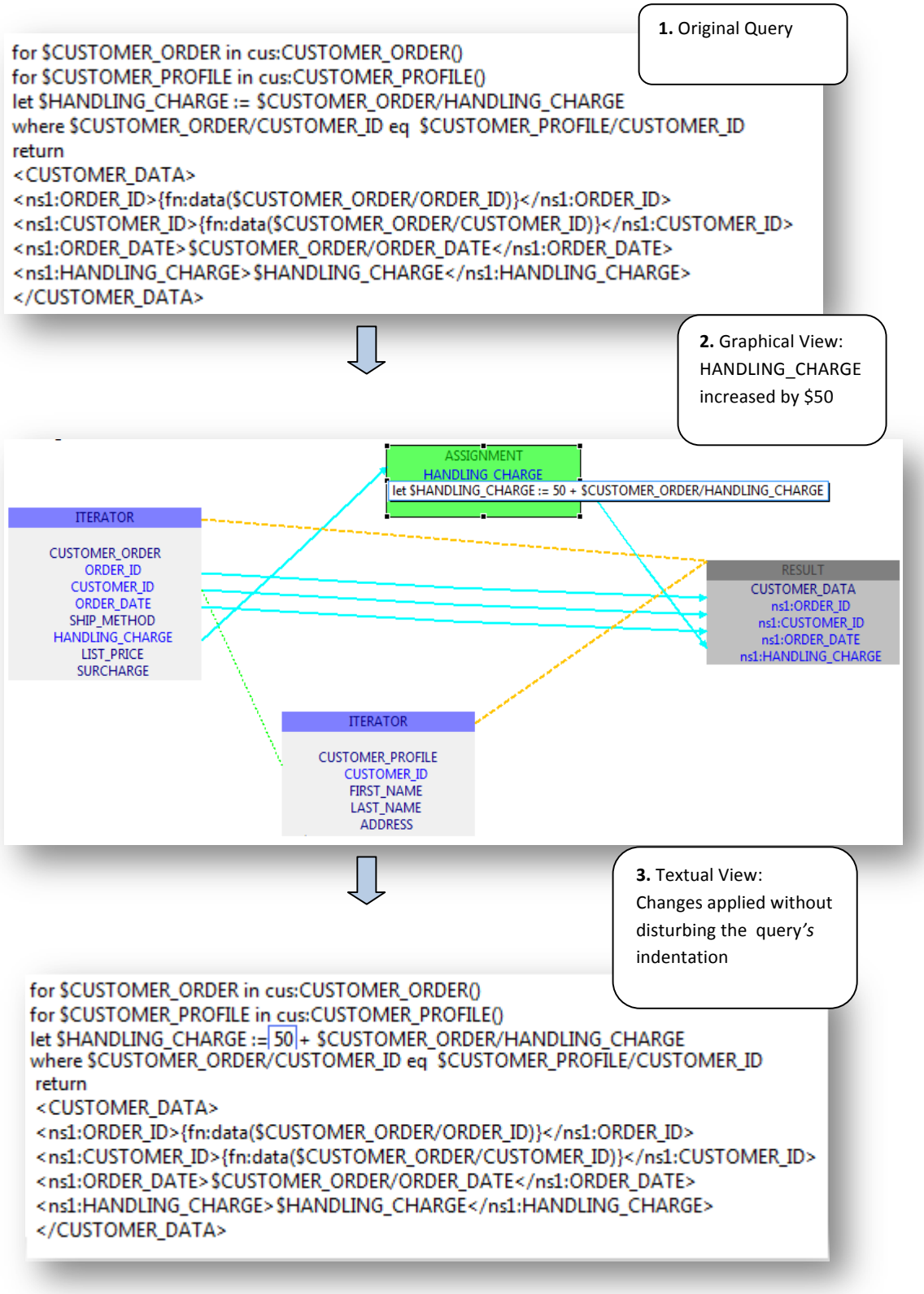


Figure 6.2: Figure showing an editing session. An expression is changed in the graphical view and the changes get applied in the textual view while preserving the original formatting.

When the expression associated with a clause is changed via an editable textbox, the modified expression may cause the overall query to become syntactically incorrect. Since XEditor does not understand the grammar rules of the language, it relies on the language-specific pluggable component to validate the query. The modified query is submitted to the language-specific pluggable component for generation of the AST. If this step is unsuccessful, it is indicative of an error in the query. This is flagged to XEditor, which must now pass on an error message to the user. XEditor does not force the user to switch to the textual view to understand what went wrong. Instead, the shape that represented the edited clause is marked in red to indicate an error. This is illustrated in Figure 6.3.

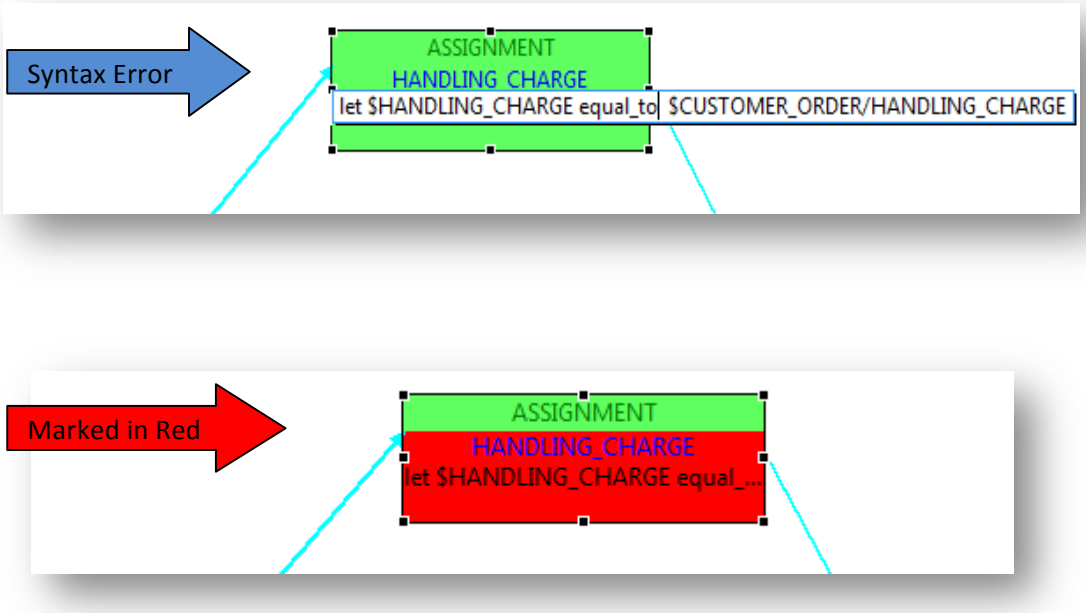


Figure 6.3: Error Handling: The user makes a change that is not compliant with the syntax rules of the language. XEditor receives input from the language-specific plug-in and indicates the error to the user.

In the event of an error, the user can edit the clause expression again and re-submit the changes. The same sequence of validation is then repeated, and the shape representing the edited clause remains colored as red until the erroneous expression has been corrected.

Once the modified query is valid, it then needs to be re-drawn on the canvas. The new query may result in new shapes needing to be drawn or older shapes needing to be deleted. Alternatively, a change may require that additional connections be drawn between two shapes or that existing connections be deleted. In such cases, the current positioning of the other shapes (those not affected by the change) in the user interface ideally should not change, as the shapes may be at their specific on-canvas locations because the user wanted them laid out that way.

When XEditor applies changes on the user interface, it tries to cause minimal disturbance in the layout of the new query with respect to the earlier version of the query. XEditor does not understand the modifications that are done to an existing clause expression. When the changed query is submitted to the language-specific pluggable component, provided that the modification was syntactically correct, what is retrieved is an ordered list of clauses and variables. At this stage, XEditor has not overridden its internal symbol table, so the internal state of XEditor still has the list of clauses associated with the query before the change.

XEditor traverses the old and the new lists to determine the clauses that are common to both, clauses that are present only in the old list, and newly added clauses that are present in the new list. The shapes related to the clauses that are common to both need not be changed and are retained on the user interface at their previous locations.

Shapes corresponding to clauses that are only present in the old list are removed. When a shape is removed, the associated connections, originating or terminating at the shape are also removed. The shapes for representing the newly added clauses are then generated fresh and laid out on the user interface alongside the previously retained shapes.

6.1.2 Deleting a Clause

Recall from Chapter 4 that XEditor considers a query to be a sequence of clauses, each being a representative of a high-level operation. Deleting a clause thus means removing the high-level operation from the query. As an example, consider the query given below.

Query Source

```
for $CUSTOMER_ORDER in cus:CUSTOMER_ORDER()
order by $CUSTOMER_ORDER/CUSTOMER_ID
return
<CUSTOMER_DATA>
       <ns1:ORDER_ID>{fn:data($CUSTOMER_ORDER/ORDER_ID)}</ns1:ORDER_ID>
       <ns1:CUSTOMER_ID>{fn:data($CUSTOMER_ORDER/CUSTOMER_ID)}</ns1:CUSTOMER_ID>
       <ns1:ORDER_DATE>$CUSTOMER_ORDER/ORDER_DATE</ns1:ORDER_DATE>
       <ns1:SHIPMENT>$CUSTOMER_ORDER/SHIP_METHOD</ns1:SHIPMENT>
</CUSTOMER_DATA>
```

The query returns a list of customer orders sorted by the attribute CUSTOMER_ID. The user may not require a sorted result, and may wish to remove the sort operation. Right clicking on the rectangular shape representing the sort operation gives the user an option to delete the operation. This is illustrated in Figure 6.4.

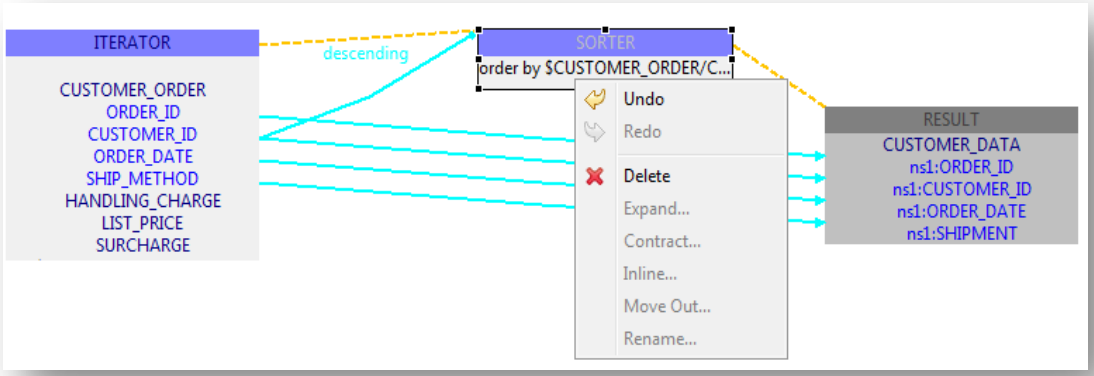


Figure 6.4: About to delete a clause from the graphical view.

If the user indeed chooses to delete the sort, changes need to be applied in both the graphical and textual views. Recall from Chapter 4 that for each clause encountered in a query, XEditor stores the corresponding start/end line and column numbers. When a clause is deleted from the graphical view, XEditor removes the portion of text enclosed between the start/end line and column numbers associated with the clause. XEditor removes the associated shape from the graphical view and re-routes any connecting lines. In the example query described above, removing the rectangular shape representing the sort operation also requires re-routing the cardinality lines. The result of the deletion is shown in Figure 6.5.

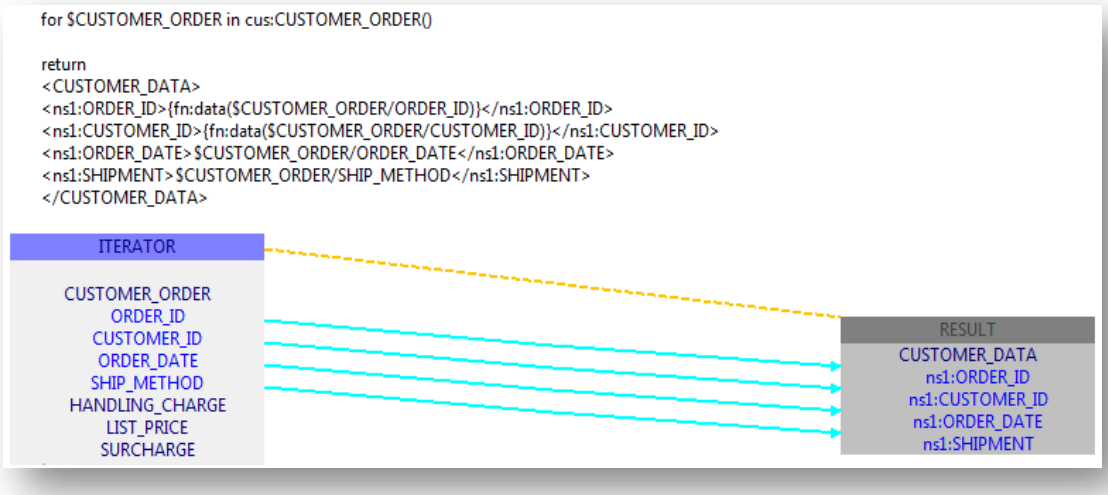


Figure 6.5: Result after deleting a clause

Deleting a high-level operation can become non-trivial in some cases. As an example, Iteration and Binding introduce new variables, and these may be referred to from other parts of the query. In case these variables are not being referred to, the clauses can be deleted in a manner similar to deletion of a sort operation. Otherwise, deleting the corresponding clause may leave some variable references undefined. In such a case, XEditor does not support the deletion of such high-level operations from the graphical view.

6.1.3 Forming or Deleting Connections

Connections are also representative of high-level operations done in the query. For example, a connection that connects two variables may indicate an equi-join involving the variables or may indicate the use of one variable in defining the other. If the user is trying to form an equi-join between two attributes, the change may result in the

introduction of a new clause. The exact expression for the new clause will depend upon the language grammar and hence is beyond the understanding of the XEditor core.

To understand the manifestation of such a user action, XEditor again relies on the language-specific pluggable component. XEditor plays a part in capturing information about the user's action, including the kind of connection the user is trying to make graphically, or attempting to delete, as well as the start and end points of the connection. On the user interface, the start/end points are merely shapes. Recall from Chapter 3 that depending on its kind, a clause is represented by one or more shapes. Each of these shapes has a unique ID which is same as the ID that was assigned to the corresponding clause at the time of insertion into the symbol table. This helps XEditor find the corresponding clause(s) that is(are) being touched. XEditor identifies the clause(s) from amongst the list of ordered clauses that it obtained earlier from the language-specific plug-in. Information that includes the type of connection, affected clause(s) and the complete list of clauses is then submitted to the language-specific component.

Upon scanning this information, the language-specific component is expected to return the modified query. A language-specific component that is not advanced enough to scan the information returned in response to a graphical change has the option of returning a null response. XEditor interprets the null response as the inability of the language-specific component to generate code for the requested graphical change. In such a case, as a fallback, XEditor opens up a text box containing the complete query. The user then must explicitly provide code that expresses the change that he/she desires. Of course a well-written language-specific component will respond with the modified query. Deleting a connection follows a similar strategy; the language-specific component

responds by returning the modified query in text form, which is then analyzed and converted into an equivalent graphical representation.

CHAPTER 7

Using XEditor for Multiple Languages

To validate the XEditor design decisions and the architecture, we have prototyped language-specific components for two languages, namely XQuery and a subset of SQL. Each of these languages' pluggable components implemented the prescribed interfaces so that it could neatly fit into and work with the core architecture. This exercise demonstrated that XEditor can indeed provide multi-lingual support and form graphical editable views of queries written in both languages.

This chapter describes how these pluggable components were built and covers example queries to show the resulting editing environments.

7.1 Interaction between XEditor and Language-Specific Plug-Ins

Recall from Chapter 2 that the pluggable component for a language must perform three main functions, which are:

1. Parse source code and form the equivalent AST.
2. Traverse the AST and identify clauses and variables together with the scopes in which they occur.
3. Generate source code for a given action taken in the graphical view.

The pluggable component thus has three sub-components. Each of the functionalities above is provided by one sub-component. A prerequisite for providing these

functionalities is to form a mapping between the operations expressible in the target query language and the high- level operations understood by XEditor.

7.2 Using XEditor for XQuery

XML is a versatile markup language that is capable of labeling the information content of diverse data sources, including structured and semi-structured documents, relational databases, and object repositories. A query language for XML can express queries across all these kinds of data, whether physically stored in XML or viewed as XML via middleware [4]. XQuery is such a language.

XQuery allows users to write queries in a similar way to the familiar SQL approach. Its equivalent of SQL's SELECT expression is called the FLWOR expression [11]. The name FLWOR, pronounced "flower", is suggested by the keywords **for**, **let**, **where**, **order by**, and **return**. The **for** and **let** clauses in a FLWOR expression generate an ordered sequence of tuples of bound variables called the tuple stream. The **where** clause is optional and serves to filter the tuple stream, retaining some tuples and discarding others. The optional **order by** clause can be used to reorder the tuple stream. The **return** clause constructs the result of the FLWOR expression. The **return** clause structures the query result and is evaluated once for every tuple in the tuple stream, after filtering by the **where** clause, using the variable bindings in the respective tuples. The result of the FLWOR expression is an ordered sequence containing the concatenated results of these evaluations. Table 7.1 briefly describes the components of a FLWOR expression and shows the high-level operations they each correspond to in XEditor.

No	FLWOR Expression Component	Description	High-Level Operation
1	for	The **for** clause generates an ordered sequence of tuples of bound variables, called the tuple stream	Iteration
2	let	The **let** clause binds each variable to the result of its associated expression, without iteration.	Binding
3	where	The **where** clause serves to filter the tuple stream	Filtering
4	order by	The **order by** clause can be used to reorder the tuple stream in ascending or descending manner.	Sorting
5	Return	The **return** clause constructs the result of the FLWOR expression.	Collecting

Table 7.1: Similarity between components of a FLWOR expression and the high-level operations understood by XEditor.

The language-specific component for XQuery transforms a given query into its equivalent AST. The AST is then traversed and, as any of the clauses mentioned in Table 7.1 is encountered, appropriate information is added to the symbol table. As described in Chapters 4 and 5, once appropriate information has been put into the symbol table, a Container object is formed and subsequently analyzed in order to form a graphical representation of the query.

Figure 7.1 shows the XQuery editing environment provided by XEditor. XEditor with a plug-in for XQuery works as a two-way graphical editing tool for the XQuery language and competes functionally with BEA's XQE (a tool written specifically for

XQuery). Table 7.2 summarizes the key differences in features provided by the current XEditor-based XQuery editor prototype and XQE.

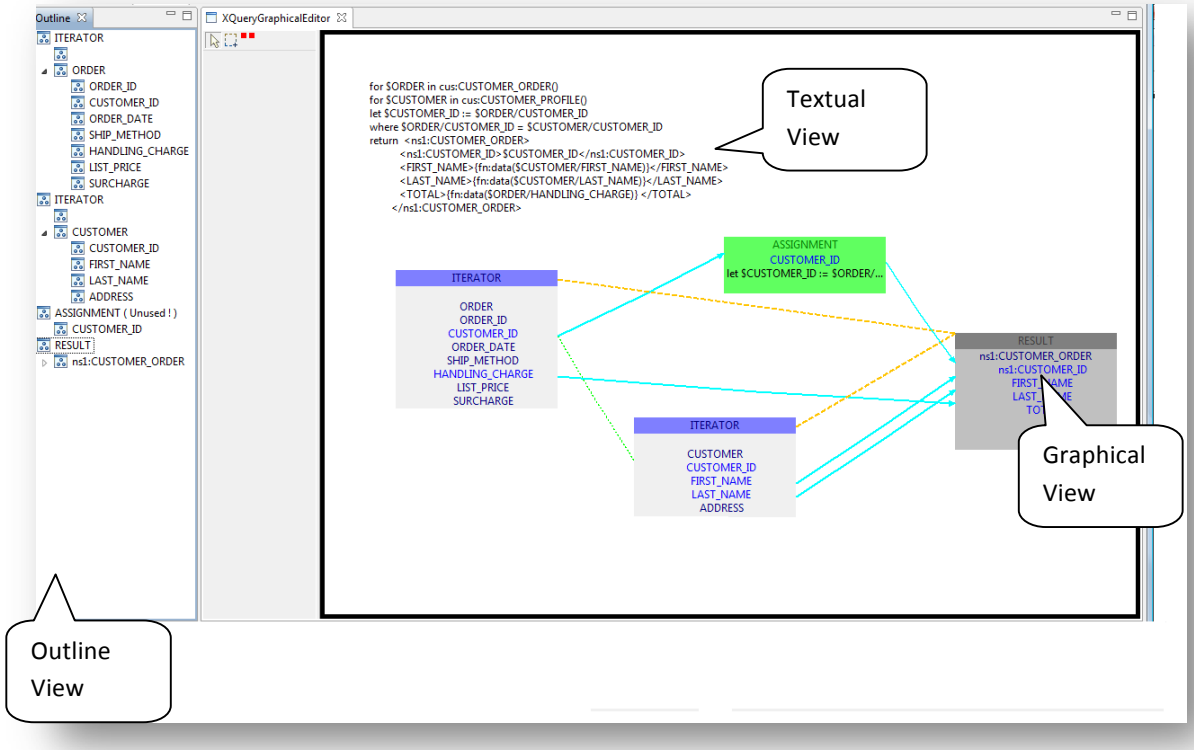


Figure 7.1: The editing environment provided by XEditor

No	Feature	BEA's XQE	XEditor	Comments
1	Editing clause expressions in graphical view	✓	✓	XQE provides partial editing of clause expressions. It provides an expression editor to edit filter/join conditions but does not allow editing of other kind of clauses. XEditor follows a generic approach whereby double clicking on the corresponding shape in the graphical view makes the associated expression editable.
2	Parsing of Prolog to form function definitions and environment variables	✓		XEditor does not yet parse the declaration of user defined functions.
3	Enumerating list of operators and library functions (for XQuery)	✓		The XEditor will expose an interface for collecting metadata about the language operators and functions.
4	Polished User Interface	✓		The XEditor user interface is evolving and shall be polished over time.
5	Handling of full XQuery language	✓		The language-specific pluggable component for XQuery is evolving to incorporate complete XQuery.
6	Support for multiple query languages		✓	Extensible to multiple languages by writing a language-specific pluggable component.
7	Handling of embedded comments in the query		✓	Comments appear as tool tips with the shape representing the associated section of the query
8	Handling of errors in graphical view		✓	The region on XEditor user interface containing the error is marked red, which can be corrected in the graphical view. XQE displays an error message and forces a switch to the text view.
9	Providing different views varying in complexity for a single query		✓	User can choose to view or hide the nested components of a query
10	Outline view of the query		✓	Outline view of the query is auto – constructed.

Table 7.2: Key differences in features provided by XEditor and XQE

7.2.1 Sample Queries in XQuery

In this section, we provide a few example queries written in XQuery and illustrate their graphical representations in XEditor.

7.2.1.1 Inner Join

We describe here a query involving an inner join. The query intends to return a list containing customer names along with the date when they placed an order. Referring to the query source in Figure 7.2, the query involves iteration over a list of customer orders and customer profiles. Customer orders and customer profiles are joined using a common attribute -"CUSTOMER_ID". For each customer, the name is constructed by concatenating the attributes "FIRST_NAME" and "LAST_NAME".

Query Source

```
for $CUSTOMER_ORDER in cus:CUSTOMER_ORDER()
for $CUSTOMER_PROFILE  in cus:CUSTOMER_PROFILE()
where $CUSTOMER_ORDER/CUSTOMER_ID = $CUSTOMER_PROFILE/CUSTOMER_ID
return
 <CUSTOMER_DATA>
      <ns1:NAME>
           {fn:data($CUSTOMER_PROFILE/FIRST_NAME) +
           fn:data($CUSTOMER_PROFILE/LAST_NAME)}
     </ns1:NAME>
     <ns1:ORDER_DATE>{fn:data($CUSTOMER_ORDER/ORDER_DATE)}</ns1:ORDER_DATE>
 </CUSTOMER_DATA>
```

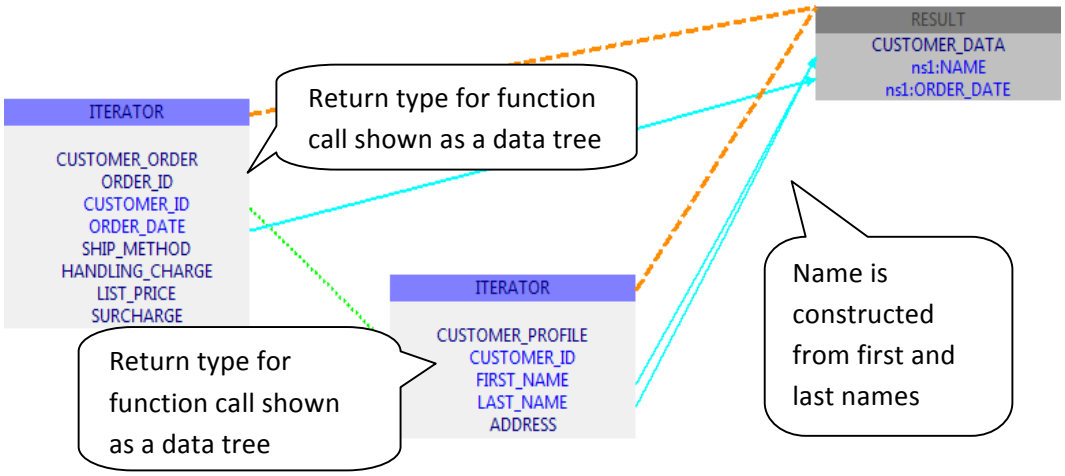


Figure 7.2 Inner Join: A query involving function calls and XML schemas and its graphical representation

Figure 7.2 shows the graphical representation of the query. The query uses user-defined functions to iterate through a list of customer orders and customer profiles. The return type of each user-defined function is shows as a schema tree. The inner join between the customer orders and customer profiles is represented using a join connection.

7.2.1.2 Outer Join

We describe here a query involving an outer join. The query source is given in Figure 7.3. The query has an outer loop iterating through customer profiles. The query uses a user-defined function to obtain a list of customer profiles. For every customer profile, a list of orders placed by the customer is built by a subquery. The subquery constructs the list by iterating through a list of orders and correlating them with the customer profiles using CUSTOMER_ID as the common attribute.

The graphical representation illustrated in Figure 7.3 depicts traversal over customer profiles using an iterator box. The traversal over customer orders happens inside a subquery and is depicted using an iterator box. A join line connects the common attribute- CUSTOMER_ID present inside the two iterator boxes and (with nesting) is symbolic of an outer-join between the two data sets. The result of the outer-join is sorted in a descending manner on the basis of the order amount and is represented using a rectangular box. For each customer, the result consists of the corresponding attributes - FIRST_NAME and LAST_NAME, followed by a list of orders placed by the customer. The result is repeated for each customer traversed during iteration. A part of the result – namely the attributes FIRST_NAME and LAST_NAME - are driven by iteration over customer profiles. This is graphically depicted using a cardinality line that connects the

iterator box with the result box. The remaining part of the result is the sorted list of orders placed by the corresponding customer and is driven by iteration over the list of customer orders and subsequent sorting of matched records. The sorted list of orders is graphically represented inside the result box and is connected to the box representing the sort operation using a cardinality line. The cardinality lines indicate the respective parts of the result that are driven by each of the iteration or the sort operation. Thus the cardinality line originating from the iterator box representing sorting over customer orders terminates at the graphical representation of the sorted output inside the result box.

```
Query Source

for $CUSTOMER in cus:CUSTOMER_PROFILE()
return
<tns:CUSTOMER>
      <FIRST_NAME>{fn:data($CUSTOMER/FIRST_NAME)}</FIRST_NAME>
      <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>
      {
      for $ORDER in cus:CUSTOMER_ORDER()
      where $ORDER/CUSTOMER_ID eq $CUSTOMER/CUSTOMER_ID
      order by $ORDER/TOTAL
      return <ORDER>
             <OID>{fn:data($ORDER/ORDER_ID)}</OID>
             </ORDER>
      }
</tns:CUSTOMER>
```

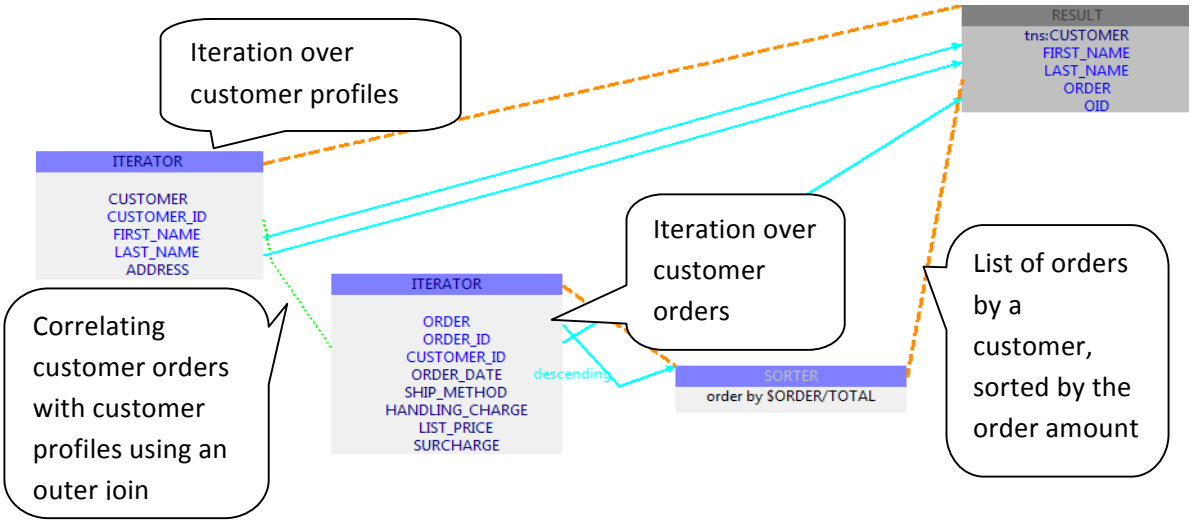


Figure 7.3: Outer Join: A query containing an outer-join and its graphical representation.

7.2.1.3 Nested Expressions

XEditor supports nested expressions. A nested expression can be shown in XEditor in multiple ways based upon the user's preference. For an example of a query containing a nested expression, consider the query presented in Figure 7.3. The query contains a **return** clause that contains a nested FLWOR expression. The graphical represenation in Figure 7.3 is an unpacked representation where the graphcial shapes representing the nested FLWOR expression are drawn alongside the shapes representing other clauses. In an alternate representation, these graphical shapes can also be drawn inside the graphical represenation of the return clause that contains the nested FLWOR expression. This is a packed representation and requires fewer shapes to represent the query. Figure 7.4 illustrates the packed representation of the query.

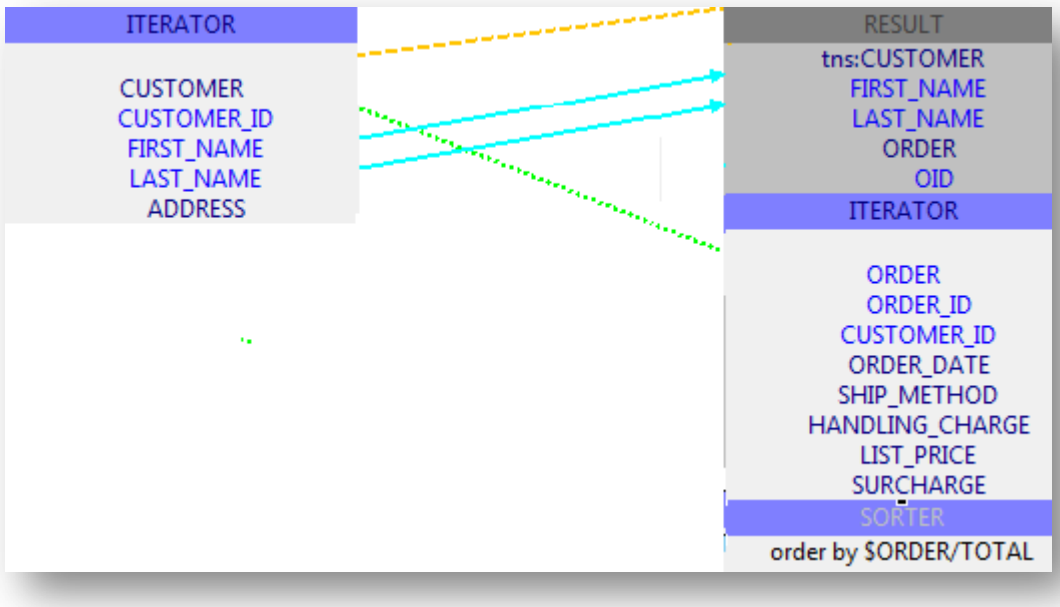


Figure 7.4: Packed representation of a nested expression

In a packed representation, any connections that exist between graphical components representing the nested expression remain hidden. Right clicking a packed representation

gives the user an option to switch to an unpacked representation and vice-versa. This is illustrated in Figure 7.6.

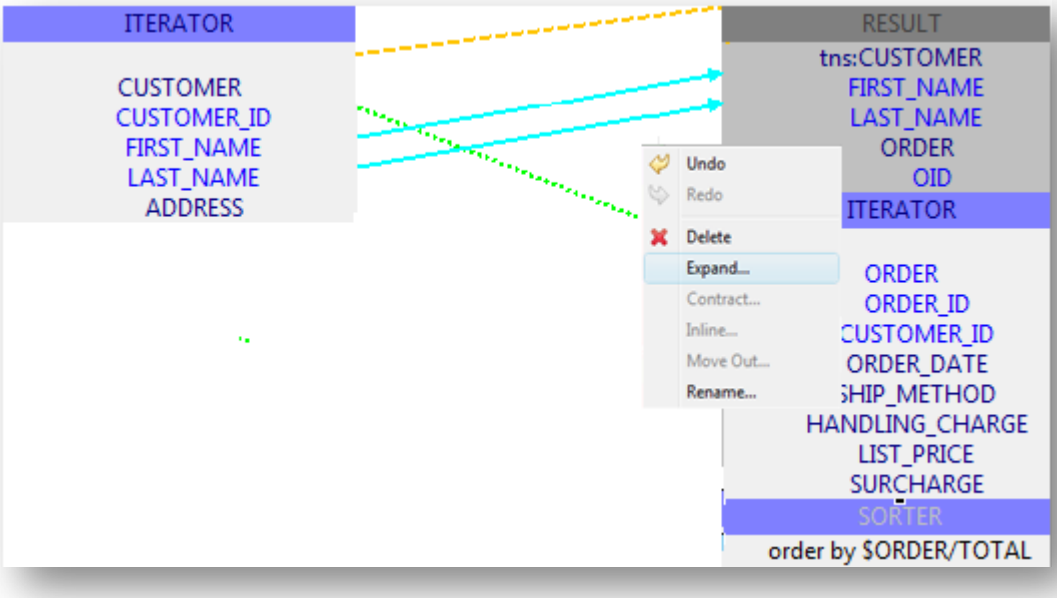


Figure 7.5: Switching between packed and unpacked representations

For the representation illustrated in Figure 7.5, choosing the 'Expand' option would result in the graphical represenation illustrated in Figure 7.3.

7.2.1.4 Order By

We next describe a query involving a sort operation. Referring to the query source in Figure 7.6, the query involves iteration over a list of customer orders, sorted by the attribute "CUSTOMER_ID", and returns specific attributes for each order.

Query Source

```
for $CUSTOMER_ORDER in cus:CUSTOMER_ORDER()
order by $CUSTOMER_ORDER/CUSTOMER_ID
return
    <CUSTOMER_DATA>
    <ns1:ORDER_ID>{fn:data($CUSTOMER_ORDER/ORDER_ID)}</ns1:ORDER_ID>
    <ns1:CUSTOMER_ID>{fn:data($CUSTOMER_ORDER/CUSTOMER_ID)}</ns1:CUSTOMER_ID>
    <ns1:ORDER_DATE>$CUSTOMER_ORDER/ORDER_DATE</ns1:ORDER_DATE>
    <ns1:SHIPMENT>$CUSTOMER_ORDER/SHIP_METHOD</ns1:SHIPMENT>
<ns1:HANDLING_CHARGE>$CUSTOMER_ORDER/HANDLING_CHARGE</ns1:HANDLING_CHA
RGE>
</CUSTOMER_DATA>
```

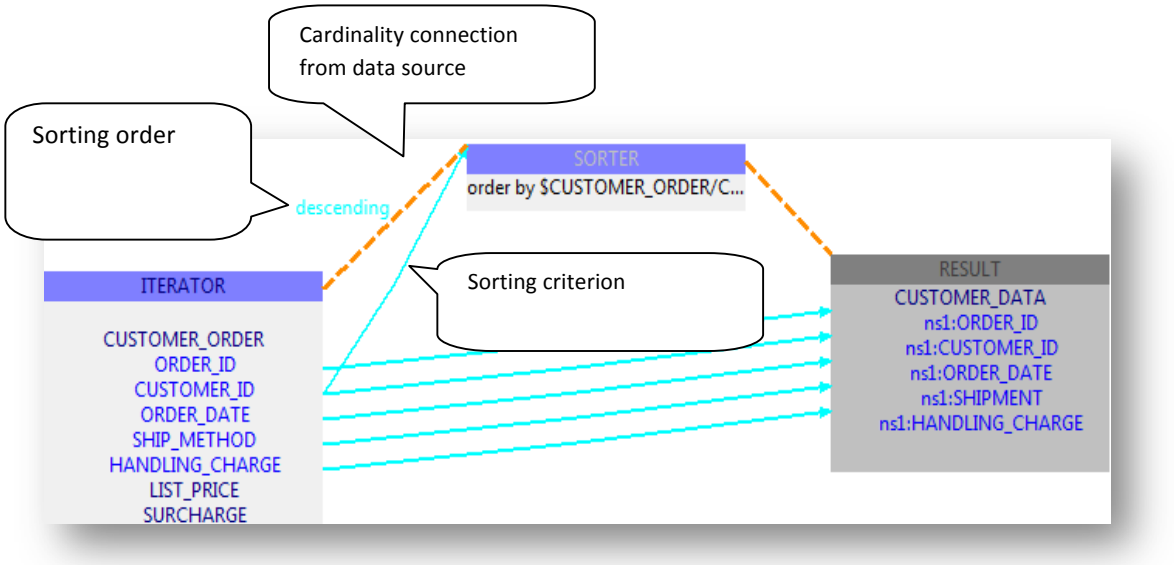


Figure 7.6: A query containing a sort operation. The graphical representation depicts sorting using a rectangular box.

7.2.2 Degree of Support for XQuery

The language-specific plug-in for XQuery works in accordance to the protocol prescribed by XEditor. It plays its part in breaking a query into a set of clauses and is able to collect information about variables as well as the scopes in which they occur. The XQuery plug-in was tested against common kinds of queries of varying complexity. The AST Reader is able to correctly handle nested queries as well as queries involving an outer/inner join. It successfully maps each kind of clause to a high-level operation that is understood by

XEditor. The plug-in currently being enhanced to cover the complete XQuery language and will be validated using the XQuery test suite [12].

7.3 Using XEditor for SQL

SQL provides the most popular query interface to relational databases. Table 7.5 shows some of the main aspects of the SQL language and their mapping to the high-level operations understood by XEditor. Continuing along the same lines as for XQuery, it is possible to develop a pluggable XEditor component for SQL. The component needs to implement the set of interfaces prescribed by XEditor. A pluggable component for a small subset of SQL92 was developed as a first proof of XEditor's multilingual capabilities.

No	SQL Query Clause	Description	High-Level Operation
1	FROM	The FROM clause specifies the collections of tables or views to which the query is to be applied.	Iteration
2	WHERE	The WHERE clause lists search conditions for items to add to a result set.	Filtering
3	ORDER BY	The ORDER BY clause specifies an ordering of the objects in the result collection.	Sorting
4	SELECT	The SELECT clause organizes the result by identifying the specific columns that form the result.	Collecting

Table 7.3: Subset of data analysis tasks expressible through SQL

7.3.1 Sample Queries in SQL

In this section, we show a few example queries written in SQL and illustrate their graphical representations in XEditor.

7.3.1.1 Select –Project -Join

Figure 7.7 shows the graphical representation of a simple select-project-join query. The query described forms a join between two tables - employee and department.

Query Source

```
SELECT e.designation  d.dept_name
FROM employee e, department d
WHERE e.department_id = d.department_id
```

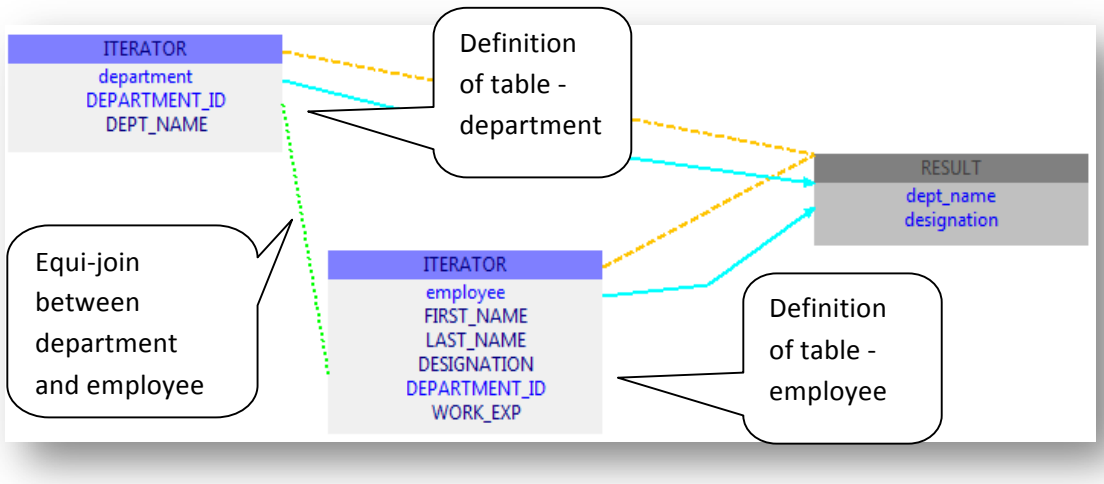


Figure 7.7: Graphical representation of a SQL select-project-join

7.3.1.2 Order By

Figure 7.8 shows the graphical illustration of a SQL query involving sorting as one of the intermediate steps before returning the result. The SQL sort operation is described by its

source(s) of input, the sorting criterion, and the sorting order (ascending or descending) for each criterion. The graphical representation for a sort operation shows each of the above.

Query Source

```
SELECT e.FIRST_NAME, e.LAST_NAME, e.DEPARTMENT_ID, d.DEPARTMENT_NAME
FROM employee e, department d
ORDER BY e.LAST_NAME
```

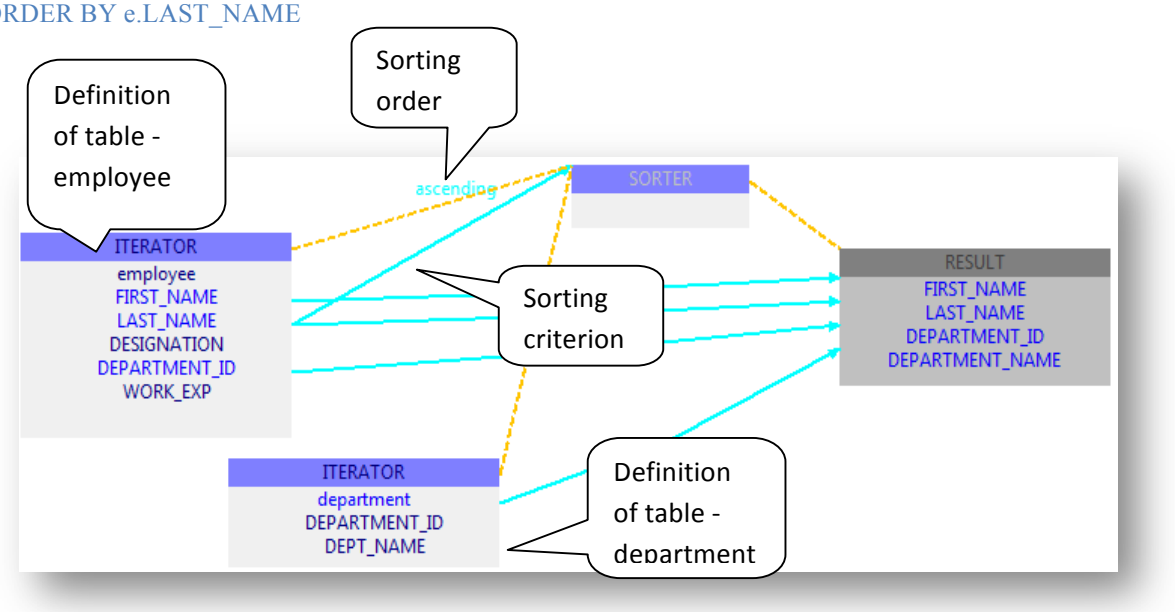


Figure 7.8: A SQL query with a sort operation.

7.3.2 Degree of Support for SQL

The language specific plug-in developed for SQL is able to handle basic SQL select-project-join queries**.** The plug-in is not advanced enough to handle nested queries and does not identify many clauses, including GROUP BY, LIKE and HAVING. The plug-in was prototyped to validate the basic architecture of XEditor and to begin to evaluate its capability to handle multiple query languages.

7.4 Incorporating other Languages

Data analysis is a common task, and a number of recent languages have been developed with built-in primitives to allow users to express and parallelize their data analysis tasks. Examples include Pig [15] from Yahoo and Hive [16] from Facebook. These languages share many common aspects with languages like XQuery and SQL. This section very briefly discusses the high-level operations expressible in Pig, emphasizing the commonality with the operations understood by XEditor.

7.4.1 Using XEditor for Pig

Pig is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs coupled with infrastructure for evaluating these programs. Pig queries articulate data analysis tasks in terms of set-oriented transformations, e.g., they apply a function to every record in a set, or they group records according to some criterion and apply a function to each group. A Pig relation is a bag of tuples and is similar to a table in a relational database; the tuples in the bag correspond to the rows in a table. As expected, the kind of high-level operations expressible in the language are similar to those well understood by XEditor. Table 7.4 shows the similarity between the basic operations in Pig and the operations modeled in XEditor.

No	Pig Language Clause	Description	High-Level Operation
1	FOR EACH	Generates data transformations based on columns of data.	Iteration
2	FILTER	Selects tuples from a relation based on some condition.	Filtering
3	ORDER	Sorts a relation based on one or more fields.	Sorting
4	DUMP	Displays the contents of a relation.	Collecting

Table 7.4: Similarity between operations in Pig and the operations modeled in XEditor.

Similar to the approach followed for XQuery and SQL, it should be possible to develop a language component for Pig which could be plugged into XEditor to allow two-way editing for Pig.

CHAPTER 8

Conclusion

In this project, we set out to build a graphical query editor that allows two-way editing, not for a specific query language, but with a plug-and-play architecture to support multiple languages. The goal was for a lightweight pluggable component to be able to plugged into the framework to make the editor provide a two-way editing environment for a given query language. The concept of having a language-agnostic framework supporting two-way editing was successfully proven. To demonstrate the concept, prototypes of language-specific pluggable components were developed for two query languages, namely XQuery and a subset of SQL. The result for XQuery was quite complete and was shown to be comparable to existing editors for XQuery. The user interface presented by the framework is currently evolving so that it can match the kind of polished user interfaces provided by commercial tools. These two prototypes demonstrated how a well- developed language-specific component can indeed leverage the flexibility and features offered by XEditor.

8.1 Discussion

The design and architecture of the XEditor framework has been initially validated via lightweight pluggable components for XQuery and a subset of SQL. The number of lines of code required for building both XEditor and the language specific plug-ins are shown in Table 8.1. The majority of the effort in this project went into building the re-usable framework of XEditor. The pluggable components, being much lighter and simpler, then

make use of the services offered by XEditor to provide their language-specific dual editing environments.

Module	Lines of Code
XEditor	10,376
XQuery plug-in	436
SQL plug-in	374

Table 8.1: Lines of code required to build XEditor and the language specific plug-ins

Size-wise, though the pluggable component for XQuery left a portion of the language uncovered, it is not expected to grow much bigger than a thousand lines of code when done. The pluggable component for SQL is primitive and was written just as an initial proof of concept.

8.2 Future Work

The following are aspects of XEditor that need to be worked on going forward:

1. Incorporating data source metadata:

   A query language, besides having a grammar, usually has a pre-defined notion of data sources that can be referred to by queries written in the language. For XQuery, an XML file accessed via the doc function might serve as a data-source, while for SQL, a table, view or a table function can act as a data source. In addition to data sources, each language comes with a library of pre-defined functions that are usable within a query. XEditor still needs to provide a mechanism to nicely import these details into the Resource Manager and to

subsequently show them on the user interface. These extensions will extend XEditor's language-neutral capabilities.

2. Providing a more polished user interface:

   The current user interface supporting graphical editing is clearly not yet advanced enough in comparison to commercially available tools. The current interface was developed with the priorities being to capture language capabilities and to get the architectural concepts right. Work is now focused on improving XEditor so that the framework can become acceptable to potential users.

3. Integration with existing source editors like XQDT:

   XQDT [13] is a very nice source editor for XQuery. The architecture of XQDT was studied in order to assess the effort required to integrate XQDT with the framework. XQDT integration will mean that the textual view of the query would be handled by such an advanced third-party editor, one that provides features like code highlighting and other user assistance. The unit of exchange between the framework and a text editor has been intentionally limited to a string buffer that contains the text for the query. This allows for minimal dependency and ease of integration. Though the required effort to integrate with XQDT was analyzed, it has yet to be implemented.

4. Open Source Contribution:

   This project is intended for eventual contribution to the Google Open Information Integration (Open II) effort [14], which aims at creating an open-source set of tools for information integration. The goal is for XQuery to be Open II's solution to XML data mapping.

REFERENCES

[1] Tiziana Catarci, Maria Francesca Costabile, Stefano Levialdi, and Carlo Batini, 'Visual Query Systems for Databases: A Survey, Journal of Visual Languages and Computing, 8(2), 215–260, (1997).Visual Query Systems for Databases: A survey

[2] Altova MapForce Professional Edition, Version 2008. http://www.altova.com.

[3] Stylus Studio, XML Enterprise Suite, Release 2. http://www.stylusstudio.com.

[4] M. Carey and the AquaLogic Data Services Platform Team. Data delivery in a Service-Oriented World: The BEA AquaLogic data services platform. *In Proc. of the ACM SIGMOD Conf. on Management of Data*, Chicago, IL, 2006.

[5] D. Braga, A. Campi, and S. Ceri. XQBE (XQuery By Example): A Visual Interface to the Standard XML Query Language. *ACM Trans. Database Syst*., 30(2):398{443, 2005.

[6] Borkar et al. Graphical XQuery in the AquaLogic Data Services Platform Submitted for Publication, Nov, 2009

[7] L. M. Haas, M. A. Hern´andez, H. Ho, L. Popa, and M. Roth. Clio Grows Up: From Research Prototype to Industrial Tool. In SIGMOD, pages 805–810, 2005.

[8] Cerullo, C and Porta, M. A System for Database Visual Querying and Query Visualization: Complementing Text and Graphics to Increase Expressiveness *Database and Expert Systems Applications, 2007. DEXA '07. 18th International Conference*

[9] P. Mork, A. Rosenthal, L. J. Seligman, J. Korb, and K.Samuel, Integration Workbench: Integrating Schema Integration Tools *in InterDB, Atlanta, GA, 2006.*

[10] Eclipse Consortium, Eclipse Graphical Editing Framework (GEF) – Version 3.0.1, 2004, http://www.eclipse.org/gef.

[11] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/

[12] XQuery Test Suite, http://www.w3.org/XML/Query/test-suite/.

[13] XQDT – XQuery Development Tools http://www.xqdt.org/

[14] Open II, Google Open Information Integration http://code.google.com/p/openii/

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing," in SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. New York, NY, USA: ACM, 2008, pp. 1099–1110.

[16] A. Thusoo, J. Sen Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony,H. Liu, P. Wyckoff, and R. Murthy, "Hive - a Warehousing Solution Over a Mapreduce Framework," in VLDB '09: Proceedings of the 35th International Conference on Very Large Data Bases. New York, NY,USA: ACM, 2009.