OPTIMIZING SPARQLER USING SHORT CIRCUIT EVALUATION OF FILTER CLAUSES

by

EDWIN RALPH PURVEE

(Under the Direction of Krzysztof J. Kochut)

ABSTRACT

The discovery of semantic relationships between two entities is an area of interest and study in the Semantic Web. Regarding the data in an RDF store as a graph allows the utilization of path searches, but with large ontologies exhaustive searches are not practical. SPARQLerR, an extension of SPARQL, sought to utilize regular expressions in the search and construction of a path which helped avoid exhaustive searches by limiting the search space during evaluation. Still, due to the filter constraint evaluation occurring after a path was constructed, the actual search space of the beginning and ending resources was not being limited. To address this problem, an optimization is proposed that uses a form of short circuit evaluation of the filter constraints in a SPARQL query to limit the search space as early as possible to avoid costly path searches between two nodes.

INDEX WORDS: SPARQLerR, Semantic Web, BRAHMS, Query Optimization
OPTIMIZING SPARQLER USING SHORT CIRCUIT EVALUATION OF FILTER CLAUSES

by

EDWIN RALPH PURVEE

BS, Southern Nazarene University, 1999

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2009
OPTIMIZING SPARQLER USING SHORT CIRCUIT EVALUATION OF FILTER CLAUSES

by

EDWIN RALPH PURVEE

Major Professor: Krzysztof J. Kochut
Committee: John A. Miller
Hamid R. Arabnia

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2009
ACKNOWLEDGEMENTS

I would like to acknowledge all the people who made this possible. First, I must acknowledge my wife Beth, who has supported me through the entire process of pursuing my graduate degree, my mother and father, through their upbringing; they have guided me to where I am today. I acknowledge my major professor, Dr. Kochut, whose patience and guidance were crucial to me successfully finishing my degree. Finally I recognize the many people, too numerous to fully state, who have influenced or aided me in my life to this point.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 The Semantic Web</td>
<td>3</td>
</tr>
<tr>
<td>2.2 RDF</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Ontologies</td>
<td>6</td>
</tr>
<tr>
<td>2.4 RDF Query Languages</td>
<td>9</td>
</tr>
<tr>
<td>2.5 SPARQL</td>
<td>13</td>
</tr>
<tr>
<td>2.6 RDF Storage Systems</td>
<td>14</td>
</tr>
<tr>
<td>2.7 Short Circuit Evaluation</td>
<td>16</td>
</tr>
<tr>
<td>3 Related Work</td>
<td>18</td>
</tr>
<tr>
<td>4 SPARQLeR Extension and Implementation</td>
<td>22</td>
</tr>
<tr>
<td>4.1 SPARQLeR</td>
<td>22</td>
</tr>
<tr>
<td>4.2 BRAHMS Architecture</td>
<td>24</td>
</tr>
<tr>
<td>4.3 SPARQLeR Architecture</td>
<td>26</td>
</tr>
<tr>
<td>4.4 SPARQLeR Modification</td>
<td>29</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table 1: DBLP information</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>39</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Not Decision Tree</td>
<td>33</td>
</tr>
<tr>
<td>2</td>
<td>And Decision Tree</td>
<td>34</td>
</tr>
<tr>
<td>3</td>
<td>Or Decision Tree</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>Execution Time for Non-Path Queries</td>
<td>41</td>
</tr>
<tr>
<td>5</td>
<td>Speed Up Ratio of Non-Path Queries</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>Number of Matches for Non-Path Queries</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>Execution Time for Path Queries</td>
<td>44</td>
</tr>
<tr>
<td>8</td>
<td>Speed Up Ratio of Path Queries</td>
<td>44</td>
</tr>
<tr>
<td>9</td>
<td>Exe. Time and Path Match for Single-Source Queries</td>
<td>46</td>
</tr>
<tr>
<td>10</td>
<td>Exe. Time of Single Source Non-Optimized Query and Speed Up Ratio</td>
<td>47</td>
</tr>
<tr>
<td>11</td>
<td>Exe. Time and Path Match for Def. Start and End Node Queries</td>
<td>49</td>
</tr>
<tr>
<td>12</td>
<td>Exe. Time of Def. Start and End Nodes Non-Optimized Query and Speed Up Ratio</td>
<td>50</td>
</tr>
</tbody>
</table>
CHAPTER 1  
INTRODUCTION

Discovery of semantic associations in large ontologies is an important area of study due to the numerous applications in which it can be used. From discovering links between suspects in law enforcement, chemical reactions in biological processes, or semantic based context searches for research, many of the potential benefits make the pursuit worthwhile. There are many obstacles that still must be overcome. Dealing with any ontology that is representative of an area of interest, one is more than likely dealing with an enormous data set.

When represented as a graph, there is so much data that without extensive filtering, one cannot effectively visualize the data. Making effective associations is a challenge. Doing it quickly and with minimal resources provides additional challenges. Typically, when looking for paths between two nodes, traditional approaches require knowing about the edges between them or conducting an exhaustive search, which as the path length increases, the total number of paths that can be searched increases exponentially or greater.

In the SPARQLeR [1] extension to SPARQL, regular expression searches were utilized to find paths between two resources in an Resource Description Framework (RDF) store more efficiently. This allowed for faster searches as nodes and edges could be eliminated during the search, but it still relied on the Filter constraints on the start and end node being checked after a potential path was constructed, which could result in the path being constructed, but being rejected due to one or both of the nodes failing the constraint conditions. It makes sense to avoid a path search, if possible, which can have a costly execution time.

The solution proposed to optimize and avoid unnecessary path searches is to evaluate the constraints in the filter expression of a SPARQL query in a short circuit fashion [2], by utilizing
the constraints defined in the filter clause throughout the query plan to eliminate potential nodes and to limit the search space for path searches if any of the known values would exclude these nodes. Even if all variable values were not known prior to the search, the filter conditions could be used to exclude it.

Chapter 2 covers background information needed as reference for this work. Chapter 3 presents related work to the optimization presented. Chapter 4 reviews the architecture of the implementation and presents our changes and implementation of the optimization. Chapter 5 presents the evaluation of the optimization and the results. Chapter 6 discusses the conclusions and offers considerations for future work.
CHAPTER 2

BACKGROUND

2.1 The Semantic Web

The Semantic Web [3] is envisioned as the next iteration of the World Wide Web. Before and currently, what we know as the web [4] was only a means of providing human understandable information through the form of web pages created in HTML. In this system, humans formatted text for humans to read, with the text being displayed by a web browser.

When doing searches for web pages, the most common way to do a search was to seek all web pages that contained key words. While some success was achieved in finding information, these searches had very little context in providing results. While more advanced search algorithms and results ranking were done by sites, e.g., www.google.com, web searches still are done by key words and not by context searches.

The Semantic Web seeks to make information not only human understandable but also have the data presented machine understandable, or that computers are able make use of the data without complex interpretation. The method to bring this about was through declarative languages which sought to define classes of information and relationships between that data. The Extensible Markup Language (XML) [5] was used as the primary means of serialization for the RDF [6] language.

While XML did provide a means of defining data, it did not provide a means of defining relationships among data, and when XML data was provided to another application, that application had to have knowledge of the data being provided to make use of it [7]. RDF provided the frame work to not only provide data but provide information and relationships
about the data, which would allow an application to make use of the data even without the application having knowledge of the data previously, as long as it can use RDF.

There are other languages that are available to help realize the concept of the Semantic Web. One of which is OWL [8], an update of DAML+OIL [9], although it cannot be reduced to RDF. DAML+OIL is a collection of triples which are used to define the object domain. OWL can be used by applications to process documents, finding the meaning of terms so that the data can then be machine understandable, thus utilizing the concept behind the Semantic Web.

The promise of this system is that information can be used for automation, reuse, and sharing of data across different platforms and networks [10]. More advanced searches could be accomplished where the query would not just search for matches of a key word, but results could be returned for information relating to the topic searched on. Intelligent use of data can be utilized for advanced applications that do not have to rely on artificial intelligence since the data is in form which can be utilized by the machine without advanced processing.

2.2 RDF

RDF’s main function is to supply metadata. RDF supplies metadata about data using Uniform Resource Identifiers (URIs). URIs describes resources as simple properties or property values. URIs appear to be Uniform Resource Locators (URLs), using the same syntax but are only there for descriptions.

The URIs can be used to define individuals, classifications, properties, and values of properties. An example of this could be

- individual: Edwin Purvee http://www.cs.uga.edu/~purvee#person
- classification: Runner http://www.cs.uga.edu/~purvee#runner
- properties: 1500m time http://www.cs.uga.edu/~purvee#1500mtime
• value of the property, eg “4:08.3”.

The actual RDF representation of this would be

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#">

    <contact:Person rdf:about="http://www.cs.uga.edu/~purvee#me">
        <contact:fullName>Edwin Purvee</contact:fullName>
        <contact:1500mtime rdf:resource="4:08.3"/>
    </contact:Person>

</rdf:RDF>
```

RDF also makes statements about resources. For example,

[http://www.cs.uga.edu/~purvee/index.html](http://www.cs.uga.edu/~purvee/index.html) has a creator whose value is Edwin Purvee. The statement is made up of the resource, or subject, the statement describes the property, or predicate, and the value, or object, of the property. RDF represents these statements as triples so our example above would become

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:exterms="http://www.example.org/terms/">

    <rdf:Description rdf:about="http://www.cs.uga.edu/~purvee/index.html">
        <exterms:creator>http://www.cs.uga.edu/~purvee#me</exterms:creator>
    </rdf:Description>

</rdf:RDF>
```
Using graph models, the subjects and objects become nodes while the predicates are arcs. Each triple represents one arc from the starting node to the ending node.

2.3 Ontologies

An ontology is a specification of a conceptualization [11]. Ontologies as used in RDF and OWL contain classifications the hierarchy of classes, and relationships between data items [12]. Ontologies are made up of classification schemes and vocabularies. Classification schemes organize objects by subject categories. Vocabularies define how the data can be used. The following is a brief example of an ontology.

class-def track-athlete
    slot-constraint personal-bests
        value-type performance

class-def track-events

class-def sprints
    subclass-of track-events

class-def distance
    subclass-of track-events

class-def sprinter
    subclass-of track-athlete
        slot-constraint competes
            value-type sprints

class-def distance-runner
    subclass-of track-athlete
        slot-constraint competes
value-type distance

This ontology describes track and field athletes that can be sprinters or distance runners. The sprinters compete in track events which are sprints and the distance runners compete in track events that are distance. Subclass-of references inheritance as a “is a type of” relationship. A slot-constraint is a behavior of the class. The XML representation of this ontology is as follows.

```xml
<class-def>
    <class name="track-athlete"/>
    <slot-constraint>
        <slot name="personal-bests"/>
        <has-value>
            <class name="performance"/>
        </has-value>
    </slot-constraint>
</class-def>
<class-def>
    <class name="track-events"/>
</class-def>
<class-def>
    <class name="sprints"/>
    <subclass-of>
        <class name="track-events"/>
    </subclass-of>
</class-def>
```

<class name="distance"/>

<subclass-of>

  <class name="track-events"/>

</subclass-of>

</class-def>

<class-def>

  <class name="sprinter"/>

  <subclass-of>

    <class name="track-athlete"/>

  </subclass-of>

  <slot-constraint>

    <class name="competes"/>

    <has-value>

      <class name="sprints"/>

    </has-value>

  </slot-constraint>

</class-def>

<class-def>

  <class name="distance-runner"/>

  <subclass-of>

    <class name="track-athlete"/>

  </subclass-of>

  <slot-constraint>
2.4 RDF Query languages

In order to be able to take advantage of the semantic capabilities of data expressed in ontologies, there has to be a mechanism to access and search the data that is available. To do this, many query languages have been developed [13]. Using these languages, searches can be done for triples, where more than just a key word can be used to find matching documents. These searches can be done by subject or any one of a number of properties or objects that exist in the ontology used. Also, cross-referencing can be done more accurately. Data can be found in context pertaining to what it is about, rather than relying on a higher percentage of key hits.

The following are a number of RDF query languages. Algae2 [14] is a constraint based query language based on Algernon, using an N3 syntax for queries and assertions. Queries collect a set of results in a string after being passed a question string. The general format for a query is

(ask '(*constraint string*):collect '?results')

The collect defines what the query will return. The constraint string defines what is being searched for, like documents, and the limitations for which they must meet. For example “all documents which were annotated by Joe.”
N3 is a declarative logic language used for querying RDF. The following is an example of a query in N3:

```n3
this log:forAll :service, :port, :binding, :bindingName.
{
 :service rdf:type wsdl:service.
 :service wsdl:hasPort :port.
 :binding wsdl:name :bindingName).
} log:implies {
 :service a agg:Q1Answer.
 :bindingName a agg:Q1Answer.
}

The notation of using `:` before a word makes that word a class or a classification, so service would be an RDF type which would have a WSDL service. These assertions are made which have to meet constraints that are given in the second block and contained in the brackets {}.

Squish [15] is a query language that borrowed much of its syntax from SQL. Here is an example of a query in Squish.

```squish
SELECT
 ?port, ?binding
FROM
 http://www.w3.org/2001/03/19-annotated-RDF-WSDL.rdf
WHERE
```
Variables start with a "?", arguments that are in FROM are URLs which are de-serialized for the basis of the query. The predicate strings, rdf::type ?service wsdl::service need a clause in USING to specify what the predicate stands for. Values will be integers or strings. This language can be used to query for data much as one would an SQL database. Except, it has the ability to search for data from ontologies, so it can search for track athletes who are distance runners out of a RDF knowledge base of people that has the appropriate data.

RQL [16] uses functional based programming, based on OQL [17]. RQL combines schema and data querying using taxonomies and classification of resources. It was extended with an RQL interpreter to be used on top of an Object-Relational Database Management System (ORDBMS). Here is an example of an RQL query.

```sql
select P1, B1
from
#service.#hasPort{P1}.#binding{B1}.#style {SD},
{B1}#name
where SD = "document"
```
RQL has much of the same capabilities as Squish and is SQL like as well. It is used in much the same fashion.

DQL [18] was designed to do simple queries on DAML+OIL knowledge bases (KB). The language is a DAML+OIL ontology so that queries and results are in DAML+OIL syntax. A query consists of a query premise and a query pattern. A query premise is a DAML+OIL KB that is asserted to the queried KB. It contains assumptions particular to the query. The query pattern is the question that is a conjunction of one or more triples where each triple corresponds to a RDF statement. The answer to the query is in the form of a KB.

RDQL [19] was based on Squish and seeks to achieve more flexibility in queries for knowledge representation. If the RDF model is a graph expressed as a set of triples, a query in RDQL is expressed as a set of triple patterns. Each triple pattern is made up of variables, URIs, and literals. Here is an example of an RDQL query.

```
SELECT ?port, ?binding

FROM <http://www.w3.org/2001/03/19-annotated-RDF-WSDL.rdf>

WHERE

 (?service   rdf:type      wsdl:service)
 (?service   wsdl:hasPort  ?port)
 (?port      wsdl:binding  ?binding)
 (?binding   wsdl:name     ?bindingName)
 (?binding   wsdl:hasBinding  ?x)
 (?x         wssoap:style  wssoap:document)

USING

  wsdl FOR <http://schemas.xmlsoap.org/wsdl/> ,
  wssoap FOR <http://schemas.xmlsoap.org/wsdl/soap/>
```
SeRQL [20] took features of RQL, RDQL, and N3 and extended with features of its own, e.g., graph transformation, RDF Schema support, XML Schema data type support, expressive path expression syntax, and optional path matching. This query language would be used when a RQL KB is available and one would like to do queries based on RQL triples. Here is an example of a SeRQL query.

```
SELECT port, binding
FROM
{}
  <rdf:type> <wsdl:service>;
  <wsdl:hasPort> {port}
{port} <wsdl:binding> {binding} <wsdl:name>
{bindingName;
  <wsdl:hasBinding> {}
}<wssoap:style> <wssoap:document>

USING NAMESPACE
  rdf = <!http://www.w3.org/1999/02/22-rdf-syntax-ns#> ,
  wsdl = <!http://schemas.xmlsoap.org/wsdl/> ,
  wssoap = <!http://schemas.xmlsoap.org/wsdl/soap/> 
```

2.5 SPARQL

SPARQL [21] is a W3C Standard for querying RDF. It can be used to retrieve results from triple patterns and optional constraints. Right now SPARQL is the most widely used and implemented semantic query language.

SPARQL can get specific data from a store of triples from variables that are defined in the Select section of the query, normally beginning with a ‘?’ . The Prefix is used to identify the ontology or dataset that will be utilized, and can be compared to the From section in a SQL
query. Inside the Where section, triple patterns are given that when a triple matches the pattern, the results of the items corresponding to the variables are returned. Inside the Where, a filter section is available that can put constraints on variables which are used to further refine the result set through comparison and Boolean operations.

The following is an example of a query from the examples included with SPARQLer:

```sparql
SELECT ?prop ?object
WHERE {
  <http://lsdis.cs.uga.edu/projects/glycomics/2006/Glyc0#high-mannose_N-glycanDerived_oligosaccharide>
    ?prop
    ?object .
}
```

In this example, the two variables whose values will be returned if matches are found are ?prop and ?object. The Subject of the triple in the triple pattern defined in the Where section is the literal <http://lsdis.cs.uga.edu/projects/glycomics/2006/Glyc0#high-mannose_N-glycanDerived_oligosaccharide> so all triples with the subject with that representation will match that pattern and will have their values of their object and predicate used to populate ?prop and ?object.

### 2.6 RDF Storage Systems

In order to query and make use of semantic data that is represented in RDF, there must be storage systems from which queries to the semantic data can be made. Currently, the most popular are Jena, Redland, and Sesame. BRAHMS will also be evaluated. These three other storage systems have built in or add on support, for the SPARQL query language.
Jena [22] is an RDF API written in Java. Its original goal was to implement RDF directly to the standard by the W3C. It is able to store RDF graphs in memory or persistently using a database or file. Jena is extremely modular, being separated into the following parts: XML parser, XML writer, Common classes and model, query engine, and the data access classes. Different parsers, data stores and query engines can be added in as long as they conform with the main Jena API. This allows a high degree of extendibility.

Redland [23] is an RDF storage system written in C. It has primary goals of being able to interface with scripting languages used with the web, mandating that it be written in a language easily callable by many different languages. Its architecture uses a more simple representation of the graph model of RDF triples and utilizes hashes for accessing the individual elements. Like Jena, Redland is able to store RDF graphs in main memory and persistently in a database or file in a modular fashion. Multiple semantic query languages, including SPARQL, are available for use with Redland.

Sesame [24], like Jena, is written in Java. It has its own API for accessing and using the RDF data, but is more web-centric, utilizing HTTP and SOAP protocol handlers to handle access in a standard way. It supports a persistent data module with a layer, called Storage And Inference Layer (SAIL), that handles the specifics of interfacing with a database or file. It also has an extensible query module allowing multiple query languages to be used against its graphs.

From a pure functionality standpoint, Jena, Redland, and Sesame, are all very similar in the base functionality that they provide. They all offer a means of representing an RDF graph, they all support persistent storage with multiple databases and file options, and they support standard semantic query languages with the ability to add on more.
BRAHMS [25] was developed to overcome the problem of conducting searches on large graph representations of ontologies. The established RDF stores were inadequate when trying to discover long semantic associations when the dataset was very large. While the query languages, of which SPARQL is included, could find simple paths, they were not able to provide the semantic discovery. BRAHMS was designed to be an extremely fast store for RDF to use main memory. The limitation to achieve the goal meant that it would be read-only. Its speed is achieved by hash tables which match URI’s by strings. Similarly to Redland, fast indexes are used for:

Subject -> (Object, Predicate)

(Subject, Object) -> Predicate

(Subject, Predicate) -> Object

Object -> (Subject, Predicate)

(Object, Predicate) -> Subject

Predicate -> (Subject, Object)

When tested for the time taken to make Semantic associations, BRAHMS outperformed Jena, Redland, and Sesame when tested with large ontologies in [25].

2.7 Short Circuit Evaluation

The optimization presented was inspired by Short Circuit Evaluation [2], which comes from compiler design for optimization based on evaluation of Boolean expressions. The check for true or false can be known from one value, and thus “short circuit” the rest of the comparisons. For example, if there is a check using the “or” operator:

if(x < 10 || y==3)
The compiler sets a jump to the code that is to be executed if the condition is met. If the first condition is true, the entire statement is true and no more evaluation is necessary. In this case, if it is found that \( x < 10 \), the execution would be set to jump to the code to execute inside the if statement. If it was false, the evaluation of \( y==3 \) would become necessary. A similar condition to this exists with the “and” operator, except it deals with the failure condition. For example:

\[
\text{if}(x<10 \&\& y==3)
\]

In this example the compiler sets a jump to the area of code outside the if statement, if the first condition is not met. If \( x \) were greater than or equal to 10, execution would jump around the if statement’s block of code or go to the else condition, as evaluating \( y \) is unnecessary due to the condition requiring true for both evaluations and would always be false at that point. Control flow can be directed in this manner efficiently without having to check all data when enough of the condition is met to determine the result of the entire expression.
CHAPTER 3

RELATED WORK

Optimizing queries in relational databases has been an area with extensive study done on it. With many RDF stores running on top of these systems, there can be techniques and algorithms that can be applied to semantic queries for improved performance. It was suggested in [26] that a schema for SQL be used to process RDF stores in a database, due to pure SQL queries being much faster than any of the other RDF stores available at that time.

SPARQL is implemented on all the major RDF storage systems evaluated here. ARQ [27] is the implementation using Jena. It serves as the query engine layer and was made to support SPARQL. On Redland, SPARQL was implemented in the Rasqal [28] library, which also implements RDQL. While ARQ is tied to Jena, Rasqal is intended to be portable, working on many systems. Sesame 2.0 included full support for the SPARQL query language, while the previous version only did through 3rd party add-ons. There has been work with optimization of SPARQL queries.

In [29] the authors utilize the Chase and Backchase [30] (C&B) algorithm when applied to SPARQL Algebra to reformulate queries to achieve faster execution. The SPARQL Algebra utilizes relational algebra [31] optimization techniques which then are utilized by the C&B algorithm. The C&B algorithm uses the chase technique [32] defined as if there is a set of instances P defined by a set of dependencies, for example C (P = SATISFIES(C)) and there a tableau CHASE\(_C\)\(T\) that is used to determine whether T is the identity mapping on SAT(C). In [30], the authors state that the chase can be used to decide equivalence of conjunctive queries. C&B has two phases. During the chase phase, the algorithm chases a sub-query if it is able to be chased, and during the backchase phase, it backtracks and attempts another chase. Using this,
the authors prove that C&B produces a containment mapping for the universal query plan when complete. The optimization demonstrated in [29], is the identification of semantically equivalent sub-queries which are less complex and faster to execute. A similar method is presented in [33] where the authors un-nested nested loop SQL queries. They converted the query to an algebraic framework and then converted the query to an equivalent semi-join query which executes more efficiently than a nested query in the constraints would, as the new query evaluates a smaller set of data and unnecessary searches are avoided.

In [34] the authors introduce pipelining for optimization of multiple SQL queries. Somewhat similar to the hardware paradigm of pipelining instructions, multiple queries were found to oftentimes use equivalent sub-expressions. Re-expressing queries to utilize the similarities was a successful approach but by solving and maintaining the results of the expressions, they could be used by the multiple queries but solved only once, saving processing time.

In [35], the authors discuss the instance of a user of Jena posting a question on why his simple query ran extremely slow over a relatively small RDF store and was told to put the specific parts of his query first, and by doing this resulted in vast improvement in execution time. They presented an approach to optimizing SPARQL queries by transforming the query to a graph model. If there were unions or joins, they used graph model algorithms to merge the individual graph models into an equivalent graph model. The authors developed rewriting rules to transform the graph models back to SPARQL queries according to heuristics they defined, and if the rewrite could be done satisfying the heuristics, the transformation would take place. The rewriting rules remove contradictory constraints, or constraints which would include all values or exclude all values, which could be a source of many wasted processing cycles, and rearrange
queries to take advantage of RDMS accelerations provided by Jena on which they were
developed. In their experiments, the authors claimed to have seen an 87% improvement in
execution time.

In [36], the authors present an optimization based on query rewriting which relied on
selectivity estimation. They utilize a cost estimation product of the subject, predicate, and object
of a triple pattern. A filter rewrite rule is applied which attempts to extract values from the filter
expression on equality and place them directly in as literals to the triple patterns. The filter
expression is then moved to be executed as early as possible in the query after the all variables
that are used in it are known. Finally, the triple patterns are ordered in increasing order
according to their cost estimation, so the most simple of the triple patterns are solved first. For
queries which utilized filter expressions checking for equivalence, this provided a large decrease
in execution time.

These previous three methods all utilize different methods to rewrite the query in a way
that is faster to execute but is equivalent and returns the same results. This is in contrast with the
method proposed here, where the filter constraints are evaluated throughout the query plan. In
[37], the authors utilize short circuit evaluation, but not for evaluation of constraint conditions on
objects or variables. Instead, they evaluate if a sub-expression can be substituted for an
expression in the evaluation of a relationship. If the sub-expression is cheaper to evaluate and
achieves the same result it is used in place of the parent expression. The expression can subsume
the descendant and an optimization can be achieved from the new expression being cheaper to
evaluate.

SPARQ2L [38], like SPARQLeR, is a proposed extension to SPARQL to support path
searches. It uses a different method for path searches, instead of using regular expressions; it
relies on constructing matrices to discover smaller expressions and then expands to find larger expressions. Though it still defines start and end nodes and could potentially benefit from the proposed optimization for processing the patterns used for discovering start and end nodes for a graph pattern.
4.1 SPARQLeR

SPARQLeR [1] stands for SPARQL Extended with Regular paths, and is an extension of the SPARQL query language to introduce constructs for the discovery of semantic associations. The authors state that a semantic association is an undirected path that connects two entities. In their implementation, the authors added the construct of the path to SPARQL, and the path triple. A traditional triple is a Subject, Predicate, and Object. In contrast the path triple is made up of the Subject, Path, and Object where the path is made up of named relationships and Object between them. The authors’ provided definitions for directed, undirected paths as collection of triples for each object that is contained in the path, and semantic association if an undirected path connects two resources.

There is no change to the base syntax, as existing SPARQL queries will work in SPARQLeR. To differentiate a path variable from other variables, SPARQLeR uses a ‘%’ instead of the ‘?’ used to designate variables in the original syntax. A path triple pattern would be designated like <object> %path <subject>, or variables could be used, as in ?object %path ?subject as a triple pattern. One example given by the authors was:

```
SELECT %path, ?res WHERE {<r> %path ?res}
```

Each element of a path can be constrained with the rdfs:_n, where n is the index of the element in the path starting at 1 with the opening resource 0. Odd numbers are properties, so rdfs:_1 could be used to put a constraint on the first property in the path, and rdfs:_2 would refer to the first resource that is not the start or end resource. The authors used the following example to show a path between resources <r> and <s> which had the first property in the path of <p>.

```
SELECT %path WHERE {<r> %path <s> . %path rdfs:_1 <p> }
```
The following is an example of finding the path and the name of the first author in a path between two publications.

PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT %path, ?name
WHERE {
  %path
  rdfs:_1 ?author .
  ?author
  foaf:name
  ?name
}

More advanced constraints can be put on paths by using regular expressions. An extension of the regex operator from the filter clause was added in the format regex(pathvar, pathexpression, pathflags). The path expression can be formed much like a string regular expression can be, but with property names, inverses, classes, using the prefix abbreviated names for manageability, and the typical operators from regex as described in [1]. The length operator is used to place constraints on the length of the path. It is used as length(pathvar) and the comparison operators (<, <=, =, >, >=) to put constraints on the length of the path, and can be combined to put a lower and upper bound on the path length.
Unlike typical constraints in the filter clause, in SPARQLeR, the length and regex are used in the construction of the path and not a check after the path is constructed. The path is constructed using a breadth first search with a deterministic finite automaton (DFA). The path would only be constructed if it matched the regular expression passed in, as the DFA was compiled from the regular expression.

4.2 BRAHMS Architecture

SPARQLeR utilizes the BRAHMS [25] RDF store. BRAHMS was designed for very fast searches of large ontologies. In order to achieve this it loads the model into main memory using a memory mapped file. Due to using this method, it is read-only. The ontology must first be converted from its RDF format into the snapshot file which is utilized as the memory mapped file. If a change is made, a new snapshot must be made and then loaded, unlike other RDF stores which utilize a relational database for persistent storage, which allows changes to the ontology in real time.

As described in [39], BRAHMS uses separate indexes and hashes for different types of resources, breaking them down into functional models. Specifically, the models used are instance nodes, literals, schema classes, schema properties and statements. Each functional model has its own set of iterators and methods of access. The design decision was made due to keeping the models separate; graph algorithms could focus only on that type of resource. The models utilize the following: values, neighbors, statistics, and validation.

Values are the actual URIs, the namespace, short name, literal value, or the sub/super-classes. As described in the background information, a resource will have a URI, but it will have a value or multiple values assigned to different properties. Neighbors are the iterators which can find other resources depending on what the resource type the iterator belongs to, what parts of
the triple are being searched for, and which parts are being provided. Statistics are used to provide the general information about each of the models, such as the number of instances and the number of relationships among the instances. Validation Boolean methods are used to check if nodes exist in the model and if they are of a given type. The models, which are implemented as classes in C++, are accessed through the RDFGraphModel Class.

One of the most useful pieces of functionality in BRAHMS is that the iterators that can be utilized to find any triple combination for each of the models. An iterator can be used to return every object, predicate and subject in a specific model. Iterators are provided to find each part of a triple based on the other items in the triple that are provided. The methods to return the desired iterators use a naming convention that begins with “get” followed by the model class to which it will apply. The next item in the name is “by” followed by the full spelling of the part of the triple that is being passed as input, or the Subject, Predicate, or Object. The last section of the name refers to the elements that will be returned for each match of the iterator but uses the first letter of the part of the triple. S for the Subject, P for the Predicate, and O for the object. The following are available for each model:

byObject_PS
byObject_SP
byPredicate_SO
byPredicate_OS
bySubject_SP
bySubject_PS
byPredicateObject_S
bySubjectObject_P
bySubjectPredicate_O

The iterator returned by byObject_PS takes the Object id as input, and then iteratively returns each predicate and subject, for every triple in the model used that has an object matching the passed in object. BRAHMS utilizes the BrahmsManager class to provide methods to create and load snapshot files and provides the instances of the RDFGraphModel class for each model that is available for use in applications. The RDFGraphModel class can then be used to retrieve iterators and data from the loaded ontology.

4.3 SPARQLer Architecture

SPARQLer is implemented in C++ by a set of classes and utilizes a parser produced by FLEX [40] and Bison [41]. The classes can be broken down into the following main areas: Query parsing/analysis, query processing, and data storage/results. The main control flow is contained in the Query class. This class calls the parser, the class that creates the query plan, and maintains the flow during the query processing.

The parsing/analysis is handled by the GraphPattern class. The syntax tree produced by the parsing is utilized by this class to produce the list of variables that will be used in analyzing the query; the data structures that will be used to extract the correct information from the RDF data store; and the construction of the data structure which handles the filter constraints. The class also serves as a high level iterator during the query execution, being called for the next match in the query plan.

The overall execution strategy is based on pipelining, a method similar to the pipelining query execution designed for relational databases [42]. The query processing is handled by several iterator classes, which make up the query plan. They can be divided between triple iterators and path match iterators. They use the naming convention to symbolize what they
produce. Starting with what they are initialized with, the number 2 to symbolize that they will be producing what comes after. They use S for subject, P for predicate and O for object.

None2SPOMatchIterator is an iterator that is not provided with any initializing values and produces all triples. This iterator will eventually iterate through all triples contained in the RDF store. Each iterator corresponds to a triple pattern in the query and is used to solve that triple pattern. The full list of triple iterators is as follows:

- None2SPOMatchIterator
- O2PSMatchIterator
- OP2SMatchIterator
- P2SOMatchIterator
- S2POMatchIterator
- SO2PMatchIterator
- SP2OMatchIterator
- SPO2NoneMatchIterator

For example, the O2PSMatchIterator is initialized with a known value or id for an object. The iterator then finds matches for the predicate and subject. All the iterators utilize the BRAHMS iterators depending on the type of data being used, as it might be a literal or an instance node. This is the direct interface that accesses the relationships and data that is represented in the ontology.

The path iterators are used to find paths. Depending on the iterator, they can search for a path between two known resources or use one resource to find an as yet unknown resource to be the start or end node depending on the type of iterator used. The path iterators make use of the BreadthFirstPathIterator which implements the breadth first search algorithm.
BreadthFirstPathIterator makes use of the class SearchTrie, which utilizes the DFA for the allowable patterns, and accesses the BRAHMS methods to access instances and nodes and properties in the ontology. Similarly to the naming conventions for the triple iterators, the path iterators include path to represent the path they produce. The following is the list of path iterators:

- O2PathSMatchIterator
- S2PathOMatchIterator
- SO2PathMatchIterator
- SPO2NonePathMatchIterator

The query plan is implemented as an array of the iterators to match each triple pattern in the query. Each iterator resolves a triple, and possibly a variable which will be used in the next triple pattern iterator to find the next match; they are dependent on the previous iterators for their input so they can produce their output. If an iterator is called to return its next value and it has no more triples it can return, as it returned them already or there were no found matches, the processing backtracks to the prior iterator to return a new match which may provide a new value which will allow the iterator to return more matches. The filter expression is a tree data structure that is contained by the Expression class. It is called at a specific index in the query plan, depending on where it was placed in the query. It is used to check the data against the constraints defined in the query. If it does not return a true, the pattern is not a match, and the processing backtracks to find the next value in the sequence that may be a potential match.

The data storage/results are handled using the VarValue, Value, Triple and Element classes. These data structures store the data from the results returned and retrieve the data for
processing and output. They handle multiple data types and are flexible enough to have additional types added without large scale rewrites of the code.

4.4 SPARQLeR Modification

The prototype of SPARQLeR did not have the full syntax implemented as defined in [1]. The path expression and limit on the path length was set in the filter clause using a function called pathlim(pathvar, pathlimit, pathexpression). A regular expression could be submitted and an upper limit could be placed on the path but the minimum length could not be set. The following is an example of a query from for the glycol ontology from the SPARQLeR prototype.

```sparql
PREFIX glyco:
<http://lsdis.cs.uga.edu/projects/glycomics/2006/GlycO#>
SELECT list(%pathway)
WHERE {
  <http://lsdis.cs.uga.edu/projects/glycomics/2006/GlycO#dolichol_phosphate>
  %pathway
  FILTER ( pathlim( %pathway, 32, 
              ((glycol:has_acceptor_subtrate|glycol:has_reactant)
                glycol:has_product)*" ) )
}
```

Our contributions and modification to the original prototype started at this point. The first extension done was to update SPARQLeR to support the use of the new regex and length operators in the filter expression. The syntax tree produced by the parser of the query already
contained support for length and the extended regex syntax. The modification came in the processing of the syntax tree. The pathlim function already implemented the base functionality of the extended regex and required very little modification. The length constraint for a maximum length was present in the implementation of pathlim as well, so implementing the change to use a length with the < and <= operators was trivial to accomplish. There was no support for minimum path lengths in the existing codebase.

Making the modification to support a minimum length path required modification to all of the path iterator classes and the breadth first path iterator class. A minimum length variable in the classes implementing paths was required; the constructors required updating, and the looping mechanisms of the breadth first path iterator to not break out of its loop for a potential match until the minimum length was met. Once the base functionality was implemented, additional checks were required for relational expressions, which the length operator fell under. The proper use of length as defined was

\[
\text{length}(\%\text{pathvar}) \{<|]||=|>|=\}\ \text{integer}
\]

And when parsed, in the constraints, the original functionality had to remain intact. To overcome this, on relational expression checks involving <,\|<=,\||=,\|>=,\|> operators, checks were conducted if one of the comparators was the length(\%pathvar) call. Using this method, the query plan could be set with the minimum or maximum path lengths for a specific path that was to be used.

When this was done, the query above could be expressed as the following:

```PREFIX gylco:
<http://lsdis.cs.uga.edu/projects/glycomics/2006/GlycO#>
SELECT list(\%pathway)
WHERE {
Minimum paths can now also be set, and the main set of the SPARQLeR syntax can be utilized for discovering paths. The major disadvantage was that if there were any constraints in the FILTER clause on the “to” or “from” node on a path, there was a very large performance hit due to the evaluation of the filter expression after a potential path was identified. By default the filter clause is evaluated after all the triple patterns it involves are discovered. What this causes though, since the path pattern and path length settings are set in the filter clause, the path and triple patterns that are checked against a constraint do not get evaluated until after the path has been tested. The actual path search between two resources can be costly, especially if it is a long path, or does not have or has a very unrestricted regular expression. In order to make execution time acceptable, paths where the start or end node would not meet the filter constraints should not be evaluated before the path search is conducted. This would limit the search space as much as possible. This can be difficult if you are required to have constraints for resources for which one may not know all values for before the path is to be constructed if one of the nodes is discovered by the path search.
4.5 Optimization

The solution proposed and we implemented here is to use a form of short circuit evaluation in order to use the known variables at each triple discovery, and at that point, evaluate if the triple can be eliminated from consideration to limit the number of resources used in path searches. For example, if there were three variables being used in a filter clause that traditionally is evaluated after a path search, and before the path search is executed, only one of the variables referenced in the expression is known, if the value is such that the expression would fail despite the values of the other variables, then that search could be avoided. Besides the intended use to avoid unnecessary path searches, this method can also be an aid to non-path queries as any instance where a sizable portion of a search space can be avoided; you would likely see a decrease in execution time as unnecessary paths are not followed.

To implement this optimization, a binary tree like data-structure was used. The leaves contained the constant values, or variable names which were used in the expressions. The nodes identified the operation that it represented (+,-,*,/,&,|,\,<,<=,=,>,!,regex). During evaluation, a structure containing the known variable names and their values is passed in. The expressions are evaluated in a depth first fashion. The aim was to eliminate potential search spaces so in the instance of an unknown, the assumption is true. This way, elimination on a return of false for the expression will only happen when it is a true false and not estimated. If a variable is unknown, the value propagated up from any comparison it is involved in is true. In addition, a flag to represent that the value was reached due to an unknown value is set as well. This is needed to be done in the event a “not” operation is performed. If a negation operation was performed on a true from an undetermined value, the false would not be accurate since the real evaluation would not be known, and so the true would still propagate up.
For an “and” node, if the left branch resulted in a false, the false would be sent to the parent, with no more evaluation, since one false in an “and” comparison will result in the expression being false. On true, the right branch would be evaluated. The flag for an unknown variable would only be passed up if both sides were true and one of the branches was the result of an unknown variable.
Figure 2 (And Decision Tree)

For an “or” node, if the left branch was identified as true and not the result of an unknown variable, true would be passed to the parent node. If the left branch was identified as false or as an unknown variable, then the right branch would need to be evaluated at that point. If both branches were false, false would be propagated to the parent node. If the right branch was true without an unknown variable, true would be passed to the parent. If both branches had unknown variables, true and the unknown variable flag would be returned to the parent node.
During the evaluation of the syntax tree from the parser, a tree of the filter clause is constructed using the structure whose behavior is described above. Using this tree once it is constructed, any value of any variable or combination of them could be used to test if it would disqualify a triple pattern return at that point. The complexity of the filter expression does not matter. If the value of a variable can result in a false, and no parts of the overall expression need to be moved or rearranged using this strategy.

When evaluating a query, SPARQLeR constructs a query plan, which is an array made up of different types of triple iterators which are selected according to the type of triple pattern
entered in the query, as discussed in the architecture section. When the query is being executed, the index starts at zero and utilizes the first iterator to retrieve the first match for that triple pattern. If there was no match, as that was the first pattern, it would return with no matches. If there is a match, the index increments and checks if it is less than the plan size. If it is a filter index it would evaluate the filter expression. If it is less than the match size and is not a filter index, it then gets the match for the iterator at the current index. If there is no match, the index backtracks and retrieves the next triple from the iterator at that index. If the index hit the plan size and the filter expression was true, a match was found and recorded. If, during backtracking, the index goes less than zero, there are no more matches and processing ends.

The short circuit optimization was inserted into this process by creating an array of triples that is equal to the plan size. Each time an iterator finds a match, the triple it finds is used in the array at the index of the triple. When the optimization is used, before the index is incremented to discover the next match, the short circuit evaluation is conducted by extracting the known variables from the triples contained in the solved triple array from 0 to the current index. The evaluation of the tree structure described above is conducted. On false, the index is not incremented, and as a result, the iterator at the current index will retrieve the next match, the prior match being ignored. Thus, with this optimization, the first time a disqualifying value for a variable is entered; it will not propagate up past that triple pattern, and instead will be replaced by the next triple in that sequence.

Most of the triple iterators rely on the variables discovered by prior triple patterns, and by the elimination of one, is where the time savings can materialize. If a variable returned from one of the triple iterators can be thought of as a street one would turn onto from a highway, a variable that results in a false is a dead end road. One can sometimes travel a long distance on it
before one realizes one has made a wrong turn and then have to backtrack to get on a good road. This strategy is very similar in that it avoids all dead ends at the earliest possible time one could determine the road would result in a dead end. Execution time is saved by not traversing the dead end which is the optimization. The limitation of this strategy is the overhead it adds to check every pattern match returned by the triple iterators.
CHAPTER 5
EVALUATION

To conduct an evaluation of the effectiveness of the optimization, it was decided that a large data set should be utilized. The evaluation was conducted using a set of queries to compare the performance of the optimization against un-optimized SPARQLeR using a set of non-path queries of increasing size, and a set of path queries evaluating filter based variables for the start and end nodes of path searches. In this way, the overall usefulness can be determined as well as avoiding inconsistencies in execution time that can sometimes be experienced with smaller evaluations.

The tests were conducted on a system with 2 Intel® Xeon™ 3.06 GHz CPUs, with 4 GB RAM, running RedHat Linux Enterprise. Code was compiled using gcc (GCC) 3.2.3 with the ‘-O6’ optimization flag. All tests were executed 4 times each using SPARQLeR without the optimization and with it.

5.1 Data Sets

SwetoDblp [43] was selected due to its size and opportunity to construct path expressions. The ontology has a relatively simple schema and is the collection of bibliographical information of computer science publications. As of August 2007, the ontology contained over 3 million literals, 2.3 million resources, 3.7 million resource-to-resource triples and 7.2 million resource-to-literal triples.
<table>
<thead>
<tr>
<th><strong>Number of Entities</strong> (in main classes)</th>
<th><strong>Number of Relationships</strong> (in main relationships)</th>
</tr>
</thead>
<tbody>
<tr>
<td>560,792 Person (foaf:Person)</td>
<td>900,440 publication-has-author (author)</td>
</tr>
<tr>
<td>561.895 Articles in Proceedings</td>
<td>438,531 contained in proceedings (isIncludedIn)</td>
</tr>
<tr>
<td>340,488 Journal Articles</td>
<td>112,303 cites publication</td>
</tr>
<tr>
<td>10,610 Webpages of persons</td>
<td>10,639 has-homepage (foaf:homepage)</td>
</tr>
<tr>
<td>9,027 Proceedings</td>
<td>10,461 has-publisher (dc:publisher)</td>
</tr>
<tr>
<td>2,530 Book Chapters</td>
<td>5,850 in series</td>
</tr>
<tr>
<td>1,235 Books</td>
<td>7,308 has affiliation (foaf:workplaceHomepage)</td>
</tr>
<tr>
<td></td>
<td>2,013 owl:sameAs (between people)</td>
</tr>
</tbody>
</table>

Table 1 (From http://lsdis.cs.uga.edu/projects/semdis/swetodblp/)

To test the scalability of the optimization, a modified version of the DBLP ontology was used. This was the same dataset used in [1] for evaluation of SPARQLeR’s scalability. This ontology has 790,635 publications with a published year attribute. As in the referenced paper, the ontology was divided up into 26 subsets made up of articles published in each year from 1981-2006 with 760,369 publications. The smallest RDF representation being 2006, and each subset adds the publications for the next year to the set as a whole, so each iteration subsequently has more publications, and thus the same ontology is used, but the instances increase.

**5.2 Non-Path Evaluation**

For testing, queries were developed which used the original SPARQL language use of triple patterns, without the use of the extension of path triples, to evaluate the usefulness of the optimization whether path triples were in use or not. The strength of the optimization revolves around eliminating potential paths from the search space as early as possible. The test queries were made to produce graph pattern triples starting from graph patterns of a shorter length and increasing. The evaluation was designed to test graph patterns from 2 nodes, increasing to 6.

The starting point was to use a graph pattern of two basic nodes, an author, and an article. The edge between the nodes is the author property between the article node and the person node.
To take advantage of filtering, a constraint was placed on the year field of the article, where the only articles that would be considered were those published from 1983-1984, with this first query returning a result set of all articles published within the stated date range and their authors’ group individually. The next query was an iteration of the first. This added to the graph an additional author, who was listed on the publication. A constraint was added to ensure the name of the authors was not the same to prevent the same person being listed in the pattern. The result returned for this query being a 3 node graph pattern of Author, Article, Author, where the two authors wrote the article.

The next iteration, bringing the pattern to 4 nodes, added an additional article, with the year constraint of being published between 1993 and 1994 and that one of the authors was the second author in the graph pattern. The next query added an author, who co-wrote the article added in the previous query, with the constraints that the author’s name would not be equal to the other two authors contained in the pattern. The final query in this sequence contained the addition of an article written by the previous author that was published between 2003 and 2004, bringing the graph pattern to 6 nodes for evaluation.

The execution time for optimized vs. non-optimized favored the unmodified SPARQLLeR implementation for 2 and 3 node graph patterns returned in the queries. The average execution time for the optimized on the first test was 18 seconds, while the non-optimized completed in 13 seconds. For 3 nodes, the optimized average was 62 seconds and the original 49 seconds. From that point on however, the optimization completely outperformed the original implementation. Only one result was run for 6 nodes with non-optimized as the time of execution exceeded three days. The extent to which the optimization outperformed the original implementation was
surprising. While there was an expected decrease in time, the results observed were extreme. See figure 4 for the results.

Figure 4 (Execution Time for Non-Path Queries)
The overhead for lower numbers of nodes is somewhat concerning, as the execution was approximately 25% slower for two nodes, as seen in Figure 4. Due to the large dataset, this may be overlooked as a two node graph pattern is not normally considered interesting. The explanation for the slowdown is that the total number of triples considered by both implementations was the same so the short circuit evaluation did nothing to shorten the search space and was only adding to the execution time with the overhead of its checks. Two nodes may be the worst case scenario when this optimization is in use.
Also worth noting is that the speed up increased with the total number of matches the query returned. Observe figure 6. The test with six nodes had a much higher number of matches than any of the other tests.
5.3 Path Evaluation

To test Path triples, constraints needed to be tested on the start and end nodes of path triple, and so four queries were used. To start with a baseline, a query was developed which tested finding a path between two constant nodes, an author and a publication; the second query was made to test a variable to node; the third a variable from node; and the fourth to test a variable from and to node. The first query found all paths of less than 8 which were made up of the relationships of author or citation of an article. The second tested with the same constant author starting node but with a constraint based publication, where the article had a known author and was published after 2006. The same path pattern was utilized (author and citation), but paths were limited to less than 9. The next query used a fixed publication, and a variable author set to authors who co-authored an article with Jeffrey D. Ullman after 2004. The same path pattern is used but the length of the path is restricted to between 4 and 7. The final path test query uses a variable from and to node. The from node is made up of authors who published an article with Jeffrey D. Ullman after 2004, and the to node is made up of publication authored by John A. Miller after 2006. The same path pattern is utilized with the path length limited to 7.

As expected, the optimization outperformed the original implementation over the path query tests. On the baseline test, the results were the same, but from that point on the clear advantage went to the short circuit evaluation. The results have still not been observed for the original implementation for the fourth test, which was the use of a variable from and to node. See figure 7 for the results.
The speed up, listed in Figure 8, was extremely high, but this was expected due to the nature of the optimization. The results show that using constraints for identifying a start or end node in a path search is usable at this point.
5.4 Scalability Evaluation

To test the scalability, the modified DBLP ontology with incrementally increasing sets of instances was used to evaluate two queries over the datasets, evaluated four times each. Similar to the original scalability test, the first query is a single-source query, where the start node is defined, but as a departure, instead of using direct URI references to the start publications and having multiple queries for each publication used, a set of triple patterns and filter constraints were utilized to construct a set of publications that was used to initialize the start node of the path. The URI of an author was used and then all articles in a date range by that author were used as the start nodes. The SPARQLεR query that was used is presented below:

```
PREFIX opus: <http://lsdis.cs.uga.edu/projects/semdis/opus#>

Select ?publication list(%path) ?endpub

Where {?
    opus:year
    ?year .

    ?publication
    opus:publication_authored_by <http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/Benini:Luca.html> .

    ?publication
    %path
    ?enpub

    FILTER( regex(%path, "(opus:cites_publication)\*") &
        length(%path) < 6 &
        ?year > 2005 ) }
```
In contrast with the original tests, the execution time appears to follow the exact same trend as the number of paths found in the search. This suggests a linear relationship between the number of paths found, and the execution time. An exponential increase in execution time would normally be expected. This could be due to the 5 hop maximum that was set. The execution time and number of matches averaged over the fourth trial runs for each year set is plotted in Fig. 9.

![Figure 9 (Exe. Time and Path Match for Single-Source Queries)](image-url)
When comparing the execution time to the non-optimized code, there was a large increase in the speed up ratio of the optimized vs. the non-optimized from the start, but then in the 2006-1997 data sets until the 2006-1992 dataset, the speed-up ratio had a steep decrease before increasing again. Observe Figure 10 bellow.

![Execution Time for Single-Source Non-Optimized Queries](image1)

![Speed Up Ratio of Optimized vs. Non-Optimized Single-Source Queries](image2)

**Figure 10 (Exe. Time of Single Source Non-Optimized Query and Speed Up Ratio)**

The final test was conducted with triple patterns and filter constraints to construct a set of both start and end nodes for the path search. A different author and similar filter constraint are
used for the start node as what was used in the first scalability test. A different author was
utilized for the end node, as well as a constraint on the year of publication of articles by that
author. The SPARQLer query that was used is presented below:

PREFIX opus: <http://lsdis.cs.uga.edu/projects/semdis/opus#>
Select ?publication list(%path) ?endpub
Where {?publication
  opus:year
  ?year .
?publication
  opus:publication_authored_by
  <http://www.informatik.uni-trier.de/~ley/db/indices
   /a-tree/b/Beneson:Zinaida.html> .
?endpub
  opus:publication_authored_by
  <http://www.informatik.uni-trier.de/~ley/db/indices
   /a-tree/v/Vehreschild:Andre.html> .
?endpub
  opus:year
  ?endyear .
?publication
  %path
?enpub
FILTER( regex(%path, "(opus:cites_publication)*") &&
The number of paths found increased exponentially with the number of papers that were included in the ontology. The execution time for the query and the number of matched paths averaged over the four trail runs is plotted in figure 11.

Figure 11 (Exe. Time and Path Match for Def. Start and End Node Queries)
This was very similar to the results of the original SPARQLeR test, and shows that with the optimization the implementation still scales well. One point that is worth discussing is the use of shorter hops than the 26 that were used in the original scalability tests. For the tests that were conducted, the execution time was extremely large, and the memory size for the search space was insufficient to get a return on the larger year sets. The original tests were utilizing
directed paths, where the relationship was only following one direction. All tests used in these tests were undirected path searches. Relationships were captured with indifference to the domain and range of the property.

When compared to the execution of the non-optimized implementation, some very interesting observations were made. The optimized implementation was slower until the 2006-1994 dataset, at which time a faster execution time was achieved. From that point, the speed up ratio increased at an exponential rate until the 2006-1987 data set, hitting a peak of a 630% speed up, before rapidly approaching a 100% speed up for the remainder of the data sets. See Figure 12 for the results.

**Figure 12** (Exec. Time of Def. Start and End Nodes Non-Optimized Query and Speed Up Ratio)
In evaluating the cause for the rapid decreases in the speed up ratio between the optimized and non-optimized execution for both single-source and defined start and end nodes, the following conclusions were reached. In the queries, the publications, used for the nodes, were produced from all publications a specific author had written with in a constraint placed on the year. The authors selected had a very limited total set of papers published. In the single source queries, the peak number of potential nodes was reached around in the 2006-1994 data set. From that point on the total number of nodes was not being reduced in further tests. The speed up from that point was dependant on the cost of the unnecessary path searches, but the total number of unnecessary searches would not change at that point. The optimization did see the ratio increasing at that point and that could be due to the cost of the path searches increasing from that point on.

In the defined start and end node query results, the searches were much more constrained due to the definition of the beginning and the end of the path. The path searches were faster, and the penalty for unnecessary path searches could have been smaller than the overhead of the optimization, leading to the slower executions until the 2006-1994 data set was reached. At that point, the penalty for the unnecessary path searches began approaching equilibrium with the time to search for good paths. The total number of articles in the ontology by the author used for the start node at 2006-1981 was 6. 4 were isolated by the constraint in 2006. The other author only had a total of 3 publications in the ontology which were fully utilized by the end node. Out of a total of 18 possible combinations for the start and end nodes without the constraints, only 6 were unnecessary, so consistently running at less than half the execution time is an acceptable performance increase, and the success of the optimization is still demonstrated.
CHAPTER 6
CONCLUSIONS AND FUTURE WORK

With path queries, the results matched what was expected that without the short circuit evaluation, using constraint based nodes in conjunction with path searches was not practical. In the original implementation, if one of the nodes was not a literal resource (and the resource was identified from a triple pattern where a constraint was applied to it), all values that could be provided by the iterator being used would be tried against the path search since no filtering was done until all processing was completed as a final check. With the optimization in place, using triple patterns to discover the start and/or end nodes have acceptable execution times, and allowing more convenient means of constructing queries with multiple start or end nodes.

For non-path queries, the results show that for relatively small graph patterns, specifically graph patterns that use only one constraint and are three nodes or less, the overhead will result in slower execution times. Despite the overhead of doing an evaluation at each triple pattern found, the reduced execution time for triple patterns that have filter constraints on multiple variables, and have graph patterns with four or more nodes see very large improvements as the result of eliminating large sections of the search space with even small number of linked nodes, as evidenced by the non-path query tests. Evaluation of constraints after all search operations is very inefficient, and the approach presented here has demonstrated the advantage of early evaluation. The queries with more than three nodes to their graph pattern were nearly unusable using the original implementation. The optimization has shown its usefulness and effectiveness, achieving execution times, in many cases, thousands of times faster than without.

For the scalability test, SPARQLeR with the optimization showed that it could successfully perform fast path searches in large ontologies. The largest potential benefit of the
optimization was displayed, where triple patterns were utilized to define a set of start and end nodes to conduct a path search. Prior work to query for suitable start or end nodes and then pasting them into a query was not needed. SPARQLeR can be more fully utilized due to the optimization.

SPARQLeR is still not fully implemented according to the definition in [1]. The flags for the regex function need to be fully implemented, which would allow a path search to include schema properties, selection of whether the search will be directed or undirected, and the regular expression syntax needs to recognize inverse relationships. The first priority of an implementation should be to address this. Another future improvement could be to extend SPARQLeR to become fully SPARQL compliant. This would involve implementing the OPTIONAL and UNION functionality. Also, in a fully compliant standard SPARQL implementation, multiple filter constraints can be added in a query, and constraints can be entered after each triple pattern, if desired. The constraints separated in this way can be thought of as having a Boolean AND relationship between them since they all must be satisfied.

If all constraints in a filter expression are compared to each other with a Boolean AND operation, each constraint can be separated out to the specific variable or after the two variables that it applies to. Of course with complicated constraints that include multiple or expressions nested with and expressions, this version of short circuit evaluation would still be the most practical solution for implementing constraint checking.

Another interesting extension would be to combine the short circuit evaluation, contained in this work, with a method like [36]. This work could be used to take the triple patterns and arrange them so that the least complex are evaluated first, followed by nodes with simple constraints which can have bad values eliminated early in processing, and then the expressions
can be evaluated in areas where individual constraints could not be separated out to apply to each pattern. Using this method, a person writing a query would not have to have advanced knowledge of writing queries efficiently, as the construction of the query plan would result in the most efficient means of execution.
REFERENCES

