2007-01-02

Skuery: Manipulation of S-Expressions Using XQuery Techniques

Kevin Burke Tew
Brigham Young University - Provo

SKUERY: MANIPULATION OF S-EXPRESSIONS USING XQUERY TECHNIQUES.

by

Kevin Tew

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2006

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Kevin Tew

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Dr. Phillip J. Windley, Chair
Date	Dr. Dan R. Olsen Jr.
Date	Dr. Michael D. Jones

ABSTRACT

SKUERY: MANIPULATION OF S-EXPRESSIONS USING XQUERY TECHNIQUES.

Kevin Tew

Department of Computer Science

Master of Science

Data query operations inside programming languages presently perform their functions through the use of domain-specific, declarative expressions and by way of course-grain, API library calls. These methods of operation are practiced by relational databases as well as semistructured XML data stores. Layers of translation, which are necessary to transform data and instructions from the domain of programming languages to data query systems, negtatively effect the performance of data query operations. Skuery resolves this impedance by adopting XML as a native data type with a native representation (SXML). Likewise, query operations are defined in a general purpose programming language (Scheme in this case) not in an external data query environment. Skuery increases programmer productiv-

ity by abstracting layers of translation and unifying computational and data query operations under the auspices of a general purpose programming language.

ACKNOWLEDGMENTS

My wife Cheryl, my professors Phil Windley, Quinn Snell, and Mark Clement, my parents Ryan and Melanie Tew, and my research colleague Joey Ekstrom

Table of Contents

List of Figures

Listings

1 Introduction

As web systems continue to scale larger and larger, principles of design, construction, and maintenance are re-implemented and re-knitted together each time a new web system is created. Today's large webs systems are only possible because of the established theory and practice in the areas of data storage and computation. This has been sufficient up to the present, however moving up the stack, the complexity of integration between data storage, communication, and computation needs to be addressed. Traditionally, large web systems have segmented computation and data query into two separate architectural tiers. These two tiers are usually implemented in different programming languages on separate platforms and following different paradigms of thought. As a result, data query integration in programming languages is painful and requires multiple layers of translation. Integration of data query and computation research is part of the process of moving the principles of abstraction and reuse up the stack of complexity exhibited by today's web systems.

The separate worlds of computation and data storage have many differences. Computational languages adopt the data types of the underlying architecture. This usually consists of integer data type, floating, and a byte addressable string data type, etc. Databases, while usually supporting these common computational data types, offer a plethora of additional types such as fixed decimal, fixed size strings, variable sized strings bounded by a max size, and non-indexable blobs of character or binary data. Layers of translation decide how to map from computational types to data storage types and vice versa. Unlike conventional programming languages, databases permit primitive data types such as integers and floats to be nullable or not defined. Nullability adds an extra dimension to data storage types that must be further accounted for and accommodated by computational processes.

Serialization routines and data marshaling are another example of a low-level layer of translation. Data often has to be serialized to canonical formats to allow for interchange between the disparate systems of computation and data query. This can involve textualizing numbers and binary formatted data, changing byte and bit ordering, converting between character encodings, etc. In many cases, further type conversion/coercion is required to map the types of general purpose programming languages to the types supported by data storage engines.

Until recently, relational databases and their associated structured query language (SQL) have been the iconic systems for query based information extraction. Computation and business logic systems typically build up a string of declarative instructions to describe the data they wish to retrieve. These strings are just sentences of a domain-specific query language (DSL). In the case of data query systems, the sentences are most often SQL expressions sent to remote query executors that process the query and return the generated results. These domain-specific sentences become opaque boxes to the computation and business systems that generated them. It becomes very difficult for the computational language to ensure syntactic correctness, type correctness, and optimizability of the generated data query statement.

XQuery [6], however, represents the potential power and value that the next generation of data query and manipulation systems will deliver. XQuery supersedes the computational limitations of relational calculus and generalizes query theory from the rigidness of relational data to the more relaxed semistructured data model. The isomorphism between S-expressions and XML makes the Scheme programming language a ripe possibility for implementing XQuery-style data query as a native feature of the language.

Skuery, formed from the concatenation of the words Scheme and query, delivers a reference implementation of integration between functional programming and data query languages. By breaking the tight coupling between query operations and relational storage systems, the query pattern becomes a generalized programming language feature. Abstracting from the common medium of XML to other forms of semistructured data such as parse trees, LDAP directories, file systems, bulk container data types, etc, the query feature of programming languages can be retargeted to any data source rather than re-implemented for each data source.

1.1 Parts Query Example

Following is an introductory XQuery example implemented first in Java using the Saxon library and then in Skuery. The reader can observe the XQuery syntax and compare it to the corresponding Skuery syntax. The example demonstrates increased programmer productivity with Skuery by reducing the quantity of code needed to describe the query. Finally the example gives a brief tutorial of the construction and execution of an XQuery.

The example XQuery transforms the linear parts list, *Listing 1*, to a hierarchical tree, *Listing 2*, showing the parent-child relationship of the components. Implemented in Java, the example requires the original query code in XQuery format as well as Java utility code to construct and initialize the interpreter environment that the query will execute in. In contrast, the Scheme implementation seamlessly embeds XQuery functionality, encoded in S-expressions, allowing unrestricted integration and flow between query and computation realms.

The XML document shown in *Listing 1* represents the data source for the query. It consists of a simple container type `partlist` that encapsulates a collection of

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<partlist>
  <part partid="0" name="car"/>
  <part partid="1" partof="0" name="engine"/>
  <part partid="2" partof="0" name="door"/>
  <part partid="3" partof="1" name="piston"/>
  <part partid="4" partof="2" name="window"/>
  <part partid="5" partof="2" name="lock"/>
  <part partid="10" name="skateboard"/>
  <part partid="11" partof="10" name="board"/>
  <part partid="12" partof="10" name="wheel"/>
  <part partid="20" name="canoe"/>
</partlist>
```

Listing 1: partlist.xml - Original Semistructure data in XML format.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<parttree>
  <part partid="0" name="car">
    <part partid="1" name="engine">
      <part partid="3" name="piston"/>
    </part>
    <part partid="2" name="door">
      <part partid="4" name="window"/>
      <part partid="5" name="lock"/>
    </part>
  </part>
  <part partid="10" name="skateboard">
    <part partid="11" name="board"/>
    <part partid="12" name="wheel"/>
  </part>
  <part partid="20" name="canoe"/>
</parttree>
```

Listing 2: Expected query result from XQuery execution.

`part` types. Each part has an identifier (`partid`), a description (`name`), and a parent part identifier (`partof`) if it is a component of another part.

Listing 2 demonstrates the desired result a `parttree` element depicting the part/sub-part relationships in a hierarchical fashion, The car part contains an engine part and a door part. A window and a lock are parts that make up the door and so forth.

```
declare namespace f="my-functions.uri";
declare function f:one_level ($p as element()) as element()
{
    <part partid="{ $p/@partid }"
          name="{ $p/@name }" >
        { for $s in doc("partlist.xml")//part
            where $s/@partof = $p/@partid
            return f:one_level($s) }
    </part>
};
<parttree>
  { for $p in doc("partlist.xml")//part[empty(@partof)]
    return f:one_level($p) }
</parttree>
```

Listing 3: q1.xq - XQuery code required to generate the expected query result for the Java implementation.

The part list XQuery in *Listing 3*, first declares a recursive user-defined function, `one_level`. The `one_level` function then takes a single part as a parameter and recursively returns all its sub-parts. The `parttree` element represents the query body. The element content of `parttree` is constructed by an XQuery FLWOR [6] expression. The FLWOR loop calls the `one_level` function for each top level part in the partlist. Top level parts are identified by the lack of a `partof` attribute. *Listing 2* is the result of executing the XQuery in *Listing 3*.

The Java configuration code, *Listing 4*, along with the XQuery code in *Listing 3* constitute the code necessary to execute the part list example on the Java platform. After using API calls to construct an XQuery interpreter, the Java configuration code loads the query into the interpreter, compiles the query, then loads the query input into the XQuery evaluation environment, and finally executes the query. The `StaticQueryContext` compiles the query source text into a `XQueryExpression`. The `DynamicQueryContext` is then created and initialized with the source XML document `partlist.xml`. Finally the XQueryExpression, `exp`, is executed in the

```
public class PartsXQuery {
  public static void main(String[] args) {
    Configuration config = new Configuration();
    StaticQueryContext staticContext
        = new StaticQueryContext(config);
    try {
      XQueryExpression exp = staticContext.compileQuery(
          new FileReader(new File("q1.xq")));
      DynamicQueryContext dynamicContext =
          new DynamicQueryContext(config);
      dynamicContext.setContextNode(i
        staticContext.buildDocument(
          new StreamSource(new File("partlist.xml"))));
      Properties props = new Properties();
      props.setProperty(OutputKeys.METHOD, "xml");
      props.setProperty(OutputKeys.INDENT, "yes");
      exp.run(dynamicContext,
              new StreamResult(System.out), props);
    } catch (Exception e) { e.printStackTrace(); } } }
```

Listing 4: Java code to load and execute the XQuery example above.

```
(define (f:one_level $p)
 `(part (@ partid ,(xpath $p "/@partid" )
           name ,(xpath "$p/@name" ))
   ,(sxq:for (($s (xpath (doc "partslist.xml") "//part" )))
      (sxq:where (eq? (xpath "$s/@partof")
                      (xpath "$p/@partid")))
       (sxq:return (f:one_level $s)))))

`(parttree ,(sxq:for
             (($p (xpath (doc ("partlist.xml")
                            "//part[empty(@partof)]"))))
             (sxq:return (f:one_level $p))))
```

Listing 5: Native embedding of XQuery in Scheme.

DynamicQueryContext, `dynamicContext`, using Properties `props`, and the query results are streamed to the `System.out` file handle.

Listing 5, demonstrates the conciseness of implementing data query as a native operation in the Scheme programming language. Note the absence of foreign DSL syntax, instantiation and initialization of an XQuery interpreter, and the need to explicitly load and compile the query source. Code sharing between the computation and data query layers is made possible because XQuery functions are defined using normal Scheme syntax. Finally, literal XML following the SXML convention is constructed using simple S-expressions.

1.2 Thesis Statement

The XQuery language abstracts the low-level details of hierarchical queries, permitting the software developer to focus on the application of XQuery techniques verses their implementation [29]. S-expressions and XML share similar characteristics; both are hierarchical, semistructured data formats and conform to a functional programming style. Integration of data query processing and programming languages increases programmer productivity by drastically reducing the code size and ancillary structures needed to describe data query operations. Embedding the XQuery standard in Scheme using list comprehension transformations improves the expressive power of Scheme and simplifies the use of data query processing in software products.

2 Background

Programming is all about data manipulation. Without input or output computational processes have no influence. Early programming languages focused on the mechanics of moving data from one location to another and the micro operations needed to manipulate data. Extensive use of abstraction has helped programming languages evolve away from the micro-mechanics of data manipulation to a broader purpose of data transformation as it passes through a program. While much change has occurred, widely used languages such as C and Java still require the developer to manage much of the mundane mechanics of data manipulation. In the realm of text processing, the programming language Perl is the counter example to C and Java. Perl's use of regular expressions allows the programmer to concisely declare the pattern of data transformations to perform. Perl then takes care of actually carrying out the work of string manipulation to accomplish the tasks described by a regular expression. Likewise, advances in data query abstraction must relieve the programmer from lower level complexities without segmenting data query operations from computational processes.

The rest of the chapter will proceed as follows. First data centric programming is presented as an example of the cognitive patterns Skuery advocates. The field of semistructured data is then briefly summarized after which XML is given as a primary example of semistructured data. Semistructured data and XML are contrasted, but shown to be interoperable. Next Scheme, a dialect of Lisp, is introduced followed by a comparison between XML and S-expressions. Having built a foundation of data centric programming, XML as a form of semistructured data, and Scheme functional programming, XQuery is introduced as an encompassing implementation of these ideas. Skuery will exploit and enhance the compatibility and similarities between XML, S-Expressions, and data query processes to create a larger abstraction between computation and data query systems. Last of all List comprehensions are presented as the union of relational optimizability and computational completeness.

2.1 Data Centric Programming

The philosophy of data centric programming encourages software developers to focus on how programs transform and manipulate data. One method of becoming more data centric is using functional programming primitives such as filter and map to force the programmer to think directly about the data involved. The Python community is an excellent example of how a data centric programming mentality can improve general purpose programming by introducing highly theoretical concepts into commonplace programming [32]. Initial versions of Python contained higher order map and filter functions. With version 2.0 Python added support for list comprehensions which placed added focus on the data passing through the program rather than the looping and conditional constructs needed to manipulate and process data. One of Skuery's primary objectives is to empower programmers with tools to slice, dice, and manipulate data without having to constantly deal with intricate and often repetitive details of subordinate data query processes. Application programmers should be freed to spend time and resources contemplating how to use data query operations, rather than how to implement and integrate data query operations. Data centric programming stresses the core importance of data and data manipulation in programming rather than distancing data query from programming languages. It is this mentality that Skuery attempts to adopt from data centric programming.

2.2 Semistructured Data and XML

"Semistructured data is best described as a graph-based, self-describing object instance model [13]." Common examples of semistructured data often result from integrating two heterogeneous data types. The data structure for storing a photograph is a good example; the title, caption, and date are textual while the image itself is a JPEG encoding (binary data). The binary data formats of a JPEG photograph or the MP3 encoding of a song are very different from the descriptive string attributes of the structures they are associated with. The exact structure of semistructured data can even be unknown or variant across particular instances of a data type.

A primary example of semistructured data is the Extensible Markup Language (XML). XML, a derivative of SGML, has become the universal format for data exchange and the de facto standard for data representation and storage. Formally described, XML is an unranked, labeled, ordered tree. An XML document is a textual encoding of hierarchical containers and sub containers [22]. XML's textual format makes it human readable and approachable by a wider audience, while its semistructured quality keeps it readable and parseable by machines as well.

Semistructured data is self-described; every data instance is annotated with its own version of the schema. The static meta-data position of schema in the relational model is relaxed with semistructured data to the point that in many cases schema becomes data. Taking XML as a working example, its replicated schema is represented in the form of element tags and attribute names. The element and attribute names of XML that give the enclosed data both structure and type are often interpreted as data themselves instead of as schema. Replication of schema, while convenient for data integration and on the fly schema evolution, results in additional space cost for storage and time cost for processing.

Semistructured data, XML specifically, provides an abstraction for the myriad of data types and data formats in existence. The lack of strong distinctions between data and schema in the semistructured data model extends its applicability as an abstraction. Skuery's use of XML as a universal data format permits programmer focus to move further up the stack of problem complexity. However XML does not define a processing or query model for asking questions about the data it encodes.

2.3 Dissimilarities between Semistructured Data and XML

The semistructured data model is similar in many aspects to the XML Infoset data model. However the models collide in a couple technical areas.

One of the small annoyances of XML to relational data researchers has been the implicit order of XML elements. Its human readable textual format is a core characteristic of XML. Text by nature has order, word order in prose, and element order in XML. Hence every structure in an XML document has a document relative position, often referred to as document order.

Semistructured data researchers have adapted their unordered graph data model assumptions to accommodate XML. An ordered tree best represents the textual format of an XML Document. Hierarchical tree structures in their primitive form are unable to directly encode graphs, trees being a proper subset of graphs. XML remedies this problem with element IDs and attributes that can contain a list of IDREFs. Elements become nodes in the graph and IDREF attributes become lists of peer nodes, which share an edge with the parent element of the attribute.

Disregarding order and viewed semantically through the use of IDs and IDREFs, XML can be easily transformed into data models fully expressing graph structure. Conversely, maintaining document order as a property of each element/node allows relational engines to order query results by document order and retain the ordered nature of the XML data model [16]. While XML diverges slightly from relational researcher's view of what semistructured data should look like, XML is sufficiently dynamic and moldable to represent the unordered graph structure of semistructured and relational data.

2.4 Scheme

Scheme [35], a dialect of LISP [25], was initially developed at MIT by Guy L. Steele Jr. and Gerald Jay Sussman. At the time of its conception, Scheme was a radical derivative of LISP, due to the introduction of static lexical scoping. Programs in Scheme are constructed using symbolic expressions (S-expressions or sexprs) to express both data and execution instructions, just as LISP programs are.

Features such as first class and higher order functions, closures, continuations, etc loosely define functional programming. While adopting static lexical scoping from Scheme, Common Lisp has branched beyond the definition of functional programming adopting constructs and techniques from a variety of programming methodologies. Proponents describe the current evolution of Common Lisp as a multi-paradigm programming language rather than just a functional style language. Scheme has stayed true to its roots as a functional language and is much more pure than Common Lisp. Skuery's extensive use of higher order functions abstracts the complexity of data query operations and make their presence in Scheme seem natural.

Scheme S-expressions representing data, often also have execution semantics that can be evaluated directly by the Scheme interpreter, without prior transformations. The opposite is also true. Because the code of Scheme programs is constructed as S-expressions, it also often has significance as data and can be manipulated in Scheme just as data is. The blurriness between code and data in the Scheme programming language is very analogous to the weak distinction between schema and data in the semistructured data model.

Because of the similarities between XML and S-expressions, Scheme is able to represent XML internally as simple S-expressions. This enables Scheme to manipulate XML natively just as if it was simply a list. However Scheme does not have a query language or a standard persistence framework.

2.5 XQuery

Just as semistructured data can be seen as an abstraction for the varying types of data, XQuery represents an abstraction of various data query languages. The language design of XQuery is modeled from functional language theory. At its most primitive level, XQuery sees the data it manipulates as simple sequences of primitive data values, much like Lisp sees everything as a list of symbols. Functional language features adopted by the XQuery standard are best represented by a few core ideas [10].

- Purity: expressions and function applications do not exhibit side effects.
- Compositionality: the ability to create new functions composed of previous more primitive functions.
- Recursion: the ability of a function to call itself with a smaller subset of values than with which it began.

XQuery is strongly typed and requires dynamic typing at runtime. While not required by the specification, design decisions were made to explicitly allow for static type checking during query analysis. Like Scheme, XQuery expressions are pure. An XQuery user however is not prohibited from introducing side effects in user-defined functions.

Being the most recent standard for XML query/manipulation, XQuery incorporates many principles and features from its predecessors. It is a companion, not a competitor technology, to previous XML manipulation languages such as XSLT and XPath. XQuery is the first XML manipulation language that natively supports function compositionality. It also supports user-defined functions, functions as first class objects, and higher order functions. Through the support of recursive functions, XQuery is not only relationally complete, but also computationally sound [8]. Philip Wadler [14] has proposed an algebra for XQuery and Torsten Grust [19] has shown that XQuery expressions can be translated into the relational algebra.

The basic structure of XQuery is the FLWOR (For, Let, Where, Order, Return) loop that processes data and performs computation. XQuery also includes all of the XPath 2.0 specification enabling path traversals and structure based matching. Furthermore, XQuery fully supports the XML Schema 1.0 specification for type specification.

XQuery was designed principally for XML data or data that can be easily transformed into XML. Its use and influence however have extended well beyond just XML. XQuery has become widely adopted in the relational model community as a next generation query language. “XQuery can be considered as a generalized interface for querying hierarchical data [29].” While not widely utilized with non-XML data, the XQuery specification can and has been applied to other semistructured data formats besides XML.

- File systems - hierarchical path matching and enhanced Unix find utility functionality [30]
- Compiler and Interpreter parse trees - optimization and instruction reordering [30]
- Directories (LDAP) - query employees whose certification has expired.
- Code Reflection - query the source code s-expressions of lisp language style programming languages.

In theory, XQuery should be applicable to any data container model that is iterable [39]. XQuery is well suited to hierarchical data and was principally created for XML manipulation. However it is not confined to these domains.

XQuery succeeds in providing a query language for XML and accommodating the varying data formats and types that XML often encodes. Its applicability to a wide range of data types makes XQuery an excellent abstraction for data query operations. XQuery however is still an external query language. While it has incorporated more computational features than its predecessors, XQuery is not a general-purpose computing language.

2.6 List Comprehensions

List comprehensions are a syntactic feature of many functional languages. The syntactic and semantic forms of list comprehensions are derived from set comprehensions described in Zermelo-Frankel set theory. However, "they add no fundamentally new [computational] power [21]." Instead, list comprehensions provide concise expression of complicated looping behavior, the basis of query.

List comprehensions have the basic form of $[< expression > \mid < qualifier >^{+}]$. Qualifiers are either generators ($x \leftarrow list$) or filters ($x < 10$). The result of the

$$\begin{aligned} &list\_times\_2\ xs && = \{times2(x) | x \Leftarrow xs, even(x)\} \\ &list\_times\_2\ [4,5,6,7] && = [8,12] \end{aligned}$$

Figure 1: List comprehension example with filter qualifier.

$$\begin{aligned} &vector3\ xs\ ys\ zs && = \{(x,y,z) | x \leftarrow xs, y \leftarrow ys, z \leftarrow zs\} \\ &vector3\ [4,5]\ [6,7]\ [3,2] && = [\ (4,6,3),(4,6,2), \\ & && \quad (4,7,3),(4,7,2), \\ & && \quad (5,6,3),(5,6,2), \\ & && \quad (5,7,3),(5,7,2)] \end{aligned}$$

Figure 2: Nested list comprehension example.

$$\begin{aligned} &vector3\ xs\ ys\ zs && = \{(x,y,z) | x \leftarrow xs, y \leftarrow ys, divisible\_by\_2(x,y), z \leftarrow zs\} \\ &vector3\ [4,5]\ [6,7]\ [3,2] && = [\ (4,6,3),(4,6,2)] \end{aligned}$$

Figure 3: Lexical scoping in list comprehensions.

$< expression >$ for each generated tuple, that is not filtered, is appended to an output result list *Figure 1*.

Multiple generators in sequence left to right, as in *Figure 2* are functionally equivalent to nested for blocks in imperative style programming. Subsequent qualifiers can reference previous variable definitions, but not later ones, which in essence provides lexical scoping *Figure 3*. Pattern matching on the left hand side of a generator can also be used as an additional filtering mechanism [21].

SQL statements are easily optimized because their expressions operate with respect to (are closed under) relational algebra. Database processing languages, which express queries as procedures instead of expressions, trade easy optimization for computational power. List comprehensions are intriguing in that they are defined in terms of lambda calculus, yet easily optimized. Further more, list comprehensions have been shown to be relationally sound, expressively equivalent to relational calculus [39].

List comprehensions are the theoretical foundations for the abstractions made by Skuery. Skuery integrates the computational completeness of Scheme with relational data query processes. Optimized performant abstractions, such as Skuery, are possible because of list comprehensions.

3 Current State of Data Query

Data query has evolved from the Indexed Sequential Access Method (ISAM) of mainframes and minicomputers to the relational and semistructured data stores of today. Looking to the recent past, relational calculus and object persistence through object-relational bridges have been the most successful data storage abstractions. SQL, the current de facto data query standard, is examined first. The CRUD methodology, which descends from SQL, is then compared and contrasted with XQuery. Finally the successes and shortcomings of object persistence are examined and groundwork for Skuery's contributions is laid out.

3.1 SQL

SQL, originally Structured English Query language (SEQUEL)[7], was created by Donald D. Chamberlin and Raymond F. Boyce at IBM as the query component for their relational database system, "System R [26]". Modern SQL is best described as a set-based, declarative, domain-specific, data query language. System R's relational model came from Dr. Edgar F. Codd and his seminal paper "A Relational Model of Data for Large Shared Data Banks [12]". Pure standard SQL is not Turing complete, principally because of its lack of recursion. When used as a general-purpose language, SQL is tedious because of its lack of support for encapsulation, modularity, and reuse. The latest editions of the SQL standard have begun to include non-relational features such as object orientation and early support for XML. Many database vendors have extended their implementations of SQL to include general computability features [41]. These extensions, however, are neither standardized nor interoperable.

XQuery, a core component of Skuery, is built directly on top of the contributions of SQL. Following the integrating model of list comprehensions, Skuery's FLWOR implementation can express the set of queries described by the relational calculus. Skuery goes beyond SQL's contributions by integrating a full set of both query and general purpose programming constructs.

3.2 XQuery/CRUD comparison

Database driven web applications follow a data query cycle of create, read, update, and delete operations, commonly referred to using the acronym CRUD. The CRUD methodology stems from the basic SQL operations INSERT, SELECT, UPDATE, and DELETE. The similarities between CRUD and XQuery leads to an informal mapping between the two data query methodologies.

		List Comprehension	
CRUD	XQuery	Course Grain	Fine Grain
Create	XML Element Construction	Result Output	Generating Qualifiers
Read	XML Query	Result Output	Generating Qualifiers
Update	Function w/ side effects	-	Result Expression
Delete	Function w/ side effects	-	Filter Qualifiers

Figure 4: CRUD, XQuery, and List Comprehension Similarities

The create operation of CRUD is usually thought of as a series of SQL INSERT statements that creates a record in a relational data store. In the XQuery model, XML element construction is comparable to CRUD's create operation. As shown in *Listing 6*, new XML fragments can be created by XQuery queries and functions. The function `generate_product_info` shows XML construction best by taking an array of values and creating an XML fragment, where element names describe the data being encapsulated.

```
declare function
generate_product_info ($i)
{
    <product name="{$i[0]}"
        <product_manager name="{$i[1]}"
        <product_asst_manager name="{$i[2]}"
        <production_quota>{$i[3]}</production_quota>
        <actual_production>{$i[4]}</actual_production>
        <defect_ratio>{$i[5]}</defect_ratio>
        <next_period_goal>{$i[6]}</next_period_goal>
    </product>
};
<report>
  <manager name="Roger">
  <report_frequency>quarterly</report_frequency>
  { for $p in production_data() ]
    return generate_product_info($p) }
</report>
```

Listing 6: XQuery code that creates XML element construction examples.

XML element construction is analogous to the creation of a database record or table row in the CRUD model of data query. In XQuery, XML elements are primitive data types; no special purpose XML construction and manipulation library is needed. XML elements can be constructed in XQuery as easily as integers and strings are constructed in other programming languages.

The read operation of CRUD denotes information retrieval or query. In SQL, CRUD reads are SELECT statements. CRUD reads can be as simple as a returning a single row of a relational table or a complex composition of joins, filters, and sub-selects. In a likewise manner, XQuery's FLWOR and XPath expressions can slice, dice, and rearrange any subset of XML source data presented to it.

Looking at the update and read operations of CRUD, the analogy with XQuery starts to weaken. The CRUD model of data query evolved from the transactional style of state change driven by imperative computation. Hence, update and delete operations change or remove previously created program state. An XQuery update standard is currently being developed, primarily motivated by the need for update interaction with transactional, relational databases. The CRUD semantics of updates and delete can be modeled in XQuery as side effects of user-defined functions. However, all programming models are inherently just transitions between states. Functional paradigms, such as XQuery model state transitions by create new state from previous program state. In contrast, imperative paradigms transition to new state by changing or deleting the current program state.

Looking at CRUD from a slightly different angle, CRUD operations can be compared to the operations of list comprehension. At a course-grain level, the results of a list comprehension can be seen as create and read CRUD operations. From a finer-grained perspective read and create operations can be compared to the generating terms of a list comprehension. The update operation is simply the execution of a list comprehension's transformational expression. Finally CRUD's delete operation can be expressed in list comprehensions through the use of filtering qualifiers.

The well know web development model of CRUD translates into XQuery syntax. CRUD operations, when viewed as functional transformations, are also expressible as list comprehensions. While there is some impedance between the imperative model of CRUD and the functional model of XQuery, equivalent semantics exist in both models.

3.3 JDBC and Java Persistence Engines

Java based persistence and query methods deserve mention when discussing data query operations in programming languages, especially in regards to large-scale web applications. The earliest member of this family is the Java Database Connectivity API (JDBC). In simplest terms JDBC is the Java client side implementation of

SQL based database interactions. JDBC consists of Java API calls for constructing, executing, and processing the results of SQL defined data query operations. Using JDBC, data query instructions are written in the external data query language SQL. In contrast, Skuery data query instructions are written in native Scheme S-Expressions.

Using JDBC as a building block, developers started creating custom Java object persistence libraries. Building upon the basics of JDBC, Java developers quickly wanted transparent object persistence. Seeing the need for a standardized, transparent object persistence, Sun Microsystems introduced Container Managed Persistence (CMP) Entity Beans as a part of EJB 2.0, which permitted easy storage and retrieval of Java objects. Persistent engines since then have done an excellent job of abstracting the details of serializing and deserializing an object back and forth to the data storage layer. They have mastered the art of object persistence, but have struggled to offer innovative solutions to the more general problem of data query. The EJB attempt is a feature called finders which makes possible simple query operations such as: return the object with primary key 12, return objects where the name attribute equals George, return all objects of the type Person, etc. More complicated finders require the developer to write vendor specific SQL that provides the persistence engine with the list of object ids to retrieve to form the result. Queries or finders that do not return predefined objects have to be implemented in using JDBC constructs.

3.4 Lisp Persistence Solutions

Elephant [5], an object database for Common Lisp, is an open source project that employees BerkeleyDB as its primary back end store. Initial support exists for

PostgreSQL and SQLite3 as well. Elephant provides simple storage and retrieval of Lisp values and Common Lisp Object System (CLOS) slot values. All serialization, object mapping, and data marshaling operations are completely transparent and require no developer intervention. Except for lambdas, closures, packages, and streams, Elephant is able to store most Lisp primitives Concurrent multiple database access as well as rudimentary cursors and transactions are also supported.

AllegroCache [1] from Franz Inc. is a commercial persistence product for use with Allegro CommonLisp. Instead of using BerkeleyDB, AllegroCache uses it own Lisp native B-tree implementation for data storage. AllegroCache supports the persistence of most data types including maps, sets, and CLOS objects. One of AllegroCache's attractive features is a prolog dialect permitting complex logic and recursive query ability. The added computational power of the embedded prolog interpreter makes implementing complex data query operations more intuitive for the developer. The prolog features of AllegroCache are implemented external to Common Lisp. While powerful, the prolog features are not syntactically and semantically integrated into Common Lisp. Skuery on the other hand makes data query operations syntactically native features in general-purpose programming languages.

4 Design, Implementation, and Methods

Skuery's architecture and construction proceeds from the layered design pattern. To accomplish its goals, Skuery must first be able to parse both native XQuery 1.0 syntax and S-Expression encoded XQueries. This requires the development of an XQuery 1.0 lexer, parser, and abstract syntax tree for Scheme. The XQuery 1.0 specification includes an XPath 2.0 implementation as a required dependecy. Currently only an XPath 1.0 implementation exists for Scheme, so modifications are necessary to meet XQuery's requirements. The most crucial component needed is a FLWOR implementation for Scheme. Code examples and diagrams illustrate design choices and methods of implementation. The chapter concludes with control XQuery implementation that Skuery will be compared with.

4.1 Skuery Processing Pipeline

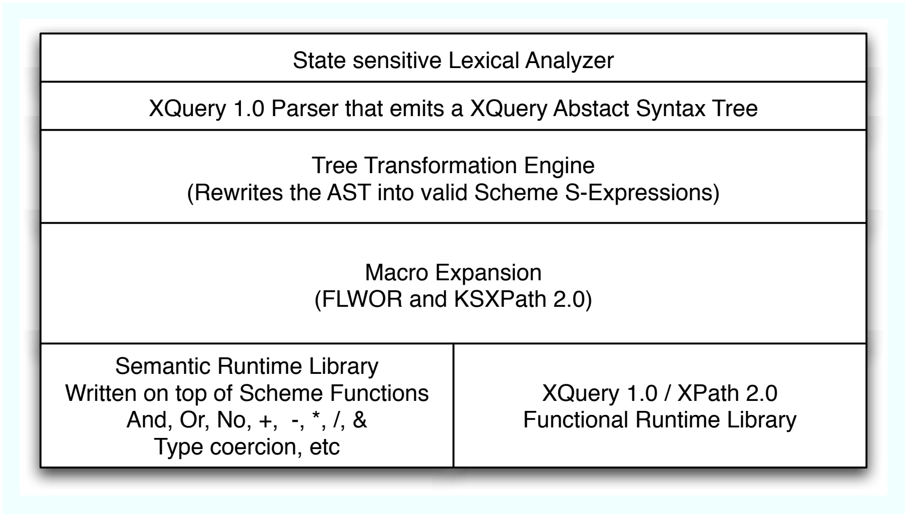
State sensitive Lexical Analyzer
XQuery 1.0 Parser that emits a XQuery Abstact Syntax Tree
Tree Transformation Engine
(Rewrites the AST into valid Scheme S-Expressions)
Macro Expansion
(FLWOR and KSXPath 2.0)
Semantic Runtime Library
Written on top of Scheme Functions
And, Or, No, +, -, *, /, &
Type coercion, etc
XQuery 1.0 / XPath 2.0
Functional Runtime Library

Figure 5: Skuery Processing Pipeline

```
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person[@id = "person0"]
  return $b/name/text()
```

Listing 7: XQuery FLWOR For Clause Implementation.

```
(begin
  (sxquery:FLOWR
    ((sxquery:let auction (doc "auction.xml")))
    (sxquery:return
      (sxquery:FLOWR
        ((sxquery:for b ((sxpath2
            `(site people (person ((equal? (@ id *text*)
                "person0"))))) auction)))
        (sxquery:return ((sxpath2 `(name ((text)))) b))))))
```

Listing 8: Skuery FLWOR For Clause Implementation.

As depicted in *Figure 5* the Skuery extension to Scheme operates in a pipeline fashion. Skuery accepts two forms of initial input: native XQuery 1.0 syntax (*Listing 7*) and S-Expression encoded XQuery syntax (Skuery syntax *Listing 8*).

In the case of native XQuery 1.0 syntax, *Figure 7*, is lexed and parsed into an XQuery abstract syntax tree described in *Appendix A*. Taking an abstract syntax tree as input, the tree transformation engine generates valid Scheme S-Expressions. FLWOR loops are expressed using macros while XPaths are represented in a S-Expression based domain language. At this point all XQuery and XPath syntax has been transformed to S-Expression, allowing full inter-operation with Scheme functions, macros, modules, etc. In order to provide stronger feature for feature compatibility with the XQuery specification, Skuery provides a runtime library of functions with XQuery semantics. The semantic runtime library contains explicit type casting functions, implicit type coercion behavior, primitive arithmetic and boolean logic operations, etc. While, Skuery operations can and do use Scheme operations natively, the runtime library provides additional compatibility for interfacing with other XQuery implementations. Skuery also implements, in Scheme, a subset of the functions and operators defined in the W3C's XQuery 1.0 and XPath 2.0 Functions and Operators specification such as fn:empty, fn:equal, fn:zero-or-one, and fn:count [24].

4.2 Parsing XQuery 1.0

Parsing XQuery syntax is a little more complicated than the traditional infix expression parsing example. XQuery token types are dependent on the current state of the parser. Sometimes a double quoted string is just a normal string of text. XQuery however allows XQuery expressions to be embedded inside value strings of XML attributes. Unlike other uses of double quoted strings in XQuery, strings used as attribute values cannot always be lexically analyzed as a single string token.

PLT Scheme comes with pre-packaged schemized versions of the Lex and Yacc parsing tools in a module called "parser-tools". PLT's lexer library helped simplify the representation of XQuery tokens, because it uses basic regular expressions to describe lexing rules. Unfortunately the regular expression support in "pareser-tools" falls considerably short of Perl Compatible Regular Expressions (PCRE) functionality. While named after Perl, PCRE is an independent C library implementation of Perl's regular expression functionality.

Support for captures is one of the most valued features of PCRE. Captures allow subsequent access to tagged sub-parts of a regular expression match, upon a successful match. The regular expression `/(\w+) \s+ (\w+)/` represents two words separated by white space. The pairs of parentheses specify the presence of two sub captures within the regular expression. The lexer generated using the "parser-tools" package does not support captures, so it returns the entire match as one string. PCRE on the other hand returns the entire match as a string, but also allows callbacks to return the sub-parts of the regular expression indicated by the pairs of parentheses. A natural extension for PLT would be to allow post-match access to the string segments that represent the matches of these named sub-expressions.

A second missing feature of PLT's lexer is the ability to easily exchange state between the generated lexer and parser. When parsing XQuery, the reference OCaml implementation employs a meta-lexer which maintains a stack of pushed lexer states. During normal operation the parser repeatedly asks the meta-lexer for subsequent tokens. The meta-lexer examines the top of the lexer state stack and dispatches lexing duties to the appropriate sub-lexer corresponding to the state at the top of the stack. Lexing state transitions are accomplished by lexing rules that modify the lexer state stack upon successful matches.

Parsers generated using the "parser-tools" package, obtain tokens by calling a 0-arity token-generator function passed to the parser instance during instantiation. PLT Lexers on the other hand are 1-arity functions. They take an input stream as their only argument. In order to satisfy the 0-arity requirement of the parser, a lambda abstraction is used to close over the lexer function with input stream value resulting in a 0-arity token generating function.

```
lexerFunc( inputSream )
myInputStream

tokenGeneratingFunc = lambdaAbstraction ()
{
  lexerFunc( myInputStream )
}

parserFunc ( tokenGeneratingFunc )
```

Listing 9: Use of closures to permit stateful lexing.

Because lambda abstractions in Scheme are closures, they preserve access to lexical variables referenced in their body. The lexer state stack and meta-lexer operations can be defined lexically and preserved by the closure, just as the input stream is in the example *Listing 9*.

In order to support both native XQuery expressions and S-Expression expressed XQuery syntactical front ends, Skuery translates each syntactic form to a common XQuery abstract syntax tree. Initially, Skuery represented XQuery abstract syntax as just inline lists. Theses lists were constructed by the parser recognition actions using native Scheme list operations such as cons, list, append, etc. As Skuery grew in supported features, the abstract syntax format converted from a primitive list format to a linked abstract data type format using a modified version of the Essentials of Programming Languages datatype module [15]. By implementing an XQuery lexer, parser, and abstract syntax tree in Scheme, Skuery can use pre-existing XQuery code and libraries in their original format. Developers are free to integrate existing XQueries with the more powerful features of Skuery abstractions.

4.3 KSXPath: Scheme Syntax for XPath 2.0

XPath 1.0 [11] can be simply described a domain-specific language for specifying tree traversals and sub-tree selections. XPath 2.0 [4] builds upon version 1.0 by adding more powerful looping constructs, decision flow control, and user-defined combinators. The XQuery 1.0 specification includes the XPath 2.0 specification as its tree traversal description language. Most importantly to Skuery, XPath 2.0 supports user-defined node filtering and selection functions during tree traversals. Having an XPath 2.0 implementation in Skuery would allow developers to use the full power of Scheme to write XPath filtering and selection predicates.

The Scheme XPath implementation (SXPath) [23] is architected using a layered approach as depicted in *Figure 6*

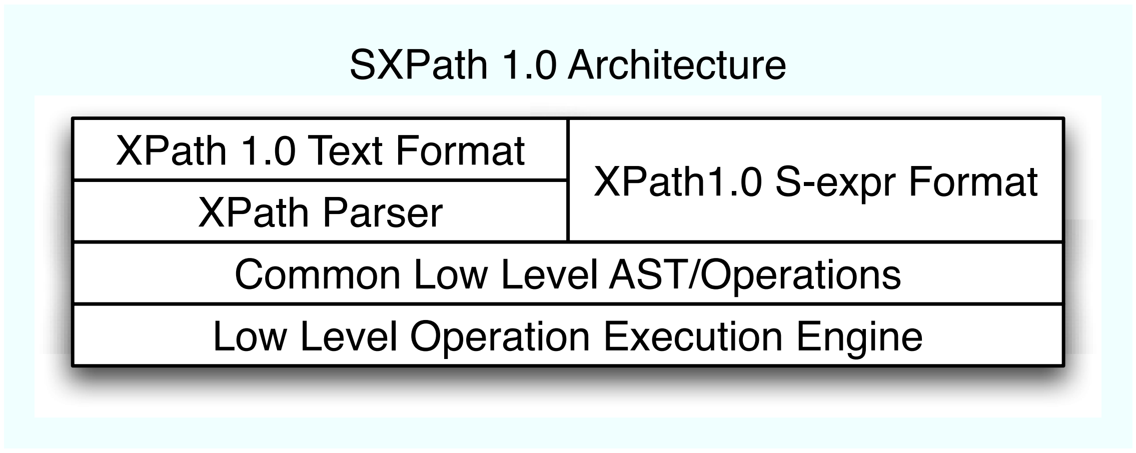

Figure 6: SXPath Architecture

SXPath like Skuery, processes both the original plain text form of the language as well as a S-expression encoded form. The S-expression form for SXPath denotes paths as nested lists of symbols representing XML element and attribute names. Both initial forms of XPath, plain text and S-expression based, are translated by SXPath into a sequence of low-level primitive operations.

Related Syntax	XPath Axis	Function	Description
ELEMENT-NAME	::child	select-kids	selects child elements that have ELEMENT-NAME or satisfy a user-defined predicate
ELEMENT-NAME	::descendant	node-closure	selects all descendants that child elements that have ELEMENT-NAME or satisfy a user-defined predicate

Figure 7: Example SXPath Low Level Functions

Further functions such as node-join, node-reduce, node-union, and filter provide the core functionality of the primitive SXPath operations. The elegance of SXPath and XPath in general is the realization that each step in the path can be represented as a projection, selection, transitive closure or a filter operation. An XPath expression can then be evaluated by combining steps as chains of sequential operations or as parallel union operations.

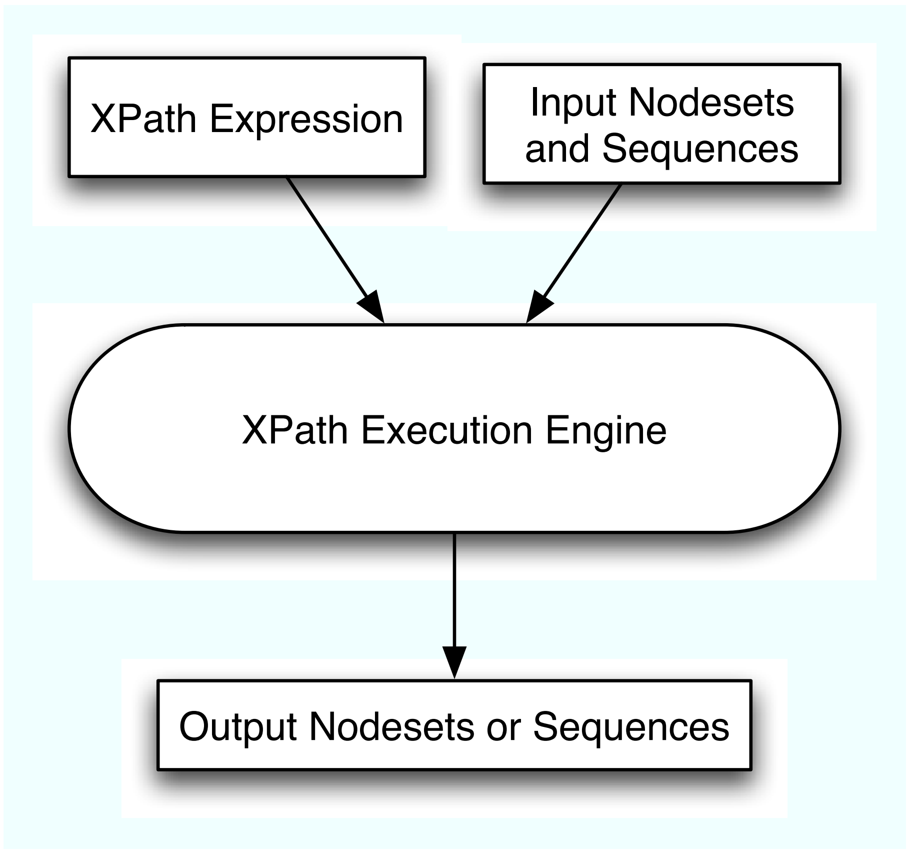

Figure 8: SXPath Architecture

Unfortunately the Scheme SXPath library only supports XPath 1.0 functionality. However the XPath 1.0 and 2.0 standards share the same low-level XML path traversal, selection, and iteration operations as their basis.

Skuery implements XPath 2.0 expressions present in XQuery 1.0 traditional syntax by transforming XQuery AST segments into a super SXPath 1.0 S-expression syntax called KSXPath 2.0. Likewise when writing Skuery queries, KSXPath S-

Expression syntax is used. KSXPath S-Expression syntax can also be used independent of Skuery as a native Scheme syntax for XPath 2.0.

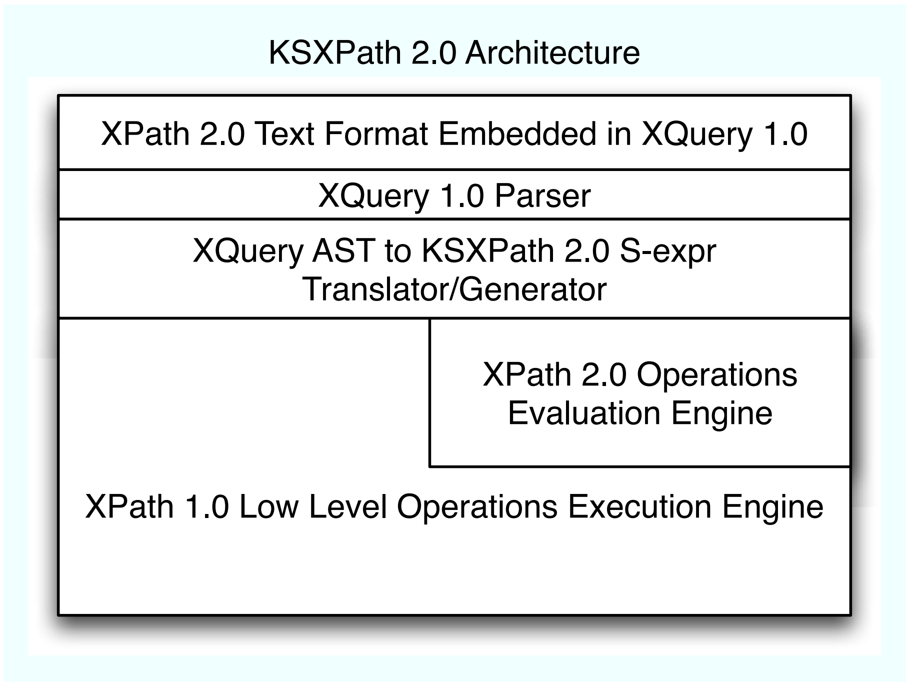
KSXPath 2.0 Architecture
XPath 2.0 Text Format Embedded in XQuery 1.0
XQuery 1.0 Parser
XQuery AST to KSXPath 2.0 S-expr Translator/Generator
XPath 2.0 Operations Evaluation Engine
XPath 1.0 Low Level Operations Execution Engine

Figure 9: KSXPath 2.0 Architecture

KSXPath reuses the low-level implementation functions of SXPath and injects its own combinators into the evaluation pipeline to implement XPath 2.0 functionality. XQuery 1.0 / XPath 2.0 permits node filtering using either a plethora of predefined functions or user-defined functions. Of course these functions must be defined in XQuery. Skuery extends this functionality by allowing arbitrary Scheme functions to be used as XPath filtering functions as long as they abide by the functional contracts specified by KSXPath. Filter functions, being predicates, have a very simple functional contract. They take a single node as an argument and return either true or false depending on whether or not the predicate represented by the filter is satisfied.

```
<People>
  <Person Name="George" Age=25 />
  <Person Name="Mike"/>
  <Person Name="Jane" Age=23 />
  <Person Name="Jill" Age=22 />
</People>
```

Listing 10: Expected query result from XQuery execution.

Given the XML document in *Listing 10*, the XPath expression `"/Foo/Bar/@Baz"` would be represented as `(Foo Bar @ Baz)` in SXML and would return the node set `[ Name="George", Name="Mike", Name="Jane", Name="Jill" ]`. A filter, such as return all the `People/Person` elements that have an age attribute, would be constructed as `"/People/Person[@Age]"` using the plain XPath text encoding or as `(People (Person @Age ))` using SXPaths S-expression encoding. In the S-expression encoded version, the outer parentheses demarcate the entire XPath expression. The inner set of parentheses demarcate a filter operation where only Person elements, which have an attribute named Age, are returned.

KSXPath extends the function calling syntax of SXPath to support the execution of user-defined filter functions in XPath 2.0 expressions. `"/People/Person [empty(@Age)]"`, which returns all `People/Person` elements without an age attribute would be encoded as `(People (Person ((empty (@Age )))))` in KSXPath. Similarly `"/People/Person[two_vowels(@Name)]"` or `(People (Person (( two_vowels (@Name )))))` might return `People/Person` elements whose name attribute value has two vowels. In this last example, using KSXPath, `two_vowels` could be a business logic filter predicate defined in Scheme instead of XQuery.

Complete implementation of XPath 2.0 requires the use of a contextual state object to store current traversal location information. During path execution, XPath 2.0 can call and execute user-defined functions acting as node selectors, filter predicates, etc. User defined XPath 2.0 functions in turn can call sub-path contextual expressions, which are dependent upon the current traversal position of parent calling traversals. A dynamic path context is usually employed to maintain this state information. SXPath 1.0 has no notion of a dynamic context, and hence Skuery's KSXPath implementation is deficient in this area of XPath 2.0 support.

SXPath lack of support for XPath 2.0 features made it unsuitable for use in Skuery. KSXPath adds the needed support of user-defined XPath 2.0 functions by building upon XPath 1.0 functionality taken from SXPath. Skuery's contribution of KSXPath 2.0 improves SXPath's XPath 2.0 conformance and enables broader support of XQuery semantics.

4.4 FLWOR Implementation

Skuery implements FLWOR using syntax-case Scheme macros. The Skuery FLWOR syntax, *Listing 11*, is a single block of imperative commands. The top level `sxquery:FLWOR` form delineates the FLWOR block. The first terms inside a FLWOR block are usually a list of `sxquery:for` terms, `sxquery:let` terms, or both.

```
(sxquery:FLWOR
 ((sxquery:for b '(1 2 3 4 5 6 7 8 9 10 11 12)))
 (sxquery:where (eq? 0 (modulo b 2)))
 (sxquery:return (* b b)))
```

Listing 11: FLWOR Block Example

```
(sxquery:FLOWR
 ((sxquery:for b '(1 2 3 4 5 6 7 8 9 10 11 12))
  (sxquery:for c '(1 2 3 4 5 6 7 8 9 10 11 12)))
  (sxquery:orderby (- 12 b)))
 (sxquery:return (list b c)))
```

Listing 12: FLWOR Block Example with two For Clauses

The `sxquery:for` and `sxquery:let` forms are analogous to generators in the list comprehension model of computation. Following the `for` clause in the *Listing 11* is a `where` clause which provides FLWOR with filtering capabilities which are analogous to the filtering qualifiers of the list comprehension model. The last form in the two examples above is the `sxquery:return` form, which specifies the returned result or generated tuple for each iteration through the loop. Borrowing from SQL terminology, the return form describes the format of each row in the result set. The return clause is analogous to the expression clause of the list comprehension model.

Continuing to barrow from the SQL relational model, FLWOR statements can have an optional `sxquery:orderby` clause, which allows ordering based on user-defined expressions. Skuery order by operations are specified inside the scoping block of enclosing `for` and `let` blocks, allowing ordering on terms and expressions that may not even be present in the final result set. Skuery deviates from the XQuery specification by executing the where and order by clauses inside the FLWOR loop where XQuery specifies that they should be evaluated before evaluating the return clause for any tuple in the processing stream.

Listing 11 and *Listing 12* demonstrate, in the most primitive form, the ability to use functions and forms from Scheme's runtime library inside data query expressions such as `FLWOR`. `eq?`, `modulo *`, `-`, `list`, and `quote(')` are all Scheme runtime functions.

The `sxquery:FLWOR` statement macro is executed recursively in order to evaluate each clause. Each recursive evaluation peels off and evaluate a clause, and the proceeds to recurse on the remaining clauses. There are two variations of the for clause that are almost nearly identical except for the `prev2` argument present in the second variation. The first variation is used when the for clause is the top level or first called FLWOR clause in the recursive evaluation. The second variation is used when the for clause is nested inside a parent for or let clause. In this second variation, `prev2` represents the partial result stream up to the current point in execution. `foldl` or left fold is used as the looping or iteration construct for Skuery's FLWOR implementation. `foldl` is tail recursive, using constant stack space during execution. The `for-loop` term is simply a lambda abstraction that binds the iterator variable for each execution of the loop. `expr` is the value generator for the sxquery:for clause. It generates a list of values, one of which is bound to the iterator variable for each pass through the `sxquery:for` clause. The `sxquery:for` clause has two other post-processing features of interest. Because of the use of space conservative `foldl`, the return sequence of results is in opposite order of evaluation. The `reverse-unless-node` assures that the returned result is in fact a sequence and then calls reverse to restore the sequence to its correct order. If the returned result is just a singleton XML node, reverse is not called and the result is returned untouched. `sort-result` is the other post processing step. If the FLWOR statement contains an `sxquery:orderby` clause, the result stream contains a sequence of `orderresult` structures instead of just the pure results of each iteration. `sort-result`

The `sxquery:let` clause is really just a `sxquery:for` clause striped of it iteration ability. `sxquery:let` simply provides variable bindings through the use of lambda abstractions.

The `sxquery:where` clause is just a simple filter implemented using Schemes standard `if` form. If the filter evaluates to true nested FLWOR expressions and clauses are executed and added to the result set. If the filter evaluates to false the `sxquery:where` clause returns the current result stream without appending to it.

```
(define-syntax (sxquery:FLOWR stx)
  (syntax-case stx
    (sxquery:for sxquery:where
     sxquery:return sxquery:orderby ORDERBYCLAUSE)

    ((_ ((sxquery:for variable expr) rest2 ...) rest ...)
     (syntax
       (sort-result (reverse-unless-node
         (let ((for-loop (lambda (variable prev)
                 (sxquery:FLOWR prev (rest2 ...) rest ...))))
             (foldl for-loop '() expr))))))

    ((_ prev2 ((sxquery:for variable expr) rest2 ...) rest ...)
     (syntax
       (reverse-unless-node
         (let ((for-loop (lambda (variable prev)
                 (sxquery:FLOWR prev (rest2 ...) rest ...))))
             (foldl for-loop prev2 expr)))))
```

Listing 13: Skuery FLWOR For Clause Implementation.

```
    ((_ ((sxquery:let variable expr) rest2 ...) rest ...)
     (syntax
       (singlefy (let ((variable expr))
                 (sxquery:FLOWR () (rest2 ...) rest ...)))))

    ((_ prev2 ((sxquery:let variable expr) rest2 ...) rest ...)
     (syntax
       (let ((variable expr))
         (sxquery:FLOWR prev2 (rest2 ...) rest ...))))
```

Listing 14: Skuery FLWOR Let Clause Implementation.

```
    ((_ prev (sxquery:where test) rest ...)
     (syntax
       (if test
           (sxquery:FLOWR prev rest ...)
           prev)))
```

Listing 15: Skuery FLWOR Where Clause Implementation.

The `sxquery:orderby` clause is a little confusing at first glance. When a `sxquery:orderby` clause is present, the code in *Listing 16* gets executed for each iteration. Further inner clauses and expressions are executed and the intermediate result is stored in the temporary variable named `result`. `ORDERBYCLAUSE` is a recursive macro that creates a list of sorting values for each result. It is an internal

```
((_ prev (sxquery:orderby expr props orderings ...) rest ...)
 (syntax
   (let ((result (sxquery:FLOWR rest ...))
         (orderclause
           (sxquery:FLOWR ORDERBYCLAUSE expr props orderings ...)))
       (if (null? result)
           prev
           (cons (make-orderresult orderclause result (quote props))
                 prev)))))
((_ ORDERBYCLAUSE expr props orderings ...)
 (syntax (cons expr (sxquery:FLOWR ORDERBYCLAUSE orderings ...))))
((_ ORDERBYCLAUSE )
 (syntax ()))
```

Listing 16: Skuery FLWOR Order By Clause Implementation.

utility function to the `sxquery:FLWOR` macro. `ORDERBYCLAUSE` is never seen nor written by a normal application developer. There is a sorting value for each defined ordering e.g. (primary, secondary, tertiary). Finally an `orderresult` structure is created from the list of sorting values and the actual result to be returned. This structure is then appended onto the result stream.

Listing 17 contains three functions that accomplish the actual work of sorting. `make-sorter` creates the comparator function that is feed as a parameter to the higher-order `mergesort` function. `sortorderresult` executes the sorting on the stream of `orderresult` structs and then projects the embedded FLWOR results using map and the `orderresult-result` member selection function. The `sort-result` function, as described above just tests to see if the result stream is a stream of `orderresult` structures, if so it passes control to the sorting functions. Otherwise `sort-result` just returns the result stream as is.

The `sxquery:return` clause is the simples of all the FLWOR clauses. It just evaluates `expr`, the body of the FLWOR statement, and returns the results. Null results or empty sequences in XQuery terminology are filtered out by the second

```
(define-struct orderresult (orderclause result props))
(define (make-sorter x)
  (let ((x (take x 3)))
    (cond
     ((equal? x '(ASCEND #t GREATEST))
       (lambda (x y) (let ((x (orderresult-orderclause x))
                            (y (orderresult-orderclause y)))
         (if (null? x) #t (if (null? y) #f (sxq:< x y))))))
     ((equal? x '(ASCEND #t LEAST))
      (lambda (x y) (let ((x (orderresult-orderclause x))
                           (y (orderresult-orderclause y)))
         (if (null? x) #f (if (null? y) #f (sxq:< x y))))))
     ((equal? x '(DESCEND #t GREATEST))
      (lambda (x y) (let ((x (orderresult-orderclause x))
                           (y (orderresult-orderclause y)))
         (if (null? x) #t (if (null? y) #f (sxq:> x y))))))
     ((equal? x '(DESCEND #t LEAST))
      (lambda (x y) (let ((x (orderresult-orderclause x))
                           (y (orderresult-orderclause y)))
         (if (null? x) #f (if (null? y) #t (sxq:> x y))))))
     (else (error "Hmmmm" "FORMAT␣DOES␣NOT␣MATCH␣~a" x)))))
(define (sortorderresult x)
  (let ((sorter (make-sorter (orderresult-props (car x)))))
    (map orderresult-result (mergesort x sorter))))
(define (sort-result x)
  (if (and (pair? x) (orderresult? (car x)))
    (sortorderresult x)
    x))
```

Listing 17: Skuery FLWOR Order By Utility Functions.

```
   ((_ (sxquery:return expr))
    (syntax expr))
   ((_ prev (sxquery:return expr))
    (syntax (let ((result expr))
      (if (null? result)
         prev
         (cons expr prev)))))
   ((_ prev () rest ...)
    (syntax
      (sxquery:FLOWR prev rest ...)))
   ((_ rest ) (syntax rest ))))
```

Listing 18: Skuery FLWOR Return Clause Implementation.

variant of the `sxquery:return` clause. The last two clauses show in *Listing 18*, are just the recursive termination rules for the `sxquery:FLWOR` macro.

The `sxquery:some` or existential qualification operator in *Listing 19* is simply a recursive function that implements logical or semantics. If `expr` evaluates to true with variable bound to a value from sequence, the existential qualifier returns true. Otherwise the operator returns false.

Universal qualification, `sxquery:every` in *Listing 20*, is implemented in the same form as existential qualification except that all the #f are inverted to #t, all the #t are inverted to #f, and the (if (expr) and (if (sxquery:every are negated to become (if (not (exr) and (if (not (sxquery:every. This simple transformation turns the logical or semantics of sxquery:some into logical and semantics for `sxquery:every`.

```
(define-syntax (sxquery:some stx)
  (syntax-case stx (sxquery:satisfies)
    ((_ ((variable sequence)) sxquery:satisfies expr)
     (syntax
       (let loop ((seq sequence))
         (if (null? seq)
           #f
           ((lambda (variable)
              (if (expr)
                #t
                (loop (cdr seq)))) (car seq))))))

    ((_ ((variable sequence) rest ...) rest2 ...)
     (syntax
       (let loop ((seq sequence))
         (if (null? seq)
           #f
           ((lambda (variable)
              (if (sxquery:some (rest ...) rest2 ...)
                #t
                (loop (cdr seq)))) (car seq)))))))))
```

Listing 19: Skuery FLWOR Some Clause Implementation.

```
(define-syntax (sxquery:every stx)
  (syntax-case stx (sxquery:satisfies)
    ((_ (variable sequence) sxquery:satisfies expr)
     (syntax
       (let loop ((seq sequence))
         (if (null? seq)
           #t
           ((lambda (variable)
              (if (not (expr))
                #f
                (loop (cdr seq)))) (car seq))))))

    ((_ ((variable sequence) rest ...) rest2 ...)
     (syntax
       (let loop ((seq sequence))
         (if (null? seq)
           #t
           ((lambda (variable)
              (if (not (sxquery:every (rest ...) rest2 ...))
                #f
                (loop (cdr seq)))) (car seq)))))))))
```

Listing 20: Skuery FLWOR Every Clause Implementation.

4.5 Using quasiquotes for escaping from SXML literals into XQuery Expressions

Scheme's quoting operators quote ('), quasiquote (‘), unquote (,) and unquote-splicing (@,) proved essential in implementing Skuery. One of XQuery's prominent features is the presence of literal XML as first class syntax in the XQuery grammar. Literal XML as a primitive data type allows for very natural and intuitive construction of XML fragments. Much like Perl or Ruby's string interpolation, XQuery expressions can be embedded in literal XML allowing for dynamic content generation.

Listing 21 demonstrates how quasiquoting permits the escaping and embedding of language expressions inside the literal XML syntax of SXML.

```
<person id=12 name="George Tanner" age="{25 + 24 / 2}">
  <parents>{findParents()}</parents>
  <siblings>{findSiblings()}</siblings>
</person>

`(person (@ (id 12) (name "George Tanner")
            (age ,(+ 25 (/ 24 2))))
  (parents ,(findParents))
  (siblings ,(findSiblings))
)
```

Listing 21: Quasiquote escaping of XQuery expression markers {}.

4.6 XQuery Conformance Tests

While Skuery was closely modeled after the semantics of XQuery, strict bug for bug or feature for feature conformance to XQuery specification was not a Skuery objective. Skuery has aimed to support and adopt a large portion of XQuery functionality. Another one of Skuery's goals was the ability to translate native XQuery syntax into Scheme executable S-expressions. To that end, sections of the XML Query Test Suite were executed against Skuery during development to ensure correctness of functions and operators.

4.7 Control XQuery Implementations

4.7.1 Saxon

Saxon (http://saxon.sourceforge.net/) is a compliant implementation of XQuery 1.0, XSLT 2.0, and XPath 2.0 written in Java. A .Net port is also available by cross-compiling the Java code to MSIL using the IKVMC compiler. Michael Kay is the principle developer behind Saxon. Saxon comes in two flavors: Saxon-B, which implements the basic conformance levels of XSLT 2.0 and XQuery and Saxon-SA, which is XML Schema aware. Saxon-B is an open source product, while Saxon-SA is a commercial product sold by Saxonica (http://www.saxonica.com/). Saxon is an optimizing XQuery interpreter that uses rewriting concepts from relational algebra to significantly improve query performance. Saxon-B is the principle control implementation of XQuery with which Skuery is compared.

4.7.2 WebIt!

WebIt! is an XML manipulation library for Scheme, developed by Jim Bender. The primary features of WebIt! are XML pattern matching and a macro based XML transformation system. Jim Bender created a very primitive prototype of FLWOR expressions to be used with his WebIt! framework. WebIt!'s FLWOR implementation is incomplete and no where near robust enough to run the XMark Benchmark suite. However, being an example of integrated use of FLWOR with Scheme code a few custom benchmarks where create to compare performance with Skuery.

5 Results and Analysis

The performance results and an example use case, *Section 5.4*, demonstrate the accomplishment of the thesis statement. Skuery performance is shown to be competitive even as a prototype implementation. The CDDB example use case demonstrates the power of language native data query operations. Skuery is able to execute FLWOR statements drawing input data from heterogeneous source locations. The data query operations in Skuery appear as just another transformational step through the program. Last of all Skuery's CDDB implementation compared against a corresponding Java implementation.

A couple initial tests were conducted, in an attempt to understand the start up overhead of the different FLWOR implementations. In the first initial test, execution time was measured for one of the simplest queries possible, returning the constant 1. During experimentation Skuery and WebIt! ran in the PLT Scheme v301.16 virtual machine, while Saxon executed inside the Java 1.5.0.06 JVM. By just returning the constant 1, a feel for the overhead of loading and initializing the respective VMs was obtained. Also included in this first test was the time to load the query modules, parse the query, evaluate the constant "1", and return the result.

As shown in *Figure 5*, WebIt! had the smallest start up costs followed by Saxon and then Skuery. While WebIt! and Skuery execute on the same virtual machine, Skuery is much larger and comprehensive than WebIt! in terms of implemented features, quantity of modules, and raw code size. Times for launching the Java and Scheme VMs are also shown for comparison. The other initial test conducted was a comparison between the XML parse times of Skuery and Saxon for various XMark test sizes. Because it uses the same XML parser as Skuery, WebIt! was not

	VM Startup Time Seconds
Skuery	2.033
Saxon	1.043
WebIt!	0.892
Only Java	0.396
Only PLT Scheme	0.119

Figure 10: Startup Overhead

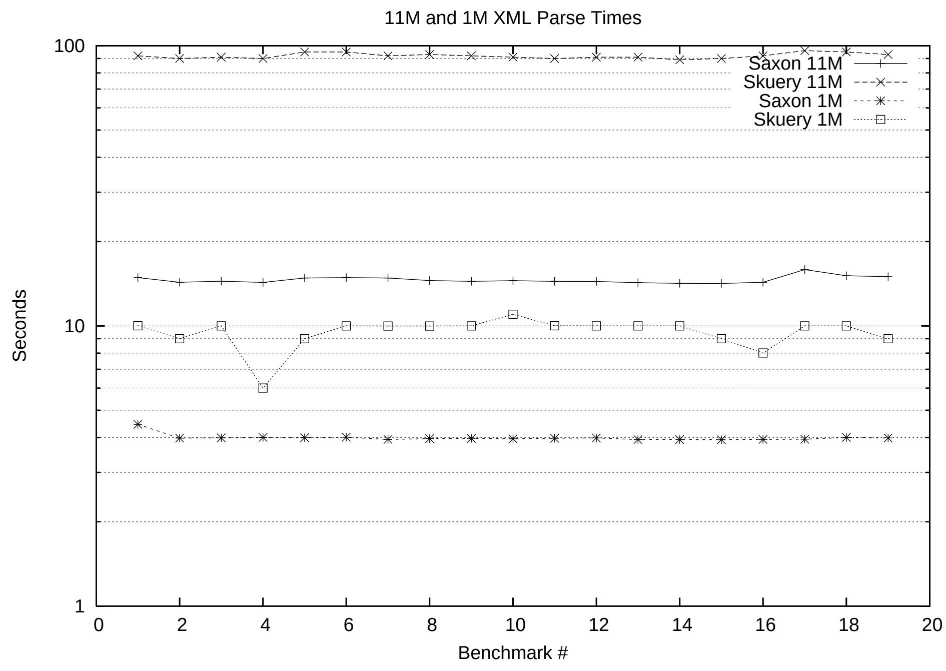

Figure 11: Saxon Skuery Parse Time - 5 Runs

included. *Figure 11* contain the parse time for Saxon and Skuery for both the 11M and 1M XMark data samples.

5.1 WebIt! Results

Some slight massaging was required to get WebIt!'s FLWOR prototype to run with current versions of WebIt! and PLT Scheme. WebIt!'s FLWOR prototype came with three example queries. The examples access a parts catalog database, consisting of three simple tables: catalog.xml, parts.xml, and suppliers.xml. All three of the queries perform a join operation across all three tables. The example tables where too small to provide significant query times. The time to parse the input XML documents along with the noise of VM start up and tear down was much larger than the time require to execute the actual example queries. To remedy the small table sizes source documents were parsed once and then each example query was executed 10,000 times.

The first comparison result, *Figure 12*, shows the performance of Skuery and WebIt! on the three example queries provided with the WebIt! FLWOR prototype. The gap between Skuery and WebIt! widens a bit from 13% on first two examples to close to 18% on the last example. Both wall clock (real) time and user time are shown in the graph.

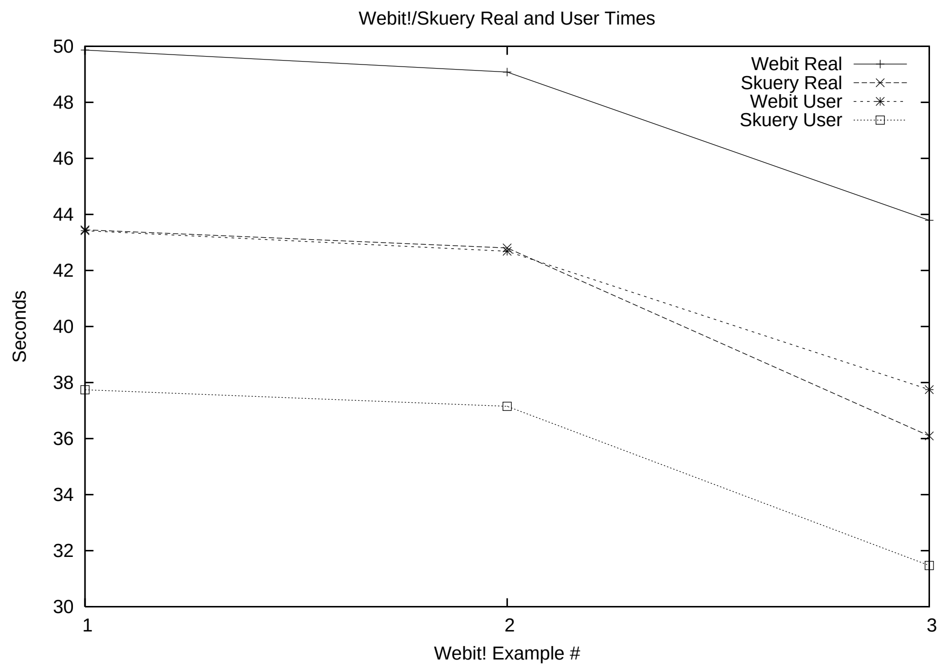

Figure 12: WebIt! vs Skuery Real and User Runtimes

5.2 XMark: An XQuery Benchmark

The database research community has created several new benchmarks for XML Queries in order to characterize the XML performance of both query execution and storage mapping techniques. The intent of the XMark designers was to create a concise, yet comprehensive set of benchmark queries. XMark covers the major features of XQuery including textual features, data analysis queries, and ad hoc queries. The developers of XMark cite query performance and the XML storage organization as the metrics of successful XML implementations and have built their benchmark to characterize the metrics.

XML Query processors have been implemented on top of a wide variety of data stores. XMLDB from Sleepycat Software is built upon the BerkeleyDB flat file format. Products from Oracle, IBM and Microsoft use a relational model as the underlying store for XML data. Object oriented databases as well as in memory storage are other alternatives. The varying paradigms of storage and indexing have a large effect on query performance. XMark attempts to stretch conventional DBMS architectures using benchmarks that emphasize textual XML document order, large deviation length strings, and hierarchical path expression queries [33].

One of the features of the XMark benchmark suite is `xmlgen` a document generator that generates test data sets of variable sizes. `xmlgen` takes a scaling factor command line parameter that determines the data size of the generated test data set. The XMark test data set sizes in *Figure 5.2* were used to characterize Skuery's performance. *Figure 5.2* provides a brief description of each benchmark.

Skuery was primarily compared against Saxon, a Java implementation of XQuery. In order to ensure correctness of query results, the outputs from running the XMark benchmarks on each implementation were compared against each other. Unfortu-

Name	Size in Bytes	xmlgen scaling factor
20k	26713	.00001
33k	33924	.0001
110k	116095	.001
1M	1161652	.01
11M	11669748	.1

Figure 13: XMark Data Sets

nately in most cases, textual comparison of XML cannot determine equality. XML outputs can often be ambiguous due to different rendering styles. Unless present in an attribute string value or a CDATA node, white space is usually insignificant and generally ignored. Some XML engines render XML elements without children as `<element_name attr1="value"/>` while others render the same value as `<element_name attr1="value"></element_name>`. Attributes of XML elements are not ordered, so XML producers often render attributes in different orders. All these and other factors complicate testing for equality between two XML fragments. The Perl XML canonicalization module, XML::SemanticDiff, was able to determine XML equality in most cases, except for differing attribute order.

#	Detailed Description	Benchmark Emphasis
1	Returns a list of names of persons who have an id of ”person0”	For loop, XPath selects
2	Returns a list of constructed increase elements containing the increase of the first bid in each open auction.	XML element construction, cost of array lookups
3	Returns the first and last increases in all open auctions where the last increase is less than or equal to two times the first increase.	filtering using where clauses
4	Returns a list of reserves for open auctions where ”person20” bid before ”person51”.	existential qualification, filtering using where clauses
5	Returns the quantity of closed actions that sold for more that 40.	string to integer casting
6	Returns the quantity of items sold on each continent	descendant XPath expressions
7	Returns the quantity of textual descriptions in the auction database	descendant XPath expressions
8	Returns a list of buyers and the number of items they purchased	join between buyers and closed auctions
9	Returns a list of buyers and the name of the items they purchased in Europe.	join between buyers, items, and closed auctions
10	Returns a list of people (reconstructed using french markup) grouped by interest profile.	extensive XML element construction, join between people and interest categories
11	Returns a list of people’s names and the number of items for sale that have a price less that 0.02% of income of that person.	join using calculated values
12	Returns a list of people’s names that have an income greater than 50000 and the number of items for sale that have a price less that 0.02% of income of that person.	join using calculated values
13	Returns a list of items in Australia.	XML element construction
14	Returns all items that have the word gold in the item description	full-text searching
15	Returns a list of emphasized keyword in annotations of closed auctions.	deep non wild card path traversals
16	Returns a list of seller ids that have emphasized keywords.	deep non wildcat path traversals, return shallow portions of the path
17	Returns a list of people that do not have a homepage.	test for the absence of elements
18	Converts the reserves of all open auctions from one currency to another.	efficiency of user-defined functions
19	List of all items ordered alphabetically by their location	sorting

Figure 14: XMark Benchmark Descriptions

5.3 Skuery and Saxon XMark Results

Figures 15 through *24* contain comparison results for Skuery and Saxon using 11M, 1M, 110k, 33k, and 20k XMark data sets The data points for each benchmark in each of the figures are the cumulative total of 5 consecutive runes. The 1M data set, *Figures 15* and *16*, show that for benchmarks 1-7 and 13-19 Skuery, while slower, remains within reach of Saxon. Skuery's performance is acceptable in most regards given its early prototype nature. Saxon is a commercially sold and supported product, developed over a period of years. Saxon also reaps the benefit of the widely used Java platform, which has received thousands of man-years of optimization and tuning. Queries 8-12 are where Skuery fails to measure up. These queries contain nested FLWOR loops that act like database joins. Performant join operations are achieved through query optimizers that use relation algebra, data size characteristic, and schema information to produce optimal query plans. Such a query optimizer was not within the proposal scope of Skuery. However, Skuery's design makes future integration of a query optimizer straightforward. Later results in *Figures 28* and *29* show vast improvements in benchmarks 8-12 by pushing projections outside FLWOR loops. Equivalent performance to Saxon however, will require complex query optimizations only provided by a query optimization engine.

Two graphs are presented for each experimental data set size: 11M, 1M, 110k, 33k, and 20k. All the graphs are identical except for scale and data set size. In each graph the y-axis represents query time measured in seconds and the x-axis is the XMark benchmark identification number. The first figure in each two-figure set has a linear scale y-axis while the second image has a logarithmic scale y-axis. Greater intuitional feel for the data is gained by contrasting the different scaled axes.

Each graph contains the same 4 measurements for both the Saxon and Skuery implementations. The first data series for each implementation measures the total runtime of the particular benchmark. This metric serves as an all-encompassing comparison between Saxon and Skuery. It includes all test startup, teardown, and VM initialization costs.

The second data series measures the total internal query execution time. This series excludes startup, teardown and VM initialization time costs. The third data series is the time to parse the input XMark XML data set. The last data series is the difference between the second and third data series. Labeled as "Internal Minus XML Parse" this data series represents solely the query execution time.

Figure 15 shows that excluding benchmarks 8-12 Skuery's runtimes hover right at 20 seconds while Saxon's hover at 10 seconds. Looking at the *Skuery Internal Minus XML Parse* data series, the data points floats right around 10 seconds. In contrast, *Saxon Internal Minus XML Parse* floats around at about 4 seconds. Skuery follows the Lisp central dogma: everything is a list. However list processing is not the most effective approach to every query. Intelligent choice of data structures during query execution should improve Skuery's performance.

Also of interest is the comparison between the Saxon and Skuery's XML parsers. The SXML parser Skuery uses hangs about 10 seconds while Saxon's parser hovers at 4 seconds. From these results, a portion of Skuery's lagging performance can be attributed to its XML engine, SXML.

Figure 18 illustrates how the bad performance on benchmarks 8-12 in *Figure 15* is exacerbated by an increase in data set size from 1M to 11M. Benchmarks 8, 11, and 12 hover together as expected; they each contain one join operation. The jump between benchmarks 8 and 9 is explained by an additional join operation in benchmark 9. Where benchmark 8 had one join operation across two data sets, benchmark 9 has two join operations across three datasets.

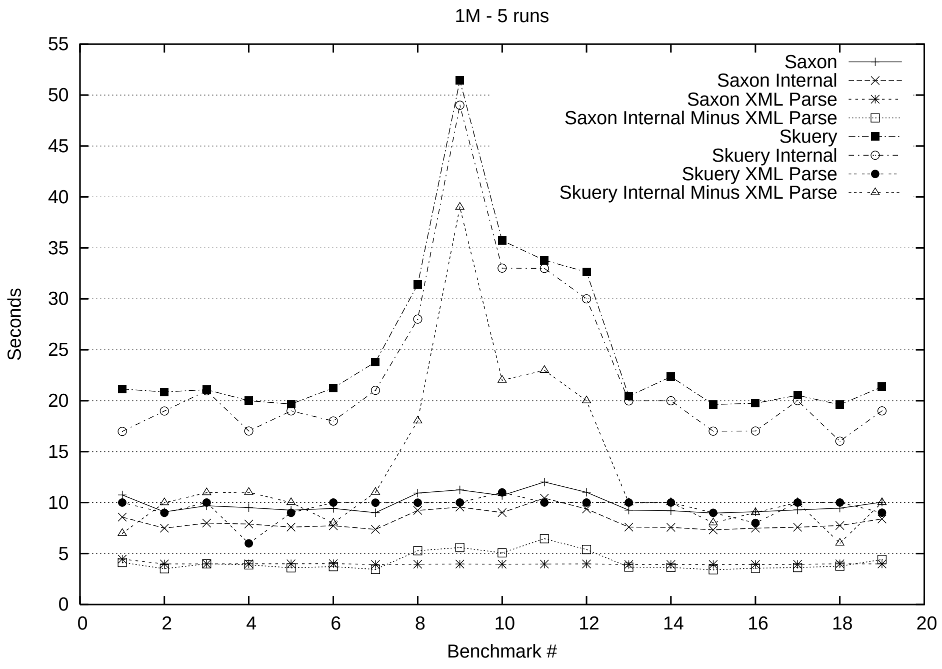

Figure 15: 1M - 5 Runs

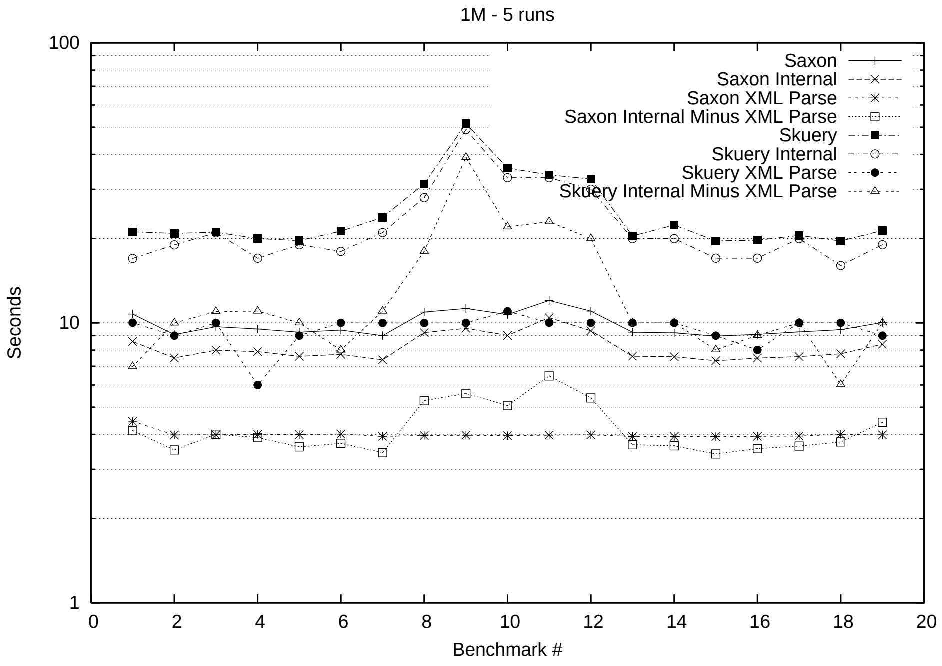

Figure 16: 1M - 5 Runs

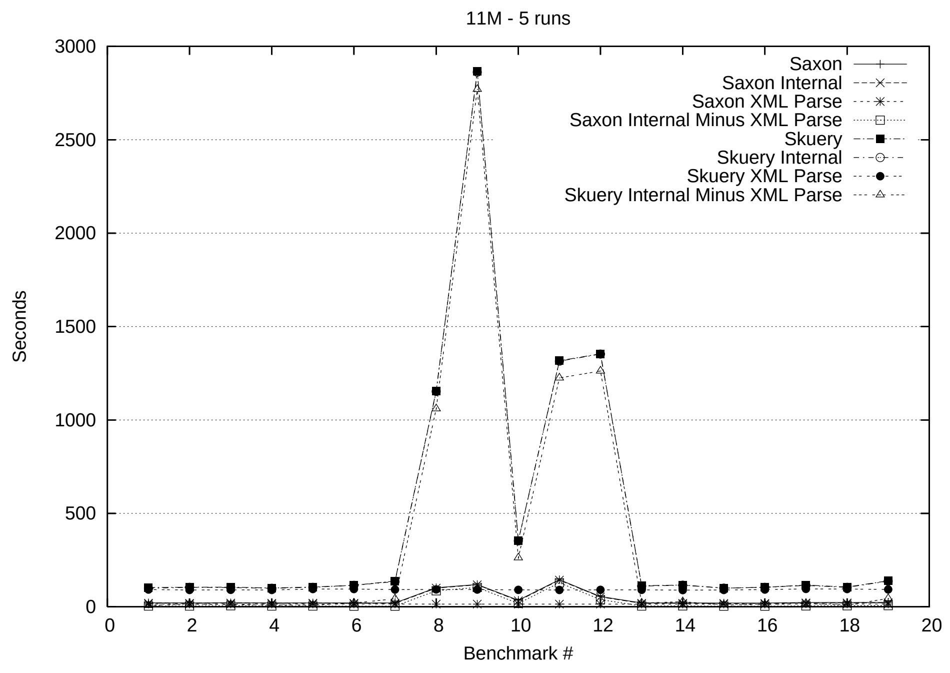

Figure 17: 11M - 5 Runs

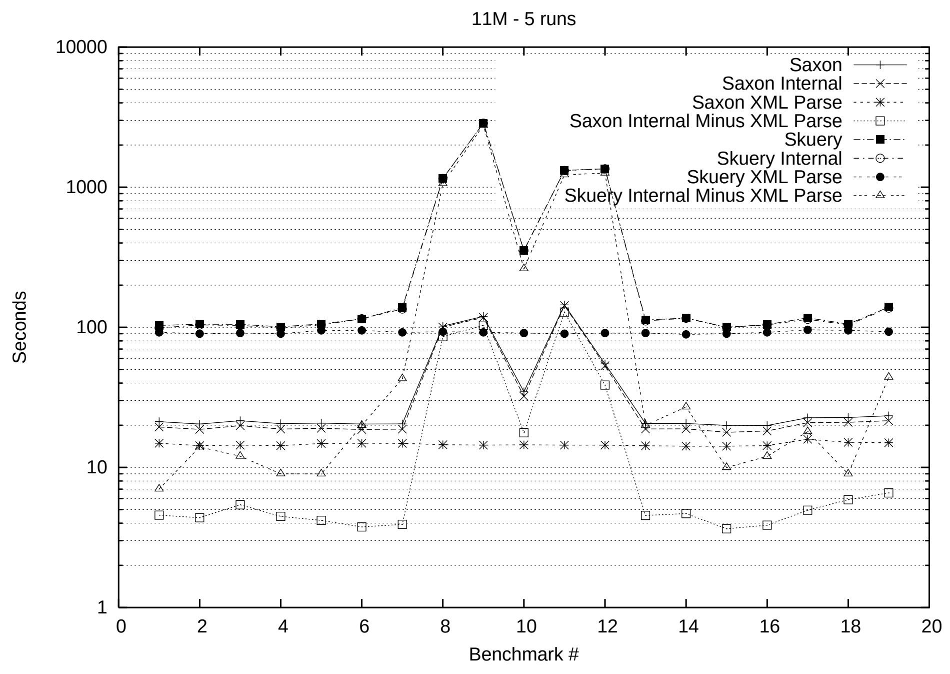

Figure 18: 11M - 5 Runs

Figure 19 shows a slight increase in Skuery run time for queries 7-12. The 110k data set size is barely big enough to show the performance costs associated with queries 8-12. *Figures 21 - 24* for the 33k and 20k data sets do not give any hint to the complexity in queries 8-12. Starting in *Figure 19* and more profoundly in *Figures 21 - 24*, erratic behavior exhibits itself due to the small data set sizes. Noise and overhead begin to out weigh the measured indicators, especially Skuery parse time which becomes very fast for small data sizes. Interestingly, Saxon parse time lands at about 1 second and never decreases below that threshold.

The first attempt at reducing the runtimes of benchmarks 8-12 was to cache intermediate results that were recalculated repeatedly. This attempt did not modify the original queries in any way. The cost of calculating hash values and storing and looking up cached results negated any speed gained by eliminating repeated calculations. In fact runtimes often increased by a small amount.

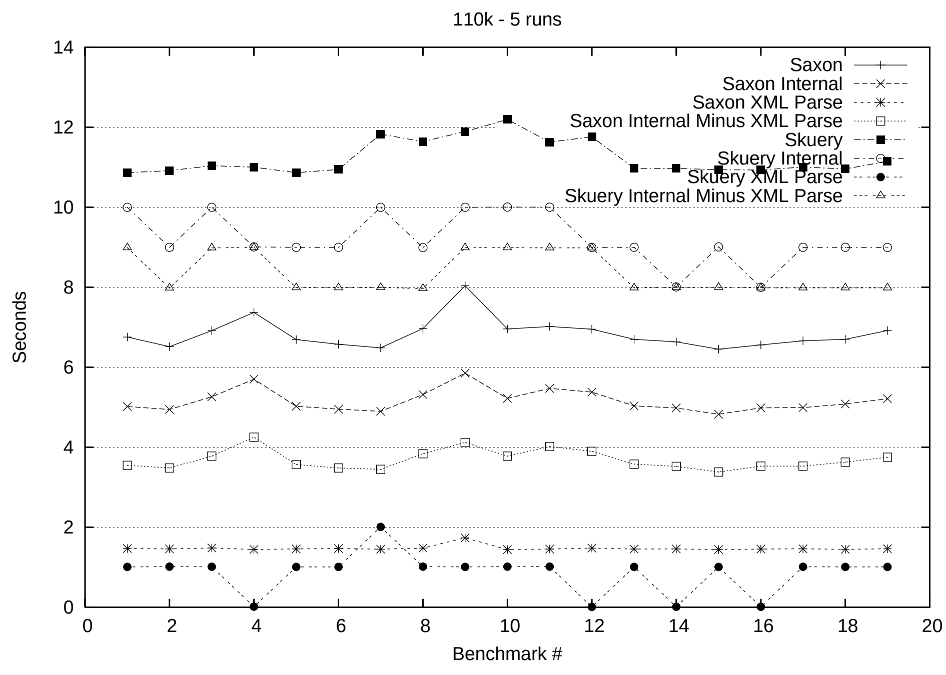

Figure 19: 110k - 5 Runs

Instead of trying to detect and cache intermediate results, the second attempt moved repeated calculations outside of iteration loops. Where possible, projections were moved outside of FLWOR selection loops. Although this optimization was achieved by rewriting the queries by hand, this is a well-known method implemented by automated query optimizers.

Figures 25, 26, and *5.3* show the timing results for unmodified and hand-optimized queries on benchmarks 8-12. It is interesting to note that the hand-optimized queries also improved the Saxon performance of a couple of the benchmarks. While Saxon is obviously rewriting and optimizing queries internally, benchmarks 8, 11, and 12 show that further optimization is possible.

The Skuery timings results from *Figure 25* were divided by the corresponding Saxon timing results to obtain the slowdown factors shown in *Figure 28*.

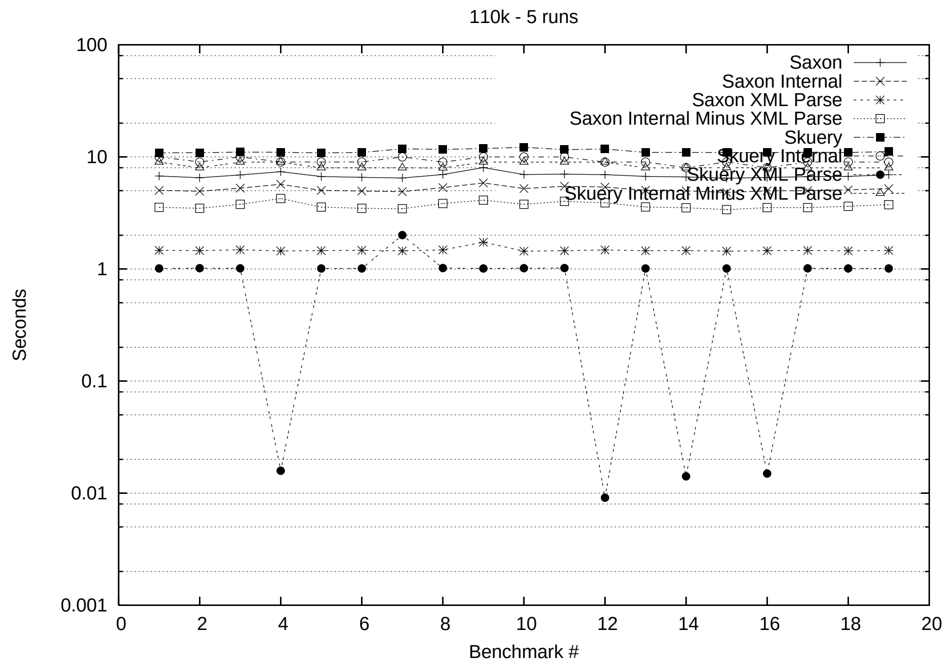

Figure 20: 110k - 5 Runs

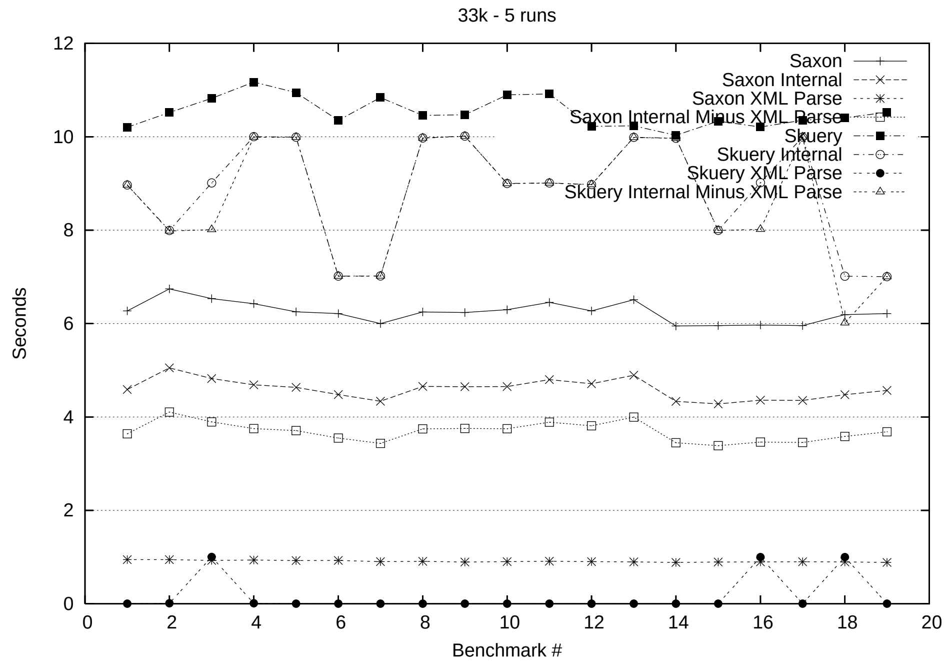

Figure 21: 33k - 5 Runs

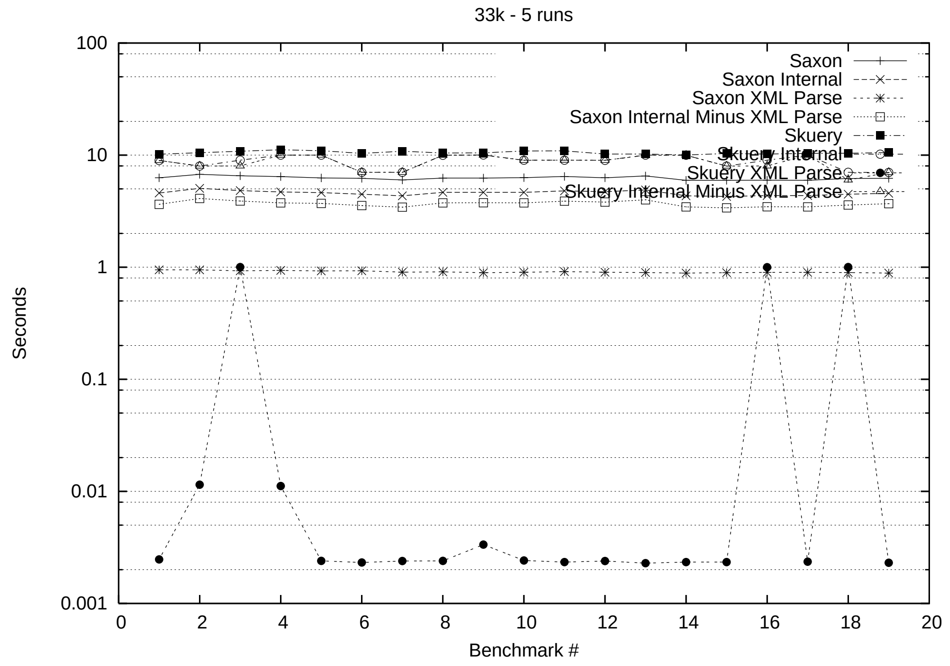

Figure 22: 33k - 5 Runs

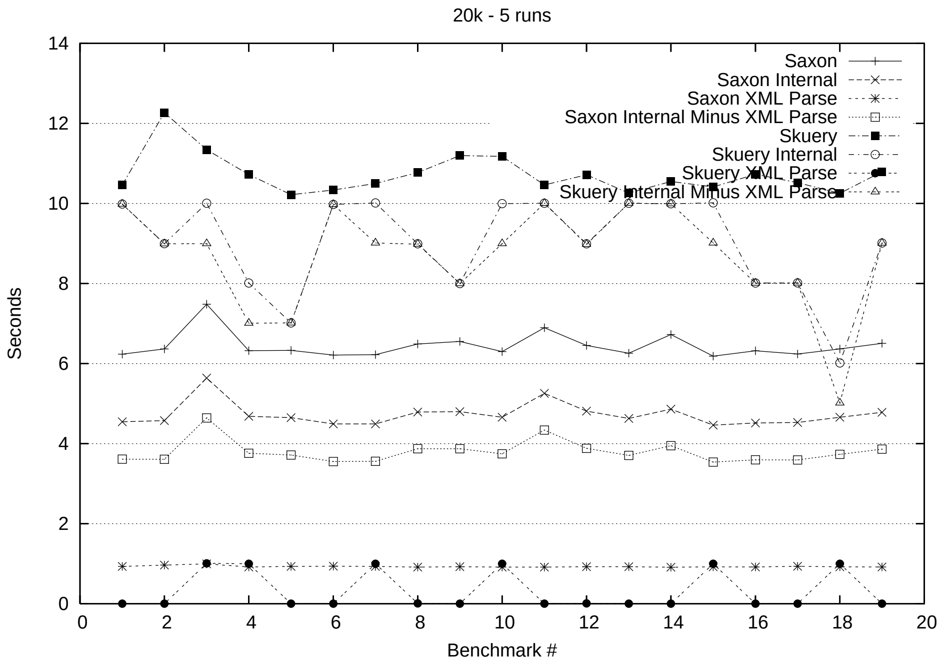

Figure 23: 20k - 5 Runs

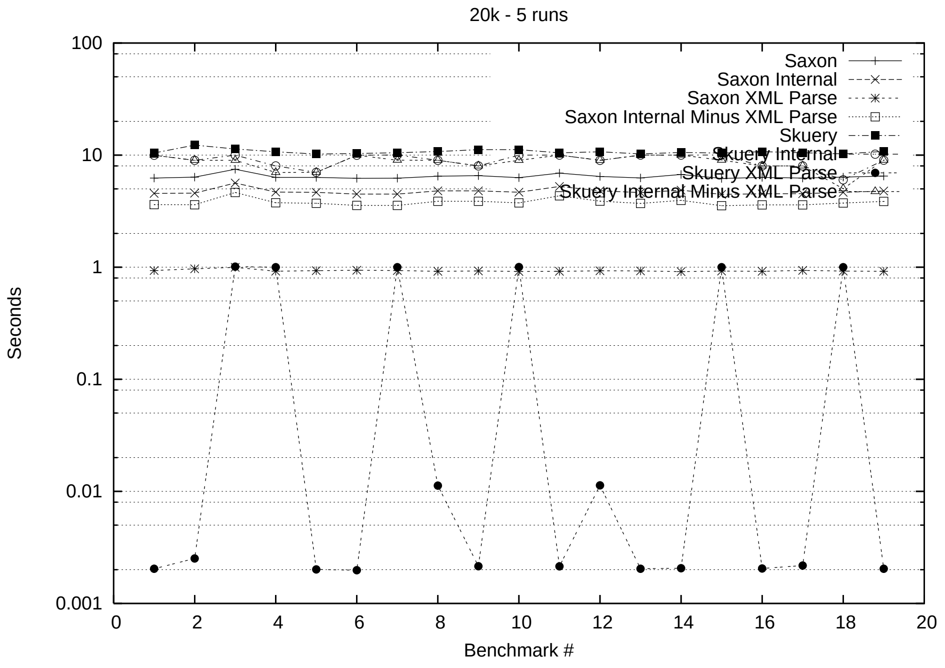

Figure 24: 20k - 5 Runs

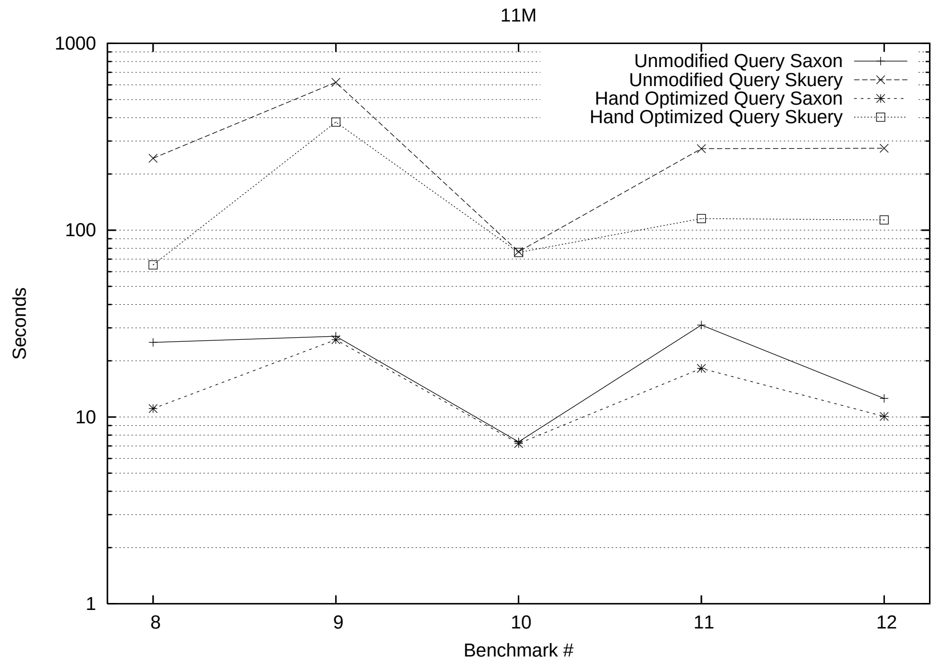

Figure 25: 11M Hand-Optimized Queries

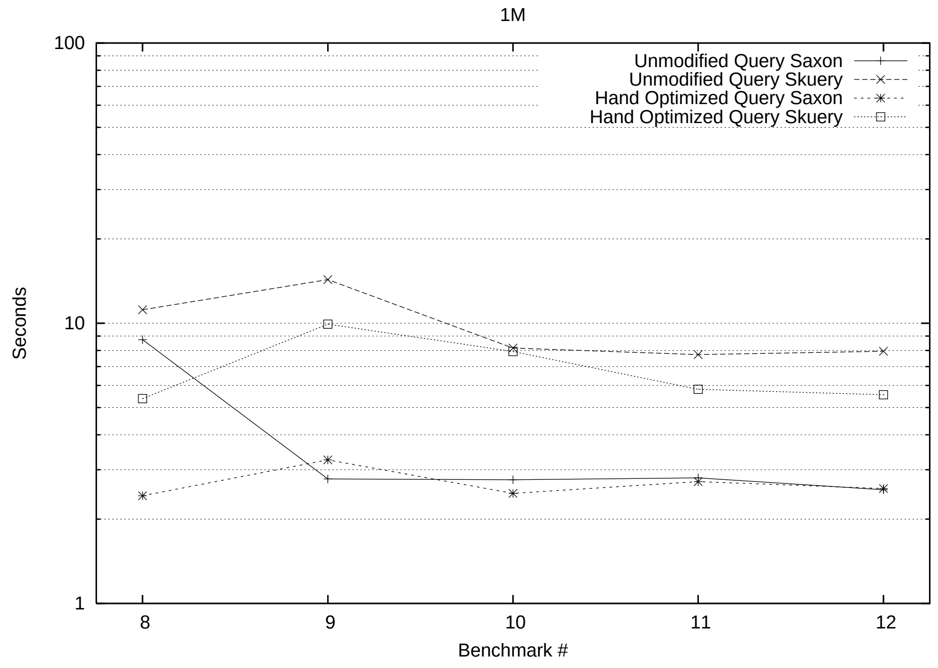

Figure 26: 1M Hand-Optimized Queries

Description	8	9	10	11	12
11M Unmodified Saxon	25.09	27.05	7.37	31.032	12.57
11M Modified Saxon	11.08	25.93	7.21	18.19	10.06
11M Unmodified Skuery	242.51	618.59	76.69	273.08	274.45
11M Modified Skuery	65.15	378.96	75.99	115.37	113.53
1M Unmodified Saxon	8.74	2.78	2.76	2.81	2.54
1M Modified Saxon	2.42	3.26	2.47	2.72	2.57
1M Unmodified Skuery	11.17	14.31	8.16	7.73	7.94
1M Modified Skuery	5.38	9.92	7.93	5.81	5.56

Figure 27: Hand-Optimized Queries

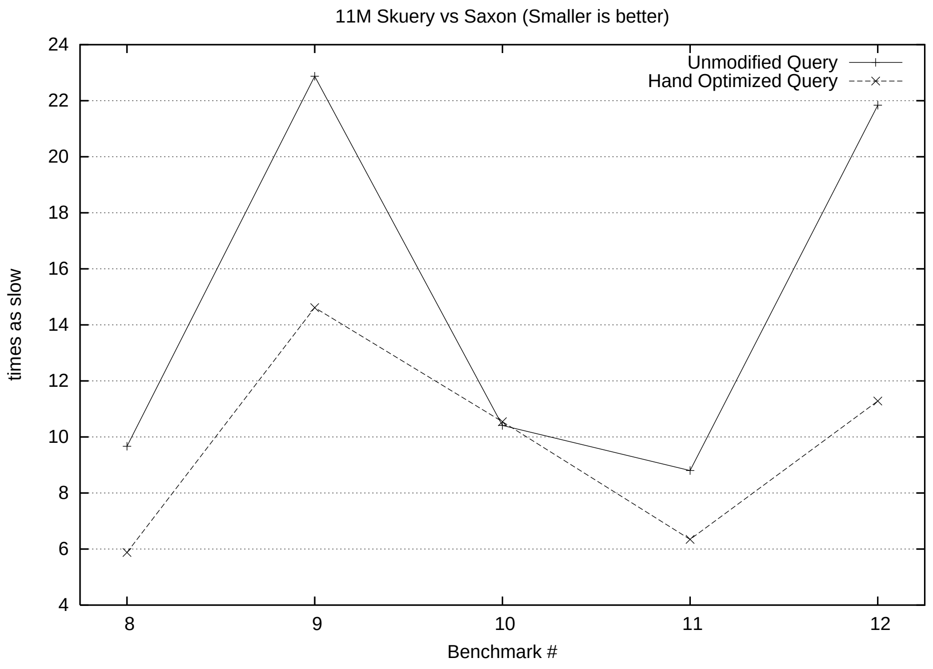

Figure 28: Skuery/Saxon Unmodified vs Hand-Optimized Queries

Figure 29 illustrates the percentage of improvement of hand-optimized queries over the unmodified queries from *Figure 28*. Finally *Figure 5.3* presents the results of *Figure 28* and *Figure 29* in table form for closer inspection.

Realizing that Skuery is an early prototype of integrating data query as a native feature of programming languages, the presented results are promising. Compar-

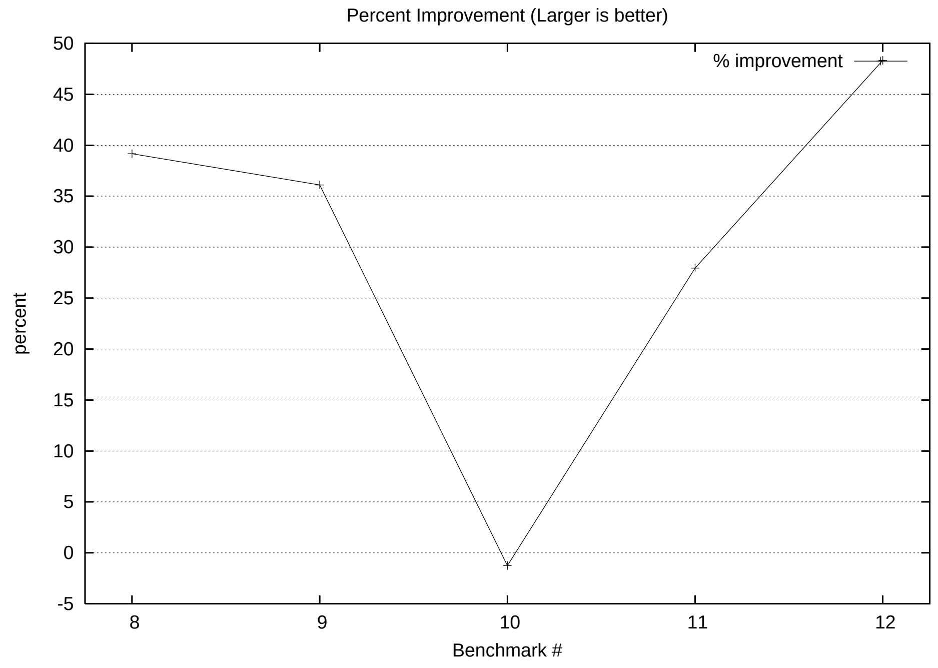

Figure 29: % improvement of Hand-Optimized Queries

Description	8	9	10	11	12
Unmodified Skuery/Saxon	9.66	22.87	10.41	8.80	21.84
Modified Skuery/Saxon	5.88	14.61	10.54	6.34	11.28
% Improvement	0.39	0.36	-0.012	0.28	0.48

Figure 30: Optimized Queries Skuery vs Saxon

isons for non-nested queries between Skuery and Saxon were competitive. Skuery struggled with the nested queries (join operations) in benchmarks 8-12. Not having a built-in query optimizer, Skuery executes these joins as full cross-product joins. While a %30 to %50 improvement is gained by pushing projections outside of FLWOR selections, a $6x$ to $14x$ slowdown is still seen when compared to Saxon. Saxon's performance comes from further optimizing join operations using on the fly indexing and query rewriting which reduces brute force $O(n^2)$ cross-product complexity. Adding well know database optimization techniques to Skuery would close the remaining gap with Saxon.

5.4 CDDB Example using Skuery

This example shows increased programmer productivity gained by using Skuery. Skuery reduces the lines of code needed to implement the query in Java by almost half. Compare *Appendix B* with *Appendix C*. The transparency and native feel of Skuery abstractions enables attention to be focuses on competitive advantages further up the stack.

The CD database (CDDB) is a repository of meta-data for compact discs. Typical stored information includes CD title, artist, and a list of track titles. Ti Kan was the creator of the CDDB idea including using the Internet to give global access and allow anyone to submit entries to the shared database. Steve Scherf continued the work of CDDB by developing cddbd the server side software that powers CDDB. The cddb software was release as open source under the GNU Public License. Users believed that their contributions to the database would also be freely available. In 1998 Gracenote commercialized the CDDB database and changed licensing terms to restrict access to the user-contributed content. In response the user community created freedb.org a CDDB replacement in which both the software and content are protected by the GPL and are freely available to all.

CDDB works by creating a near unique signature from simple characteristics of an audio CD. The signature consists of a 8 digit hexadecimal number generated from the number of tracks on the CD, the starting offset of each track, and the total playing time of the CD in seconds. This signature is then used to query a remote CDDB site, normally over the Internet. The remote CDDB service then replies with the CD title, artist, and a list of track titles.

This example illustrates the use of Skuery query functionality in a simple Scheme CDDB client. Two databases are queried in the example to produce a list of CD tracks that have been listened to more than 30 times by the user. The first database depicted in *Listing 22*, represents the attention data of the user, possibly downloaded from a semi-intelligent personal disc-man. Recoded information includes a signature of the CD in terms of number of tracks, each tracks starting offset, and the total length of the CD in seconds. A listen counter for each track is also maintained in the personal disc-man attention database. Each time the user listens to a track the associated listen counter is incremented.

The second database used in the example is the freedb.org CDDB. A simple Scheme client side library for accessing CDDB was developed to retrieve CD meta-data in an XML format.

Listing 23 is the Skuery query that actually combines the data from the listening attention database and the CDDB database. `attn-source` is the attention database in XML format. The first `sxquery:for` statement loops over each CD in the attention database using `attn-cd` as the current CD in the iteration. The following `sxquery:let` statement retrieves the CDDB meta-data for `attn-cd` into the cddb-data variable. The next three `sxquery:let` statements are just simple projections on `cddbd-data` using XPath expressions. These projected values will be used later in the query. The next `sxquery:for` statement loops over each track in the current `attn-cd`. The subsequent two `sxquery:let` statements project the track number and listen counter for the current `attn-track`. The `sxquery:where` clause filters out tracks that have been listened to 30 or less times. The `sxquery:return` clause builds the track description list. Using the `tracknum` of the attn-track, the letrec statement assigns the corresponding cddb track meta-data information to `cddb-track`. The `tracktitle` is then projected from `cddb-track`.

```
<collection>
  <cd id="900bc00b" tracks="11" seconds="3010">
    <track number="1" offset="183" numberoflistens="10"/>
    <track number="2" offset="25405" numberoflistens="60"/>
    <track number="3" offset="46255" numberoflistens="40"/>
    <track number="4" offset="68468" numberoflistens="80"/>
    <track number="5" offset="88893" numberoflistens="30"/>
    <track number="6" offset="107990" numberoflistens="20"/>
    <track number="7" offset="130000" numberoflistens="90"/>
    <track number="8" offset="143303" numberoflistens="100"/>
    <track number="9" offset="159225" numberoflistens="10"/>
    <track number="10" offset="180455" numberoflistens="50"/>
    <track number="11" offset="202453" numberoflistens="100"/>
  </cd>
  <cd id="b10dec0d" tracks="13" seconds="3566">
    <track number="1" offset="183" numberoflistens="10"/>
    <track number="2" offset="15935" numberoflistens="90"/>
    <track number="3" offset="35195" numberoflistens="70"/>
    <track number="4" offset="51198" numberoflistens="30"/>
    <track number="5" offset="64188" numberoflistens="50"/>
    <track number="6" offset="84758" numberoflistens="80"/>
    <track number="7" offset="113530" numberoflistens="100"/>
    <track number="8" offset="135323" numberoflistens="20"/>
    <track number="9" offset="157313" numberoflistens="40"/>
    <track number="10" offset="186675" numberoflistens="100"/>
    <track number="11" offset="209520" numberoflistens="10"/>
    <track number="12" offset="231050" numberoflistens="60"/>
    <track number="13" offset="250953" numberoflistens="30"/>
  </cd>
  <cd id="b907e20e" tracks="14" seconds="2020">
    <track number="1" offset="150" numberoflistens="90"/>
    <track number="2" offset="10425" numberoflistens="20"/>
    <track number="3" offset="21975" numberoflistens="80"/>
    <track number="4" offset="31575" numberoflistens="60"/>
    <track number="5" offset="42975" numberoflistens="50"/>
    <track number="6" offset="52650" numberoflistens="30"/>
    <track number="7" offset="62850" numberoflistens="10"/>
    <track number="8" offset="77325" numberoflistens="70"/>
    <track number="9" offset="88875" numberoflistens="40"/>
    <track number="10" offset="97425" numberoflistens="100"/>
    <track number="11" offset="109350" numberoflistens="30"/>
    <track number="12" offset="121275" numberoflistens="20"/>
    <track number="13" offset="130800" numberoflistens="50"/>
    <track number="14" offset="140250" numberoflistens="30"/>
  </cd>
</collection>
```

Listing 22: CD Listener Attention Database

Listing 24 shows the result of executing the query in *Listing 23* on the attention data in *Listing 22*.

The CDDB example shows the conciseness and elegance of XQuery semantics in S-expression form. In the Skuery implementation, *Listing 23* and *Appendix B*, of the CDDB the data query operation is transparent, appearing as just another language construct. The Java implementation, *Appendix C*, on the other hand implements XQuery as an external domain-specific language. The XQuery code sentences are embedded in Java's string data type. With the omnipresent low cost of data acquisition, data query operations appear in every facet of business and influence decisions. Skuery seamlessly integrates heterogeneous business data sources with-

```
(require "skuery-runtime.ss")
(require (prefix cddb: "cddb.ss"))
(require (lib "pretty.ss"))
(require (only (lib "1.ss" "srfi") drop))
(require (planet "sxml.ss" ("lizorkin" "sxml.plt" 1 3)))

(define attn-source (doc "cds.xml"))

(pretty-print
 (sxquery:FLOWR
  ((sxquery:for attn-cd ((sxpath "/collection/cd") attn-_source)))
  ((sxquery:let cddb-data (cddb:query-cd-data attn-cd)))
  ((sxquery:let title
    (car ((sxpath "/@title/text()") cddb-data))))
  ((sxquery:let artist
    (car ((sxpath "/@artist/text()") cddb-data))))
  ((sxquery:let tracks ((sxpath "/track") cddb-data)))

  ((sxquery:for attn-track ((sxpath "/track") attn-cd)))
  ((sxquery:let tracknum
    (string->number (car ((sxpath "/@number/text()") attn-track)))))
  ((sxquery:let listens
    (string->number (car ((sxpath "/@listens/text()") attn-track)))))
  (sxquery:where (> listens 30))
  (sxquery:return
    (letrec ((cddb-track (car (drop tracks (- tracknum 1))))))
             (tracktitle (car ((sxpath "/@name/text()") cddb-track))))
    (list title
          artist
          tracktitle
          listens)))
```

Listing 23: Example CDDB Query using Skuery

```
(("Help!" "Beatles" "Dizzy miss Lizzie" 40)
 ("Help!" "Beatles" "Yesterday" 50)
 ("Help!" "Beatles" "Tell me what you see" 40)
 ("Help!" "Beatles" "You like me too much" 100)
 ("Help!" "Beatles" "It's only love" 40)
 ("Help!" "Beatles" "Act naturally" 70)
 ("Help!" "Beatles" "Another girl" 50)
 ("Help!" "Beatles" "I need you" 60)
 ("Help!" "Beatles" "You've got to hide your love away" 80)
 ("Help!" "Beatles" "Help!" 90)
 ("The Joshua Tree" "U2" "Mothers of the disappeared" 100)
 ("The Joshua Tree" "U2" "Exit" 50)
 ("The Joshua Tree" "U2" "Trip through your wires" 100)
 ("The Joshua Tree" "U2" "In God's country" 90)
 ("The Joshua Tree" "U2" "Bullet the blue sky" 80)
 ("The Joshua Tree" "U2" "With or without you" 40)
 ("The Joshua Tree" "U2" "I still haven't found what
    I'm looking for" 60)
 ("THE BEST OF EAGLES" "EAGLES" "PEACEFUL, EASY FEELING" 90)
 ("THE BEST OF EAGLES" "EAGLES" "DESPERADO" 70)
 ("THE BEST OF EAGLES" "EAGLES" "BEST OF MY LOVE" 50)
 ("THE BEST OF EAGLES" "EAGLES" "LYIN' EYES" 80)
 ("THE BEST OF EAGLES" "EAGLES" "TAKE IT TO THE LIMIT" 100)
 ("THE BEST OF EAGLES" "EAGLES" "HOTEL CALIFORNIA" 40)
 ("THE BEST OF EAGLES" "EAGLES" "NEW KID IN TOWN" 100)
 ("THE BEST OF EAGLES" "EAGLES" "HEARTACHE TONIGHT" 60))
```

Listing 24: List of Tracks Listened to more than 30 times

out abandoning the power and comfort of a general purpose programming language.

6 Future Work

6.1 Perpetual and Subscription Queries

Skuery provides the basis for a variety of future research directions. The limitation of bandwidth and size of the Internet has resulted in a data set where the notion of computability must be rethought [3]. The problem is further complicated by the fact that during the time that it takes to crawl the web, much of the content changes. In this environment, questions about the data set as a whole at any given point in time are unanswerable. New simplified query models such as RSS subscriptions and Google alerts provide users with a continuous stream of result sets. Managing and innovating upon this new data framework is one approach to future research.

6.2 Relevance Queries

Adam Bosworth gave the he best description of the Google queries millions of people perform each day, when he called them relevance queries. Relational research and products have focused on optimizing queries that have an exact computable answer. The relational calculus and accompanying operational semantics of relational algebra are the elegant and powerful solutions to such problems. Unfortunately, exact computable answers do not scale to the magnitude of information available from the World Wide Web. Even if scalability were possible, Google users are not interested in the comprehensive computable questions that relational queries have traditionally answered. Instead users are interested in relevance.

Characteristics of a relevance query are distinct from those of traditional relational queries. First, exact repeatability is not required. Need for ad hoc historically repeatability is extremely infrequent. Historical repeatability when important is almost always predictable and known in advance. Periodically archiving query result sets, rather than maintaining re-computability can most often meet such needs. With respect to time, relevance queries often fall into two common categories. First, near real time relevance such as what has been associated with the terms of the query in the past hour, 24 hours, week. Second, long term popularity, query responses that are accepted repeatedly by users.

Relevance information needs to be filtered to suit the querier. But the filtering mechanism is more prepositional than conditional. The relational term `where` should be replaced with `with respect to` when attempting to filter relational data. Common `with respect to` terms include geographical locality, spheres of influence, time periods, employers, buddy lists, blog roles, peer groups, entities, etc. These `with respect to` terms are difficult to index in the relational sense, primarily because there is no authoritative catalog of `with respect to` terms. `With respect to` terms are self-assigned by queriers, most powerfully in the form of tagging. So given the fact that filtering terms of relevance data are defined and associated outside the presence of the query provider, the only way to efficiently filter is by using inverse full text indexes. Google gives best effort queries service based on only two or three words supplied from a user's tagging folksonomy. The key words normally come from the mental tagging folksonomy inside the brain. However better `with respect to` filtering is transparently available, by using XQuery to augment mentally recalled folksonomies with personal folksonomy storage services such as del.icio.us.

The insight here is to note that mental folksonomy tags can serve as personal triggers, automatically and transparently enhanced by a record of our previous attention, instead of global triggers across the creations and attention of the whole web. The systems of interest here, Google and Del.icio.us, are largely data storage facilities with a query interface. The logic to tie the two systems together using user generated key words, would most likely be recursive in nature, as queries are sent to Google, then to Delicious, then to Google, etc. Given the opaque nature of the Google query interface, the integration described above would certainly require computational completeness that relational algebra systems cannot provide. The principles of XQuery coupled with a general purpose programming language create the perfect tool to solve such relevance query problems. Skuery successfully demonstrates the viability of such an idea.

6.3 Distributed Query Optimizers

The CDDB example in *Section 5.4* pulls the source data for its query from two disparate data sources: a web service and an XML database. Ownership and maintenance of these two sources are completely independent. In fact the two data source are completely unaware of each other. The query in the CDDB example is really a distributed query between three entities: the CDDB web service, the XQuery process, and the CD listening attention database. The entities in the simple CDDB example interact using trivial simple selection queries. As queries become more involved, additional details such as the data set schema, filtering criteria, and physical storage attributes enable dramatic optimizations that reduce query time and resources. The CDDB example query demonstrates the distributed tendency of semistructural data such as XML. Better understanding of distributed optimization for non-relational queries may be a rich area of future study and innovation.

6.4 Data Query Features for Dynamic Languages

One of the principle reasons for having adopted Scheme, as the general-purpose language for Skuery was the power provided through its simplicity. A minimal syntax and lists as the core data type, allows Scheme and other Lisp dialects to represent XML as a primitive data format. Scheme's powerful macro facility frees developers from the dogmas of Lisp's original designers and allows for transparent extension of the syntax and features of Scheme itself. These two features of Lisp dialects allow Skuery to become an integrated feature of the core language rather than a bolt on set of data query APIs.

The lessons learned during development and design of Skuery can easily be adopted by the next generation of dynamic languages, such as Perl6. Perl5 has already proven itself as an excellent tool for manipulation of plain text and web site generation. The next major version, Perl6, will add several features to enhance Perl's current ability to integrate with more structured relational and hierarchical data. Perl6 will include a feature called Rules that enables developers to dynamically modify the syntax of the core language to adopt the data processing model of Skuery. Much like Schemes Macros, Perl6 will allow parse time and compile time intervention by user code to expand syntactic extensions written using the Perl6 Rules feature. The Perl6 Rosetta project aims to integrate the fundamental data query and object relational principles of The Third Manifesto. The Third Manifesto advocates that returning to the fundamental theories of the relational model restores representational power to data models, enables model evolution, and more powerful machine based query optimization.

Perl6 will be build upon Parrot, a JIT-able byte code compiled virtual machine, specifically designed for dynamic languages such as Perl5, Perl6, Python, PHP, Ruby, Lisp, XQuery, etc. One of the promises of Parrot is the ability to share libraries across programming languages. Shared code is attractive, to the extent that core implementations of FLWOR, XPath, and other data query operations could be shared across languages. Skuery argues that data query operations should be first class syntactic features of a programming language and not just an appendage API library. By providing shared data engine primitives and leaving syntactic integration to individual language implementers, the Parrot engine could reduce the complexity and cost of natively integrating data query into programming languages.

6.5 Data and Business Process Orchestration Languages

Another area of interest is the utilization of XQuery as a data orchestration language for lower level use in enterprise data storage tiers. In this scenario an XQuery engine could become a core infrastructure piece providing amalgamated data in the fabric of a service-oriented architecture. XQuery will continue as the progenitor of successive generations and dialects of abstract integration between data query and computational operations.

7 Conclusion

Skuery demonstrates that integration of data query operations as a native feature in general purpose programming languages increases programmer productivity by reducing code size. Data query representation using Skuery in Scheme renders concisely and with a natural aesthetic. Skuery achieves its native feel because of its full integration with Scheme's syntactic form, S-expressions. The performance of Skuery's abstractions is within reach of commercial XQuery implementations. Building upon the standards of XML and XQuery and the theoretical insights of functional programming, Skuery returns the focus of the programmer from the details of micro data manipulation to influential orchestrations of data transformations. Upon these premises, Skuery stands as an example and sets a bar of expectations for future innovations integrating computation and data query.

A XQuery Abstract Syntax

```
(module abstract-syntax mzscheme
  (provide (all-defined))
  (require (file "datatype.ss"))

  (define-datatype ModuleAST
                   (ModuleMainASN Version Main)
                   (ModuleLibraryASN Version Library)
                   (ModuleMainNoVersionASN Main)
                   (ModuleLibraryNoVersionASN Library))
  (define-datatype MainAST (MainASN Prolog QueryBody))
  (define-datatype LibraryAST (LibraryASN ModuleDecl Prolog))
  (define-datatype VersionAST (VersionASN Version Encoding))
  (define-datatype ModuleDeclAST (ModuleDeclASN Name URI))
  (define-datatype PrologAST (PrologASN))
  (define-datatype QueryBodyAST (QueryBodyASN Expr))
  (define-datatype BaseURIDeclAST (BaseURIDeclASN URI))
  (define-datatype CollationDeclAST (CollationDeclASN Collation))
  (define-datatype DefaultNamespaceDeclAST (DefaultNamespaceDeclASN Namespace))
  (define-datatype FunctionDeclAST (FunctionDeclASN Name Params Type Body External))
  (define-syntax FunctionDeclASNCons
    (syntax-rules ()
      ((_ Name Params Type BodyOrExternal)
        (if (eq? BodyOrExternal 'EXTERNAL)
          (FunctionDeclASN Name Params Type () #t)
          (FunctionDeclASN Name Params Type BodyOrExternal #f)))))
  (define-datatype FunctionBodyAST (FunctionBodyASN Body))
  (define-datatype ParamListAST (ParamListASN))
  (define-datatype ParamAST (ParamASN Name Type))
  (define-datatype NamespaceDeclAST (NamespaceDeclASN Name URI))
  (define-datatype DeclaredVariableAST (DeclaredVariableASN Name Type Expr))
  (define-datatype ExprListAST (ExprListASN))
  (define-datatype FLOWR_AST (FLOWR_ASN))
  (define-datatype ForAST (ForASN ForVarsList))
  (define-datatype ForVarAST (ForVarASN Variable Type PositionVariable Expr ))
  (define-datatype LetAST (LetASN LetVarList))
  (define-datatype LetVarAST (LetVarASN Variable Type Expr))
  (define-datatype WhereAST (WhereASN Expr))
  (define-datatype OrderByAST (OrderByASN))
  (define-datatype OrderSpecAST(OrderSpecASN Expr Modifiers))
  (define-datatype OrderSpecModifiersAST
                   (OrderSpecModifiersASN Ascend/Descend Empty Greatest/Least Collation))
  (define-datatype ReturnAST (ReturnASN Expr))
  (define-datatype QuantifiedAST
                   (SomeASN QuantifiedList Expr)
                   (EveryASN QuantifiedList Expr))
  (define-datatype QuantifiedVarAST (QuantifiedVarASN Variable Type Expr ))
  (define-datatype TypeswitchAST (TypeswitchASN MatchExpr CaseList Variable ReturnExpr))
  (define-datatype CaseClauseAST (CaseClauseASN Variable SequenceType Expr))
  (define-datatype IfAST (IfASN Test TrueCase FalseCase))
  (define-datatype OrAST (OrASN Left Right))
  (define-datatype AndAST (AndASN Left Right))
```

```
(define-datatype InstanceOfAST (InstanceOfASN Left Right))
(define-datatype TreatAsAST (TreatAsASN Left Right))
(define-datatype CastableAsAST (CastableAsASN Left Right))
(define-datatype CastAsAST (CastAsASN Left Right))

(define-datatype <eq>AST (<eq>ASN))
(define-datatype <ne>AST (<ne>ASN))
(define-datatype <lt>AST (<lt>ASN))
(define-datatype <le>AST (<le>ASN))
(define-datatype <gt>AST (<gt>ASN))
(define-datatype <ge>AST (<ge>ASN))
(define-datatype <=>AST (<=>ASN))
(define-datatype <!=>AST (<!=>ASN))
(define-datatype <<>AST (<<>ASN))
(define-datatype <<=>AST (<<=>ASN))
(define-datatype <>>AST (<>>ASN))
(define-datatype <>=>AST (<>=>ASN))
(define-datatype <IS>AST (<IS>ASN))
(define-datatype <<<>AST (<<<>ASN))
(define-datatype <>>>AST (<>>>ASN))

(define-datatype RangeAST (RangeASN From To))
(define-datatype UnaryAST
                 (PositiveASN Value)
                 (NegativeASN Value))
(define-datatype UnionAST (UnionASN Left Right))
(define-datatype SetAST
                 (IntersectASN Left Right)
                 (ExceptASN Left Right))
(define-datatype PathAST (PathASN))
(define-datatype RelativePathAST (RelativePathASN))
(define-datatype StepAST (StepASN Axis NodeTest))
(define-datatype StepPredicateAST (StepPredicateASN Step PredicateList))
(define-datatype AbbrevStepAtAST (AbbrevStepAtASN Step))
(define-datatype FilterStepAST (FilterStepASN Expr PredicateList))
(define-datatype WildcardAST
                 (WildcardASN Wildcard)
                 (WildcardNameASN Wildcard Name))
(define-datatype PrimaryExprAST (PrimaryExprASN Expr))
(define-datatype PredicateAST (PredicateASN Expr))
(define-datatype ValidationAST (ValidationASN Expr SchemaMode SchemaContext))
(define-datatype SchemaContextAST (SchemaContextASN Context))
(define-datatype FunctionCallAST (FunctionCallASN Name Args))
(define-datatype XMLElementAST (XMLElementASN Name Attributes Content))
(define-datatype XMLCommentAST (XMLCommentASN Comment))
(define-datatype XMLProcessingInstructionAST (XMLProcessingInstructionASN Name Details))
(define-datatype XMLCDataAST (XMLCDataASN Data))
(define-datatype CompareDocCtorAST (CompareDocCtorASN Expr))
(define-datatype CompareElementCtorAST (CompareElementCtorASN Name Expr1 Expr))
(define-datatype CompareAttributeCtorAST (CompareAttributeCtorASN Name Expr1 Expr))
(define-datatype CompareTextCtorAST
                 (CompareTextEmptyCtorASN)
                 (CompareTextCtorASN Text))
```

```
(define-datatype CompareNamespaceCtorAST (CompareNamespaceCtorASN Name Expr))
(define-datatype CompareCommentCtorAST (CompareCommentCtorASN Expr))
(define-datatype ComparePICtorAST
                 (ComparePICtorEmptyASN)
                 (ComparePICtorASN Expr1 ExprA))
(define-datatype XMLElementContentAST (XMLElementContentASN))
(define-datatype XMLAttributeListAST (XMLAttributeListASN))
(define-datatype XMLAttributeAST (XMLAttributeASN Name Value))
(define-datatype EnclosedExprAST (EnclosedExprASN Expr))
(define-datatype AtomicTypeAST (AtomicTypeASN Type))
(define-datatype SequenceTypeAST
                 (SequenceTypeASN Type Occurance)
                 (EmptySequenceTypeASN))
(define-datatype ItemTypeAST (ItemTypeASN))
(define-datatype ElementTestAST
                 (ElementTestEmptyASN)
                 (ElementTestASN SchemaContextPath Name)
                 (ElementTestNodeNameASN NodeName TypeNAme))
(define-datatype AttributeTestAST
                 (AttributeTestEmptyASN)
                 (AttributeTestASN SchemaContextPath Name)
                 (AttributeTestNodeNameASN NodeName TypeNAme))
(define-datatype ProcessingInstructionsTestAST
                 (ProcessingInstructionsTestEmptyASN)
                 (ProcessingInstructionsTestASN Detail))
(define-datatype DocumentTestAST
                 (DocumentTestEmptyASN)
                 (DocumentTestASN Name))
(define-datatype CommentTestAST (CommentTestASN))
(define-datatype TextTestAST (TextTestASN))
(define-datatype NodeTestAST (NodeTestASN))
(define-datatype SchemaContextLocationAST (SchemaContextLocationASN Path Name))
(define-datatype SchemaImportAST (SchemaImportASN Prefix Name Location))
(define-datatype SchemaGlobTypeAST (SchemaGlobTypeASN Name))
(define-datatype SchemaPrefixAST
                 (SchemaPrefixASN Name)
                 (SchemaPrefixDefaultElementNamespaceASN))
(define-datatype VariableAST (VariableASN Name))
(define-datatype NOT_IMPLEMENTED_AST (NOT_IMPLEMENTED_ASN))

(define-datatype AddAST (AddASN arg1 arg2))
(define-datatype SubtractAST (SubtractASN arg1 arg2))
(define-datatype MultiplyAST (MultiplyASN arg1 arg2))
(define-datatype DivideAST (DivideASN arg1 arg2))
(define-datatype IntegerDivideAST (IntegerDivideASN arg1 arg2))
(define-datatype ModuloAST (ModuloASN arg1 arg2))

(define-datatype StringConcatAST (StringConcatASN))
(define-datatype INTEGER_AST (INTEGER_ASN integer))
(define-datatype DOUBLE_AST (DOUBLE_ASN double))
(define-datatype DECIMAL_AST (DECIMAL_ASN decimal))
(define-datatype EmptySequenceAST (EmptySequenceASN))
)
```

B Skuery CDDB Support Code

```
(module cddb mzscheme
 (require (lib "pregexp.ss"))
 (require "kprelude.ss")
 (require (planet "sxml.ss" ("lizorkin" "sxml.plt" 1 3)))

 (provide (all-defined))

 (define (connect x)
  (tcp-connect x 8880))

 (define (send port message)
  ;(printf message)
  (fprintf port message)
  (flush-output port))

 (define (until-end func x state)
  (let ((line (read-line x)))
   (if (not (string=? line ".\r"))
    (until-end func x (func line state))
    state))
 )

  (define (print-read-line x)
   (printf (read-line x)))

(define (print-until-end stream)
 (until-end
  (lambda (line state)
   (display line))
  stream
  ()))

  (define (process-result stream)
   (let ((result (reverse (until-end parse-it stream ()))))
    ;(display result)
    result))

(define (print-and-return x)
 (printf "~a~n" (if (pair? x) (cdr x) x))
 x)

(define (match-DTITLE x)
 (pregexp-match "^DTITLE=(.*)\\s+/\\s+(.*)\\s+$" x))
(define (match-TTITLE x)
 (pregexp-match "^TTITLE(\\d+)=(.*)\\s+$" x))

(define (parse-it line state)
 ;(display line)
 (let ((result (match-DTITLE line)))
  (if result
   (cons result state)
   (let ((result (match-TTITLE line)))
    (if result
     (cons result state)
     state)))))
(define (make-cd-xml TITLE ARTIST TRACKS)
 `(cd
   (@ (title ,TITLE ) (artist ,ARTIST))
   ,@TRACKS
  ))

  (define (make-track-xml TRACK NAME)
   `(track (@ (track ,TRACK) (name ,NAME))))

  (define (cd-list->xml x)
   (let ((TITLE (caddar x))
         (ARTIST (cadar x))
         (TRACKS (track-list->xml (cdr x))))
    (make-cd-xml TITLE ARTIST TRACKS)))

  (define (track-list->xml x)
   (map (lambda (x)
         (make-track-xml (+ (string->number (cadr x)) 1) (caddr x)))
    x))

(define (query arg)
 (let-values (((in out)(connect "us.freedb.org")))
  (read-line in)
  (send out "cddb hello tewk tewk.com schemeCDDB 0.0.1~n") (read-line in)
  (send out (format "cddb query ~a~n" arg))
  (letrec ((match (pregexp-match "\\d+\\s+(\\S+)\\s+(\\S+)" (read-line in)))
           (genre (cadr match))
           (id (caddr match)))
   (send out (format "cddb read ~a ~a~n" genre id)))
  (let ((result (cd-list->xml (process-result in))))
   (send out "quit~n") (read-line in)
   result
  )))

(define (gen-search-string cd)
 (let ((ID (car ((sxpath "/@id/text()") cd)))
       (TRACKS (car ((sxpath "/@tracks/text()") cd)))
       (SECONDS (car ((sxpath "/@seconds/text()") cd)))
       (OFFSETS (map (lambda (track) (car ((sxpath "/@offset/text()") track)))
                     ((sxpath "/track") cd))))
  (format "~a ~a ~a ~a" ID TRACKS (print-list OFFSETS) SECONDS)))

  (define (query-cd-data arg)
   (query (gen-search-string arg)))
)
```

C jCDDB Implementation

```
package jcddb;

import java.net.Socket;
import java.io.*;
import java.util.regex.*;
import java.util.Properties;
import org.w3c.dom.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import javax.xml.parsers.*;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import net.sf.saxon.Configuration;
import net.sf.saxon.query.DynamicQueryContext;
import net.sf.saxon.query.StaticQueryContext;
import net.sf.saxon.query.XQueryExpression;
import net.sf.saxon.om.*;

public class Main
{
  Socket socket;
  BufferedReader in;
  BufferedWriter out;
  Pattern dTitle;
  Pattern tTitle;
  Pattern trackInfo;
  Document doc;

  public Main()
  {
    try
    {
      socket = new Socket("us.freedb.org", 8880, true);
      in = new BufferedReader( new InputStreamReader( socket.getInputStream() ));
      out = new BufferedWriter( new OutputStreamWriter( socket.getOutputStream() ));
      dTitle = Pattern.compile("DTITLE=(.*)\\s+/\\s+(.*)$");
      tTitle = Pattern.compile("TTITLE(\\d+)=(.*)$");
      trackInfo = Pattern.compile("\\d+\\s+(\\S+)\\s+(\\S+)");
    }
    catch (IOException ioe)
    {
    }
  }

  public static void main(String[] args) {
    Main main = new Main();
    main.example();
  }
```

```
public static Element cddb_query_cd_data(Element x)
{
  String trackoffsets = "";
  NodeList ns = x.getChildNodes();
  for (int e = 0; e < ns.getLength(); e++)
  {
    Node n = ns.item(e);
    if (n.getNodeType() == Node.ELEMENT_NODE)
    {
      trackoffsets += ((Element) ns.item(e)).getAttribute("offset") + " ";
    }
  }

  String keystring = String.format("%s %s %s %s",
    x.getAttribute("id"),
    x.getAttribute("tracks"),
    trackoffsets,
    x.getAttribute("seconds"));
  Main main = new Main();
  return main.query(keystring).getDocumentElement();
}

public void example()
{
  Configuration config = new Configuration();
  StaticQueryContext staticContext = new StaticQueryContext(config);
  try
  {
    XQueryExpression exp = staticContext.compileQuery(
        "declare namespace jcddb=\"java:jcddb.Main\";\n" +
        "for $x in doc(\"cds.xml\")/collection/cd \n" +
        "let $cdtextdata := jcddb:cddb_query_cd_data($x) \n" +
        "let $title := $cdtextdata/@title \n" +
        "let $artist := $cdtextdata/@artist \n" +
        "let $tracks := $cdtextdata/track \n" +
        "for $y in $tracks \n" +
        "let $tracktitle := $y/@title \n" +
        "let $tracknum := $y/@tracknumber\n" +
        "let $numlistens := $x/track[$tracknum + 1]/@numberoflistens \n" +
        "where $numlistens > 30 \n" +
        "return ( string($title), string($artist), string($tracknum + 1), " +
        "string($tracktitle), string($numlistens), \"\n\") \n"
        );
    DynamicQueryContext dynamicContext =
      new DynamicQueryContext(config);
    Properties props = new Properties();
    props.setProperty(OutputKeys.METHOD, "text");
    props.setProperty(OutputKeys.INDENT, "yes");
    exp.run(dynamicContext, new StreamResult(System.out), props);
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}

public Document query(String arg)
{
  readline();
  write("cddb hello tewk tewk.com kjavaCDDB 0.0.1\n");
  readline();
  write(String.format( "cddb query %s\n", arg));
  Matcher matcher = trackInfo.matcher(readline());
  matcher.lookingAt();
  write(String.format( "cddb read %s %s \n", matcher.group(1), matcher.group(2)));
  Document xmlDoc = processResult();
  System.out.println();
  write("quit\n");
  readline();

  // Serialisation through Tranform.
  try
  {
    DOMSource domSource = new DOMSource(xmlDoc);
    StreamResult streamResult = new StreamResult(System.out);
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer serializer = tf.newTransformer();
    serializer.setOutputProperty(OutputKeys.ENCODING,"ISO-8859-1");
    serializer.setOutputProperty(OutputKeys.INDENT,"yes");
    //serializer.transform(domSource, streamResult);
  }
  catch (TransformerException te)
  {
  }

  return xmlDoc;
}

void write(String string)
{
  try
  {
    //System.out.println(string);
    out.write(string);
    out.flush();
  }
  catch (IOException ioe)
  {
  }
}

String readline()
{
  try
  {
    String indata = in.readLine();
    //System.out.println(indata);
    return indata;
  }
  catch (IOException ioe)
  {
    return "";
  }
}


Document processResult()
{
  try
  {
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance ();
    DocumentBuilder db = dbf.newDocumentBuilder ();
    doc = db.newDocument ();
    Element root = doc.getDocumentElement();

    String data = readline();
    data = readline();
    while (!data.equals("."))
    {
      Element result = parseIt(data);
      if (result != null)
      {
        if (result.getNodeName() == "cd")
        {
          doc.appendChild(result);
          root = doc.getDocumentElement();
        }
        else
        {
          root.appendChild(result);
        }
      }
      data = readline();
    }
  }
  catch (ParserConfigurationException pce)
  {
  }
  return doc;

}

Element parseIt(String line)
{
  Matcher dM = dTitle.matcher(line);
  if ( dM.lookingAt() )
  {
    Element el = doc.createElement ("cd");
```

```
      el.setAttribute("artist", dM.group(1));
      el.setAttribute("title", dM.group(2));
      return el;
    }
    else
    {
      dM = tTitle.matcher(line);
      if ( dM.lookingAt() )
      {
        Element el = doc.createElement ("track");
        el.setAttribute("tracknumber", dM.group(1));
        el.setAttribute("title", dM.group(2));
        return el;
      }
      else
      {
        return null;
      }
    }
  }
}
```

References

[1] Jans Aasman and John Foderaro. Allegrocache object persistence in lisp. In *International Lisp Conference 2005*, 2005.

[2] Serge Abiteboul. Querying semi-structured data. In *ICDT '97: Proceedings of the 6th International Conference on Database Theory*, pages 1–18, London, UK, 1997. Springer-Verlag.

[3] Serge Abiteboul. Semistructured data: From practice to theory. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 379, Washington, DC, USA, 2001. IEEE Computer Society.

[4] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernndeand Michael Kay, and Jonathan Robi eand Jérôme Siméo. *XML Path Language (XPath) Version 2.0.* http://www.w3.org/TR/xpath20/, 2005.

[5] Andrew Blumberg, Ben Lee, and Robert L. Read. *Elephant an object database for Common Lisp.* http://common-lisp.net/project/elephant/, 2006.

[6] Scott Boag, Don Chamberlin, Mary F. Fernndez, Daniela Florescu, and Jonathan Robie an Jérôme Siméo. *XQuery 1.0: An XML Query Language.* http://www.w3.org/TR/xquery/, September 2005.

[7] Raymond F. Boyce and Donald D. Chamberlin. Using a structured english query language as a data definition facility. *IBM Research Report*, RJ1318, 1973.

[8] Michael Brundage. *The XML Query Language.* Addison, Wesley, 2004.

[9] T. Bhm and E. Rah. Xmach-1: A benchmark for xml data management. In *In Proceedings of German database conference BTW2001*, March 2001.

[10] Don Chamberlin, Denise Draper, Mary F. Fernández, Michael Kay, Jonathan Robie, Michael Rys, Jérôme Siméon, Jim Tivy, and Philip Wadler. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison Wesley, Reading, Massachusetts, August 2003.

[11] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0*. http://www.w3.org/TR/xpath, 1999.

[12] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[13] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with stored. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 431–442. ACM Press, 1999.

[14] Mary Fernandez, Jérôme Siméon, and Philip Wadler. An algebra for XML query. *Lecture Notes in Computer Science*, page 11, 2000.

[15] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages Second Edition*. MIT Press, Cambridge, MA, 2001.

[16] R. Goldman, J. Mchugh, and J. Widom. From semistructured data to XML: Migrating the lore data model and query language. In *Workshop on the Web and Databases (WebDB '99)*, pages 25–30, 1999.

[17] Peter M. D. Gray. *The Functional Approach to Data Management: Modeling, Analyzing, and Integrating Heterogeneous Data*. Springer, November 2003.

[18] Torsten Grust. Monad comprehensions: A versatile representation for queries.

[19] Torsten Grust and Jens Teubner. Relational algebra: Mother tongue—xquery: Fluent. In *Proceedings of the first Twente Data Management Workshop on*, 2004.

[20] Paul Hudak and Phillip Wadler. Report on the functional programming language haskell. Technical Report YALEU/DCS/RR-666, Yale University, Department of Computer Science, December 1988.

[21] S.L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Series)*. Prentice Hall, May 1987.

[22] Oleg Kiselyov. Sxml specification 3.0, March 2004.

[23] Oleg Kiselyov and Kirill Lisovsky. Xml, XPath, XSLT implementations as SXML, SXPath, and SXSLT.

[24] Ashok Malhotra, Jim Melton, and Norman Walsh. *XQuery 1.0 and XPath 2.0 Functions and Operators*. http://www.w3.org/TR/xpath-functions/, 2005.

[25] John L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[26] Paul McJones. *SystemR*. http://www.mcjones.org/System_R/.

[27] Erik Meijer. Confessions of a used programming language salesman (getting the masses hooked on haskell). Submitted to ICFP 2006, 2006.

[28] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[29] Oleg Paraschenko. *An approach to portable implemenation of the XQuery language.* http://xmlhack.ru/protva/xquery-eng.pdf, 2004.

[30] Oleg Paraschenko. *Reusing XML Processing Code in non-XML Applications.* http://uucode.com/texts/genxml/genxml.html, 2005.

[31] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, March 2002.

[32] Mark Pilgrim. *Dive Into Python.* APress, 2002.

[33] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. *VLDB*, pages 974–985, 2002.

[34] Peter Seibel. *Practical Common Lisp.* Apress, September 2004.

[35] Guy L. Steele, Jr. and Richard P. Gabriel. The evolution of Lisp. *ACM SIGPLAN Notices*, 28(3):231–270, 1993.

[36] Dan Suciu. An overview of semistructured data. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 29(4):28–38, 1998.

[37] Dan Suciu and Limsoon Wong. On two forms of structural recursion. In Georg Gottlob and Moshe Y. Vardi, editors, *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*,

volume 893 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 1995.

[38] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 306–313, New York, NY, USA, 1995. ACM Press.

[39] Phil W. Trinder. Comprehensions – a query notation for dbpls. In *1990 Glasgow Database Workshop*, pages 95–102, March 1990.

[40] D. Weinreb, N. Feinberg, D. Gerson, and C. Lamb. *An object-oriented database system to support an integrated programming environment.* Prentice Hall, 1991.

[41] Wikipedia. *Wikipedia: SQL.* http://en.wikipedia.org/wiki/SQL, May 2006.